# Testing SPIN's LTL Formula Conversion into Büchi Automata with Randomly Generated Input

Heikki Tauriainen and Keijo Heljanko*

Helsinki University of Technology,
Laboratory for Theoretical Computer Science
P. O. Box 5400, FIN-02015 HUT, Finland
{Heikki.Tauriainen, Keijo.Heljanko}@hut.fi

**Abstract.** The use of model checking tools in the verification of reactive systems has become into widespread use. Because the model checkers are often used to verify critical systems, a lot of effort should be put on ensuring the reliability of their implementation. We describe techniques which can be used to test and improve the reliability of linear temporal logic (LTL) model checker implementations based on the automata-theoretic approach. More specifically, we will concentrate on the LTL-to-Büchi automata conversion algorithm implementations, and propose using a random testing approach to improve their robustness. As a case study, we apply the methodology to the testing of this part of the SPIN model checker. We also propose adding a simple counterexample validation algorithm to LTL model checkers to double check the counterexamples generated by the main LTL model checking algorithm.

## 1 Introduction

Model checking of linear temporal logic (LTL) properties can be done using the automata-theoretic approach [15]. This model checking method employs a translation of properties expressed in LTL into finite-state automata over infinite words (Büchi automata), which are then used to determine whether a given system model satisfies a given LTL property.

An essential requirement for the correctness of model checking results is the correctness of the model checker implementation itself. Therefore, a model checker which has itself been verified or proved correct would certainly be a tremendous advantage in ensuring the correctness of the verification results. However, full verification of complex software of this kind – especially when implemented in an imperative general-purpose programming language such as C – is somewhat out of reach of current software verification techniques. Nevertheless, even methods for only partially improving the robustness of model checkers would still be welcome.

We will propose a method for testing and improving the robustness of a part of a model checker. We focus on the translation of LTL formulas into Büchi

* The financial support of Academy of Finland (Project 47754), the Emil Aaltonen Foundation and the Nokia Foundation are gratefully acknowledged.

automata (LTL-to-Büchi conversion), which seems to be among the most difficult steps in LTL model checking to implement correctly. The method is based on the comparison of several implementations against each other. As a concrete example, we will test the LTL-to-Büchi conversion implementation in the model checker SPIN [6, 5] using a random testing methodology. This work can be seen as a continuation of the work published in [13] with the following extensions:

- We describe a systematic method of using the results of the comparison (together with a single behavior in the system model) to find out the particular implementation which failed, an improvement over simply detecting result inconsistencies. The method is based on model checking an LTL formula directly in the witness behavior by applying computation tree logic (CTL) model checking techniques. This special case of model checking LTL in witness behaviors is based on the ideas first presented in an extended version of [8].
- We discuss an additional application of the previous technique in LTL model checking tools for validating counterexamples.
- We extend the test procedure for LTL-to-Büchi conversion algorithm implementations to make use of non-branching state sequences (paths) as reachability graphs, taking advantage of some additional checks provided by the direct LTL model checking method in comparing the behavior of the implementations.
- We present experimental results on using the test procedure on up-to-date versions of SPIN.

The rest of this paper is organized as follows. In Sect. 2, we describe the general testing procedure for comparing the results produced by different LTL-to-Büchi translation algorithm implementations with each other. Section 3 introduces a practical implementation of the test procedure into a randomized testbench for LTL-to-Büchi translation algorithm implementations. In Sect. 4, we propose a method for validating counterexamples provided by LTL model checkers. Section 5 reports the results of using the test procedure on the LTL-to-Büchi translation algorithm implementations of different versions of SPIN. Conclusions with some directions for future work are presented in Sect. 6.

## 2 Testing Procedure

In order to test the correctness of different LTL-to-Büchi conversion algorithm implementations in practice, the test procedure itself should be efficient in finding errors in the implementations. It should also be as reliable and simple as possible to avoid errors in the test procedure implementation itself.

Testing LTL-to-Büchi conversion algorithm implementations requires input for the implementations, i.e. LTL formulas to be converted into Büchi automata. We could try to test the correctness of a single implementation by using it to construct automata from an LTL formula and its negation and then checking whether the intersection of the languages accepted by these two automata is

empty. (This could be checked with the help of a synchronous composition of the two automata, see e.g. [14].) If this result is found to be nonempty, we can conclude that the implementation does not work correctly.

However, this simple method has some disadvantages. First of all, if only a single implementation is used, this check is not sufficient to show that the obtained automata actually represent the formula and its negation correctly. (As a trivial example, this check would not detect the error in an otherwise correct Büchi automaton generator which always negates every input formula before generating the automaton.) We can gain more confidence in the correctness of the implementation by performing the language emptiness check against an automaton constructed for the negation of the formula using another independent conversion algorithm implementation. Therefore, if we have several implementations available, we can use each of them to construct automata from the formula and its negation and then check whether all the pairwise synchronous compositions of some automaton constructed for the formula and another automaton constructed for its negation are empty.

Another problem with this approach is that we cannot use the results of the emptiness checks to infer the correctness of any implementation (even on a single LTL formula) even if no failures are detected. Namely, implementations with errors may generate automata which pass all these checks but which still do not accept the exact languages corresponding to the input formulas. (For example, an implementation which always generates the empty automaton regardless of the LTL formula would always pass the emptiness check with any automaton.)

The test method's reliability in this respect could be increased by extending the procedure with one additional step. In addition to checking the emptiness of the intersection of the languages accepted by a pair of automata constructed from the formula and its negation, we could also check that the *union* of these languages (once again representable as a Büchi automaton, see e.g. [14]) forms the universal language. However, this problem is in general PSPACE-complete in the size of the resulting Büchi automaton, see e.g. [14]. In principle, the universality check can be done with a Büchi automata complementation procedure. However, this is a potentially hard-to-implement task [11], contradicting the goal of trying to keep the test procedure implementation as simple as possible.

In practice, it would also be an advantage if the test procedure were able to give some justification for the incorrectness of an implementation instead of a simple statement that the checks failed without exactly revealing which one of the implementations was in error. For instance, a concrete example of an input incorrectly accepted or rejected by some automaton might be of help in debugging the implementation with errors.

As a compromise between the difficulty of implementation and the reliability and practical applicability of test results, we will not try to compare the obtained automata directly. Instead, we use a testing method based on the automata-theoretic model checking procedure for LTL, see e.g. [4, 14, 15]. Basically, this involves model checking LTL formulas in models of systems and then checking that the model checking results obtained with the help of each individual

LTL-to-Büchi conversion algorithm agree. Although this testing method requires more input (the system models) in addition to the formulas, it can also easily provide counterexamples for confirming the incorrectness of a particular implementation if the verification results are inconsistent. In addition, all steps in this test procedure are relatively straightforward to implement.

In this section, we discuss the general testing procedure and the methods for analyzing the test results. Since the methods for generating input for the testing procedure are independent of the procedure itself, we will leave the description of some possible methods for input generation until Sect. 3.

### 2.1 Automata-Theoretic Model Checking

We assume the system model to be given as a finite-state reachability graph, an explicit-state representation capturing all the behaviors of the system as infinite paths in the graph. The properties of the system are modeled using a finite set $AP$ of atomic propositions, whose truth values are fixed independently in each state of the graph.

Formally, the reachability graph is a Kripke structure $K = \langle S, \rho, s_0, \pi \rangle$, where

- $S$ is a finite set of states,
- $\rho \subseteq S \times S$ is a total transition relation, i.e. satisfying $\forall s \in S : \exists s' \in S : (s, s') \in \rho$,
- $s_0 \in S$ is the initial state and
- $\pi : S \mapsto 2^{AP}$ is a function which labels each state with a set of atomic propositions. Semantically, $\pi(s)$ represents the set of propositions that hold in a state $s \in S$, and the propositions in $AP \setminus \pi(s)$ are false in $s$.

The executions of the system are infinite sequences of states $s_0 s_1 \ldots \in S^\omega$ such that $s_0$ is the initial state of the system and for all $i \geq 0$, $(s_i, s_{i+1}) \in \rho$.

By using the function $\pi$ for labeling each state with a subset of $AP$ including exactly the propositions which are true in the state, the infinite behaviors of the system can alternatively be represented as infinite strings of state labels.

The task of the conversion algorithms we wish to test is to expand a given LTL property expressed as a formula over $AP$ into a Büchi automaton. This automaton is supposed to accept exactly those infinite strings of state labels (system behaviors) which are models of the formula.

Formally, a Büchi automaton is a 5-tuple $A = \langle \Sigma, Q, \Delta, q_0, F \rangle$, where

- $\Sigma = 2^{AP}$ is an alphabet,
- $Q$ is a finite set of states,
- $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation,
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is a set of accepting states.

An execution of $A$ over an infinite word $w = x_0 x_1 x_2 \ldots \in \Sigma^\omega$ is an infinite sequence of states $q_0 q_1 q_2 \ldots \in Q^\omega$ such that $q_0$ is the initial state and for all $i \geq 0$, $(q_i, x_i, q_{i+1}) \in \Delta$.

Let $r = q_0 q_1 q_2 \ldots \in Q^\omega$ be an execution of $A$. We denote by $\inf(r) \subseteq Q$ the set of states occurring infinitely many times in $r$. We say that $r$ is an *accepting* execution of $A$ iff $\inf(r) \cap F \neq \emptyset$.

The automaton accepts an infinite word $w \in \Sigma^\omega$ if and only if there is an accepting execution of $A$ over $w$.

The goal is now to check whether the reachability graph and the Büchi automaton have any common behaviors, i.e. whether any infinite behavior of the system is accepted by the Büchi automaton. In LTL model checking, the answer to this question is used to confirm or refute whether the system satisfies a given LTL property $\varphi$. This actually involves using an automaton $A_{\neg\varphi}$, constructed for the *negation* of the property. Since the model checking of LTL properties requires *all* system behaviors to be considered, any system behavior accepted by $A_{\neg\varphi}$ is sufficient to falsify the given property.

However, in our testing approach, we are not interested in making any such conclusions about the system as a whole. For our needs, it is sufficient to use the LTL-to-Büchi conversion algorithms to construct an automaton $A_\varphi$ simply for the given property $\varphi$. In effect, we will be using the constructed automaton for actually model checking the CTL* formula $\mathbf{E}\,\varphi$ instead of the LTL formula $\varphi$ in the system [7].

Actually, the reachability graph can be seen as a Büchi automaton whose every state is accepting. This allows checking the existence of common behaviors by computing the synchronous product of the reachability graph with the property automaton (see e.g. [4]). Intuitively, this corresponds to enumerating all the parallel behaviors of the system and the automaton such that the behaviors agree at each step on the truth values of the atomic propositions in the property formula, when both the system and the automaton start their execution in their respective initial states.

Computing the synchronous product (for details, see e.g. [13]) results in another finite-state nondeterministic Büchi automaton having an accepting execution (specifically, a cyclic state sequence with an accepting state) if and only if any infinite system behavior is accepted by the property automaton. The existence of an accepting execution is equivalent to the question whether any nontrivial maximal strongly connected graph component with an accepting state can be reached from the initial state of the product. Determining the answer to this question can be done with the well-known algorithm due to Tarjan [12] for enumerating the maximal strongly connected graph components in linear time in the size of the product automaton.

### 2.2 Result Cross-comparison

The existence of system behaviors satisfying the property should not depend on the particular Büchi automaton used for computing the product. However, errors in the LTL-to-Büchi conversion algorithm implementations may result in incorrect Büchi automata.

The basic test objective is to use the different LTL-to-Büchi conversion algorithm implementations to obtain several Büchi automata from a single LTL

formula. Each of the automata is then synchronized separately with the reachability graph, and the results are checked for the existence of accepting cycles. Assuming that all the used LTL-to-Büchi implementations are error-free (in which case all the obtained automata will accept the same language), this check should always produce the same set of system states from which an accepting cycle can be reached. Any inconsistencies in the results suggest an error in some conversion algorithm implementation. We will show in Sect. 2.4 how we distinguish the incorrect implementation from those for which the results are correct.

The synchronization of the system with the property automaton is often performed only with respect to the unique initial states of the two automata. However, we proceed by synchronizing the property automaton separately in every state of the reachability graph, i.e. considering each state of the reachability graph in turn as the initial state. In this way, we can obtain more test data for comparison from a single reachability graph. The additional computation corresponds to checking the existence of behaviors satisfying the property in *every* state of the system, instead of only in its initial state. We call this the *global synchronous product*.

It is straightforward to extend the standard method for computing the product to perform all the synchronizations simultaneously. Basically, this requires ensuring that the initial state of the property automaton is paired with each state of the reachability graph while maintaining the product closed under its transition relation. This approach also takes advantage of the possible sharing of substructures between the different product automata, resulting in the same worst-case result size ($|Q| \cdot |S|$) as in the single-state method. (The actual occurrence of the worst case is, however, certainly much more probable with this approach.)

The check for accepting cycles is also easily extended to the global product. The only difference here is that the product automaton now has more than one state in which the reachability of any accepting cycle must be checked.

As a result, we obtain for each Büchi automaton conversion algorithm implementation a set of yes/no answers to the question of the existence of behaviors satisfying the LTL formula beginning at each state of the reachability graph. Assuming the Büchi automata were constructed from an LTL formula $\varphi$, these answers correspond to the truth values of the CTL* formula $\mathbf{E}\,\varphi$ in each system state.[1] These answers can then be compared against each other in each individual state as described above.

### 2.3 Checking the Results for Consistency

Once the previous testing procedure has been completed for a single LTL formula, it is useful to repeat the whole procedure again for the negation of the same formula. In addition to another result cross-comparison check, this allows

[1] By exploring the global synchronous product with an on-the-fly version of Tarjan's algorithm, we could thus use the approach also as an efficient LTL model checking subroutine for an on-the-fly CTL* model checker. (See [1].)

a simple consistency check to be performed on the two result sets obtained using each individual LTL-to-Büchi conversion algorithm implementation.

The check is based on the fact that no system state can simultaneously satisfy an LTL property *and* its negation, even though both may well remain *unsatisfied* in the state. (For an LTL property, this will occur if there are several behaviors beginning at the state, some of which satisfy the property individually, while others satisfy its negation.) However, the *nonexistence* of a behavior satisfying the LTL property implies that the negation of the property holds in that state. Therefore, it follows that the answer to the existence of an accepting execution cannot be negative for both the formula and its negation in any state, as that would imply that both the property and its negation hold in the state at the same time.

This check is very easy to perform on the two result sets obtained using a particular LTL-to-Büchi conversion algorithm implementation. Any violation of the previous fact immediately confirms an error in the implementation.

The basic steps of the test procedure are illustrated in Fig. 1.

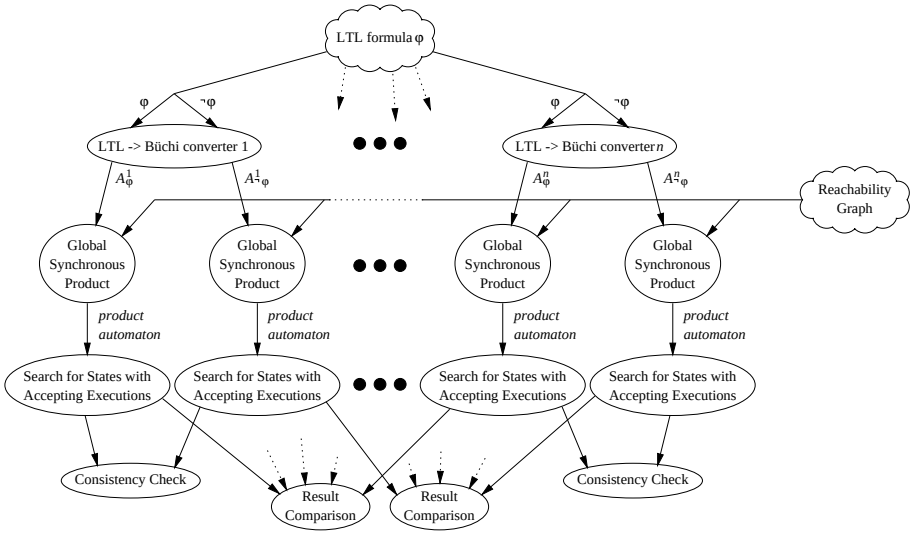


**Fig. 1.** Basic test procedure

### 2.4 Cross-comparison Result Analysis

The result cross-comparison check is not in itself enough to reveal the implementation or implementations in error. Of course, running several independent implementations against each other may help to single out the incorrect one if a pattern can be detected in the occurrence of result inconsistencies, e.g. when the

implementations generally agree with each other with the exception of a single implementation which sometimes fails the check with all the other implementations.

However, simply searching for patterns in the detected inconsistencies may not be an adequate strategy if the used implementations are not independent, or if in some cases only few of the many tested implementations are actually correct, while all the others give incorrect results. Therefore, we will need more systematic methods to distinguish the incorrect implementation from the correct ones.

Our approach for detecting the implementation with errors is based on first finding (see below) a concrete example behavior of the system (i.e., a witness) for which the model checking results obtained using two different LTL-to-Büchi conversion algorithm implementations disagree. The LTL formula is then model checked in this example behavior separately. Instead of using for this purpose a general LTL model checking algorithm, whose implementation correctness would be equally difficult to confirm as that of the implementations we wish to test, we use a *direct LTL model checking algorithm for a single state sequence*. This approach is based on the intuition that a restricted algorithm is easier to implement correctly than any of the tested translation algorithms and should therefore be more reliable. Using this algorithm to model check the formula separately in the example behavior, we obtain a strong suggestion about which one of the tested LTL-to-Büchi conversion algorithm implementations had failed.

The first step of the analysis is to find a system state in which the result sets obtained using two LTL-to-Büchi translation algorithms disagree. In this state, another result set claims the existence of an infinite system behavior satisfying the LTL formula. We use the product automaton associated with this result set to extract an actual infinite system behavior which is accepted by the Büchi automaton used for constructing the product. This can be done using the standard model checking techniques for extracting counterexamples (actually witnesses in our case), see e.g. [2].

It is important to note that the obtained infinite system execution always consists of a prefix of system states followed by a state cycle which repeats infinitely often. Another important property of the execution is that each of its states has exactly one successor. The execution is therefore an infinite path in the reachability graph. These characteristics of the execution allow us to apply CTL model checking techniques for evaluating the given LTL formula in the execution.[2] (We use the standard semantics for CTL, see e.g. [7].)

[2] This idea was first presented as a remark in an extended version of [8], available at <URL: `http://www.cs.rice.edu/%7Evardi/papers/`>.

Let $\tau$ be a function mapping LTL formulas to CTL formulas, defined recursively as follows:

$$\begin{aligned}
\tau(\textsc{True}) &= \textsc{True} \\
\tau(\textsc{False}) &= \textsc{False} \\
\tau(P) &= P \text{ for all atomic propositions } P \in AP \\
\tau(\neg\varphi) &= \neg\tau(\varphi) \\
\tau(\varphi \vee \psi) &= \tau(\varphi) \vee \tau(\psi) \\
\tau(\mathbf{X}\,\varphi) &= \mathbf{AX}\,\tau(\varphi) \\
\tau(\varphi\,\mathbf{U}\,\psi) &= \mathbf{A}\big(\tau(\varphi)\,\mathbf{U}\,\tau(\psi)\big)
\end{aligned}$$

Intuitively, this transformation simply replaces all temporal operators in the original LTL formula with a corresponding quantified CTL operator.[3] It is clear that the transformation can be done in linear time in the size of the formula.

Let $K = \langle S, \rho, s_0, \pi\rangle$ be a finite-state Kripke structure representing the witness behavior extracted using the product automaton as described above. Each state of this structure has exactly one successor, and every state of the structure is reachable from its initial state. Let $x = s_0 s_1 s_2 \ldots \in S^\omega$ denote the unique infinite state sequence which begins in the initial state $s_0 \in S$.

Let $\varphi$ be an LTL formula over the atomic propositions in $AP$. We have the following:

**Theorem 1.** *The infinite state sequence $x$ satisfies the LTL formula $\varphi$ if and only if the CTL formula $\tau(\varphi)$ holds in $s_0$, the initial state of the Kripke structure $K$.*

*Proof.* By induction on the syntactic structure of the formula.

This result allows us to use CTL model checking techniques for checking the satisfiability of the LTL formula separately in the system behavior. For example, we can apply a global CTL model checking algorithm (see e.g. [7]) to the path, evaluating each of the subformulas of $\tau(\varphi)$ in turn in each state of the path in order to finally obtain the truth value of $\tau(\varphi)$ in the initial state. The complexity of this algorithm is $\mathcal{O}(|F| \cdot |S|)$, where $|F|$ denotes the number of subformulas of $\tau(\varphi)$. By the previous theorem, the result tells also the truth value of the LTL formula $\varphi$ in the execution. This result can then be used to detect which one of the tested LTL-to-Büchi converter implementations was probably in error in this test case, i.e. which automaton incorrectly accepted or rejected the execution.

## 3 Automated Testing of the Implementations

We have implemented the testing procedure of the previous section into a testbench for LTL-to-Büchi translation algorithm implementations. The C++ source

[3] Actually, due to the special characteristics of the executions, even the semantics of the CTL path quantifiers **A** and **E** coincide. Therefore, we could as well use the **E** quantifier without affecting the results presented here.

code for this program is available through Heikki Tauriainen's homepage at <URL: `http://www.tcs.hut.fi/%7Ehtauriai/`>.

### 3.1 Test Program Operation

The testbench model checks randomly generated LTL formulas in randomly generated reachability graphs, using each of the different LTL-to-Büchi conversion algorithm implementations in turn to expand the LTL formulas and their negations into Büchi automata.

In addition to generating the input, the program performs for each used implementation the global synchronous product computation, together with the check for states beginning some execution satisfying the property. Finally, the program compares the results obtained using each LTL-to-Büchi conversion algorithm with each other and reports whether any inconsistencies were detected. After this, the test procedure is repeated using another randomly generated LTL formula and/or a reachability graph.

The testing can be interrupted in case any error is detected, for example, when an implementation fails to generate a Büchi automaton (e.g., due to an internal assertion violation in the implementation). The program provides a command-line interface through which the user can examine the formula, the reachability graph and the Büchi automata generated by the different implementations more closely, optionally invoking a path checking algorithm for LTL to determine which one of the implementations was in error.

The test program collects statistics on the failure rates of the different implementations and records the average sizes of the Büchi automata generated by the individual implementations. Also the average running times of the different implementations are recorded. These capabilities allow the test program to be also used for simple benchmarking of the different implementations.

Interfacing the test program with any LTL-to-Büchi translation algorithm implementation requires writing an additional program module, whose purpose is to translate the input formulas and the outputted Büchi automata between the representations used by the testbench and the LTL-to-Büchi converter implementation. A module is already provided for doing this for SPIN's input syntax and the resulting Büchi automata (called "never claims" in SPIN terminology).

### 3.2 Generating the Input

The testbench implementation uses randomly generated LTL formulas and reachability graphs as input for the test procedure, allowing the user some control over the behavior of the input generation algorithms through the use of several parameters.

Simple random testing is by no means sufficient for proving the absolute correctness of any LTL-to-Büchi conversion algorithm implementation. In addition, even though seemingly "random" input generation methods are very easy to come up with, these methods may easily create some kind of bias in the output. This makes it hard to analyze the general efficiency of the test procedure

in finding errors in the tested implementations. However, since the correctness of the test procedure is independent of any particular input generation method, we do not consider this to be a major weakness. As a matter of fact, it has been our experience that even simple random input generation methods have been quite effective in uncovering flaws in many practical implementations, thus being helpful in improving their robustness.

We shall now describe the input generation methods used in the testbench implementation.

**Random LTL Formulas.** The test program generates the random LTL input formulas using a recursive algorithm similar to the one in [3] to obtain formulas containing an exact given number of symbols (logical operators, Boolean constants or propositional variables).

The behavior of the algorithm can be customized through the use of several parameters. For example, these parameters allow changing the number of available atomic propositions or setting the relative priorities of choosing any particular logical operator in the algorithm. This can be also used for disabling the use of some operators altogether.

The complete set of available operators for use in the generated formulas is $\neg$ (logical negation), $\mathbf{X}$ ("Next"), $\Box$ ("Always" or "Globally"), $\Diamond$ ("Eventually" or "Finally"), $\wedge$ (logical conjunction), $\vee$ (logical disjunction), $\rightarrow$ (logical implication), $\leftrightarrow$ (logical equivalence), $\mathbf{U}$ ("Until"), and $\mathbf{V}$ ("Release", the dual of $\mathbf{U}$).

Pseudocode for the algorithm is shown in Fig. 2. The argument $n$ for the algorithm denotes the desired number of symbols in the generated formula.

**Reachability Graphs.** The algorithm used in the test program implementation for constructing random reachability graphs is presented in Fig. 3. The goal of the algorithm is to generate graphs with a given number of states $n$, with the additional requirement of ensuring the reachability of every graph state from the initial state of the graph. Since the test procedure is concerned with infinite system behaviors, the algorithm will also ensure that each graph state has at least one successor.

Beginning with the initial state of the graph, the algorithm processes each node of the graph in turn. In order to ensure the reachability and the graph size requirements, the algorithm first chooses a random state already known to be reachable from the initial state (line 8) and connects it to some yet unreachable state, if there are still any available (lines 14–19). Then, random edges are inserted between the chosen node and all other graph nodes (lines 20–27). The probability of inserting these edges can be controlled with the parameter $p$. Finally, if the chosen state still has no successors, it is simply connected to itself in order to avoid any finite terminating behaviors (lines 28–29).

The truth values of the atomic propositions are chosen randomly in each processed state (lines 10–13). The parameter $t$ denotes the probability with which any of the propositions is given the value "TRUE" in a state.

```
1   function RandomFormula (n : Integer) : LtlFormula
2   begin
3       if n = 1 then begin
4           p := random symbol in AP ∪ {True, False};
5           return p;
6       end
7       else if n = 2 then begin
8           op := random operator in the set {¬, X, □, ◇};
9           φ := RandomFormula(1);
10          return op φ;
11      end
12      else
13          op := random operator in the set {¬, X, □, ◇, ∧, ∨, →, ↔, U, V};
14          if op ∈ {¬, X, □, ◇} then begin
15              φ := RandomFormula(n − 1);
16              return op φ;
17          end
18          else begin
19              x := random integer in the interval [1, n − 2];
20              φ := RandomFormula(x);
21              ψ := RandomFormula(n − x − 1);
22              return (φ op ψ);
23          end;
24      end;
25  end;
```

**Fig. 2.** Pseudocode for the formula generation algorithm

The algorithm repeats these steps for each state of the graph, until all states have been processed.

**Random paths.** The test program can also use an alternative method of generating random *paths* as the reachability graphs used as input for the test procedure. These paths simply consist of a given number of states connected to form a sequence whose last state is connected to some randomly chosen previous state in the sequence, thereby forming a cycle. The atomic propositions are then given random truth values in each state as in the previous algorithm.

Generating random paths as reachability graphs has the advantage of allowing us to perform an additional cross-comparison for the model checking results obtained using the different LTL-to-Büchi translation algorithms. The check is based on the use of LTL path checking algorithm based on the methods discussed in Sect. 2.4. This algorithm is first used to evaluate the given LTL formula $\varphi$ in the executions beginning at each state of the path. In this restricted class of reachability graphs, the previously computed model checking results for the CTL* formula $\mathbf{E}\,\varphi$ using each LTL-to-Büchi translation algorithm should now exactly correspond to the results returned by the path checking algorithm in each state of the path. This follows from the fact that in this class of reachability graphs, the semantics of CTL* path quantifiers $\mathbf{E}$ and $\mathbf{A}$ coincide. For the same reason, the model checking results computed for the CTL* formula $\mathbf{E}\,\neg\varphi$ should be exactly the opposite.

```
1   function RandomGraph(n : Integer, p : Real ∈ [0.0, 1.0], t : Real ∈ [0.0, 1.0])
              : KripkeStructure
2   begin
3       S := {s_0, s_1, ..., s_{n-1}};
4       NodesToProcess := {s_0};
5       UnreachableNodes := {s_1, s_2, ..., s_{n-1}};
6       ρ := ∅;
7       while NodesToProcess ≠ ∅ do begin
8           s := a random node in NodesToProcess;
9           NodesToProcess := NodesToProcess \ {s};
10          π(s) := ∅;
11          for all P ∈ AP do
12              if RandomNumber(0.0, 1.0) < t then
13                  π(s) := π(s) ∪ {P};
14          if UnreachableNodes ≠ ∅ then begin
15              s' := a random node in UnreachableNodes;
16              UnreachableNodes := UnreachableNodes \ {s'};
17              NodesToProcess := NodesToProcess ∪ {s'};
18              ρ := ρ ∪ {(s, s')};
19          end;
20          for all s' ∈ S do
21              if RandomNumber(0.0, 1.0) < p then begin
22                  ρ := ρ ∪ {(s, s')};
23                  if s' ∈ UnreachableNodes then begin
24                      UnreachableNodes := UnreachableNodes \ {s'};
25                      NodesToProcess := NodesToProcess ∪ {s'};
26                  end;
27              end;
28          if there is no edge (s, s') in ρ for any s' ∈ S then
29              ρ := ρ ∪ (s, s);
30      end;
31      return ⟨S, ρ, s_0, π⟩;
32  end;
```

**Fig. 3.** Pseudocode for the reachability graph generation algorithm

Therefore, using single paths as random reachability graphs gives an additional algorithm to be used in testing the different implementations against each other, providing also for limited testing of a single LTL-to-Büchi conversion algorithm implementation: if the input consisted of more general graphs, at least two implementations would always be required in order to be able to perform any testing based on the cross-comparison of the results given by the different implementations.

## 4 Application of the LTL Path Checking Algorithm in LTL Model Checking Tools

We suggest an additional application of the methods of Sect. 2.4 for model checking LTL formulas in single state sequences. Namely, an LTL path checking algorithm could also be used in practical LTL model checking tools for validating the counterexamples produced by the tool. Integrating the path checking algorithm as an additional last step of the model checking process into a model checker could give some assurance that the counterexample produced by the tool is really correct. In addition, any errors detected in this phase suggest possible errors in the model checker implementation.

The results of a global CTL model checking algorithm, when applied to the verification of an LTL property in a single system behavior, can also be easily used to automatically produce a proof or a refutation for the property in the behavior. We have used this idea in the testbench implementation for justifying to the user the claim for the failure of one of the tested LTL-to-Büchi conversion implementations when analyzing contradictory results.

## 5 Testing SPIN's LTL-to-Büchi Conversion

We used our test program implementation for testing the LTL-to-Büchi conversion algorithm implemented in the model checker SPIN [6, 5]. The implementation is originally based on the algorithm presented in [4] with several optimizations. We used the testbench on SPIN versions 3.3.7, 3.3.8, 3.3.9 and 3.3.10, which was the most recent version available at the time of writing.

As a reference implementation, we used another implementation based on an open source C++ class library [10] (extended with some locally developed code), originally a part of the Åbo System Analyser (ÅSA) model checking package [9]. This is an independent, very straightforward implementation of the Büchi automaton construction algorithm in [4]. Even though the tested implementations are based on the same algorithm, we find the independence of the actual implementations far more relevant, since our focus is not on testing the correctness of the abstract algorithm (which is already known to be correct [4]). We have also used the testbench on implementations based on different algorithms for converting LTL formulas into Büchi automata, such as the algorithm of [15] implemented in the tool PROD [16], but due to some limitations in PROD's

input syntax (namely, the lack of support for the **X** and **V** operators) we did not include that implementation in the tests made here.

We ran the tests using both the more general random graph algorithm and the random path algorithm for generating reachability graphs with 100 states (in the random graph algorithm, a random edge between two states was added with the probability 0.2). The random LTL formulas used as input for the LTL-to-Büchi translation algorithm implementations consisted of 4 to 7 symbols. Five atomic propositions (with equal probability of being used in a formula) were available for use in the generated formulas. In generating the reachability graphs, each proposition had an equal probability of being true or false in each individual state of the graph. For each reachability graph generation method and for each different number of formula symbols, we ran each LTL-to-Büchi converter implementation on 4,000 randomly generated formulas and their negations. In total, each individual implementation was therefore run on 64,000 input formulas. Additionally, a new reachability graph was generated after every tenth generated formula.

The randomly generated formulas were partitioned into four batches of equal size, using in each batch a different subset of formula symbols in generating the formulas. The symbol sets used in the different batches were

(1) atomic propositions; no Boolean constants; operators $\neg, \Diamond, \Box, \wedge, \vee, \rightarrow, \leftrightarrow$, $\mathbf{U}, \mathbf{V}$
(2) atomic propositions; Boolean constants TRUE and FALSE; the same operators as in (1)
(3) atomic propositions; no Boolean constants; all operators in (1) together with the **X** operator
(4) atomic propositions; all Boolean constants and logical operators.

Each available operator had an equal priority of being selected into a generated formula by the algorithm in Sect. 3.2; however, each Boolean constant (when included in the symbol set) had the smaller probability of 0.05 of being selected, compared to the probability of 0.18 used for each of the five propositional variables.

The tests were run using Linux PCs. Table 1 shows the failure rates of each implementation during the conversion of an LTL formula into a Büchi automaton. All the tested implementations except SPIN 3.3.9 sometimes failed to produce acceptable output (interpreted as a failure to generate an automaton). The reported failures of the reference implementation are due to its failure to produce any output after running for 12 hours. On the other hand, all SPIN versions never consumed more than only a few seconds of running time, showing the straightforward reference implementation very inefficient in practice.

All but one SPIN version failed in some cases to produce acceptable output. Occasionally, the never claims produced by SPIN versions 3.3.7 and 3.3.8 were syntactically incorrect. Moreover, in some cases both versions failed due to an internal error on some input formulas (all of which contained the logical equivalence operator $\leftrightarrow$) without producing any output. Version 3.3.9 never failed

**Table 1.** Büchi automaton generation failure statistics

| Number of symbols in formula | Implementation | Number of Büchi automaton generation failures (of 4,000 attempts) | | | |
|---|---|---|---|---|---|
| | | (1) | (2) | (3) | (4) |
| 4 | ÅSA | 0 | 0 | 0 | 0 |
| | SPIN 3.3.7 | 38 | 72 | 84 | 92 |
| | SPIN 3.3.8 | 38 | 72 | 84 | 92 |
| | SPIN 3.3.9 | 0 | 0 | 0 | 0 |
| | SPIN 3.3.10 | 0 | 0 | 0 | 0 |
| 5 | ÅSA | 0 | 0 | 0 | 0 |
| | SPIN 3.3.7 | 100 | 158 | 138 | 167 |
| | SPIN 3.3.8 | 100 | 158 | 138 | 167 |
| | SPIN 3.3.9 | 0 | 0 | 0 | 0 |
| | SPIN 3.3.10 | 10 | 13 | 13 | 9 |
| 6 | ÅSA | 0 | 0 | 0 | 0 |
| | SPIN 3.3.7 | 142 | 168 | 215 | 198 |
| | SPIN 3.3.8 | 142 | 168 | 215 | 198 |
| | SPIN 3.3.9 | 0 | 0 | 0 | 0 |
| | SPIN 3.3.10 | 8 | 7 | 8 | 5 |
| 7 | ÅSA | 1 | 2 | 2 | 2 |
| | SPIN 3.3.7 | 138 | 220 | 248 | 293 |
| | SPIN 3.3.8 | 138 | 220 | 248 | 293 |
| | SPIN 3.3.9 | 0 | 0 | 0 | 0 |
| | SPIN 3.3.10 | 12 | 4 | 10 | 5 |

to produce correctly formatted output; however, version 3.3.10 again failed on some input formulas, reporting an internal error instead.

Finally, Tables 2 and 3 contain the number of input formulas failing the result cross-comparison check between the implementations. The tables also include the total number of failed consistency checks for each individual implementation. The results of Table 2 were obtained using randomly generated reachability graphs as input for the testing procedure, while the results of Table 3 are based on using randomly generated paths as input. The results are grouped according to the used set of formula symbols.

The results show that there were cases in which SPIN versions 3.3.7, 3.3.8 and 3.3.9 failed the result cross-comparison check with version 3.3.10 and the reference implementation, however SPIN 3.3.9 failed only if the input formulas were allowed to contain Boolean constants or **X** operators. SPIN 3.3.10 never failed the result cross-comparison check with the reference implementation. SPIN versions 3.3.7, 3.3.8 and 3.3.9 also occasionally failed the result consistency check. However, the relatively rare occurrence of these failures seems to suggest that the result cross-comparison check is more powerful of these methods for detecting errors in an implementation. However, unlike the consistency check, this test always requires a separate analysis of the results produced by two implementations to determine which one of them is incorrect.

The test results show a clear improvement in the robustness of SPIN's LTL-to-Büchi conversion algorithm implementation since version 3.3.7. SPIN 3.3.10

**Table 2.** Result cross-comparison statistics (random graphs)

| Formula symbol set | Implementation | Total number of consistency check failures | Total number of result cross-comparison failures / number of comparisons performed | |
|---|---|---|---|---|
| (1) | | | ÅSA | SPIN 3.3.10 |
| | ÅSA | 0/4000 | — | 0/7989 |
| | SPIN 3.3.7 | 0/3894 | 907/7788 | 907/7777 |
| | SPIN 3.3.8 | 0/3894 | 907/7788 | 907/7777 |
| | SPIN 3.3.9 | 0/4000 | 0/8000 | 0/7989 |
| | SPIN 3.3.10 | 0/3989 | 0/7989 | — |
| (2) | ÅSA | 0/3998 | — | 0/7985 |
| | SPIN 3.3.7 | 4/3841 | 849/7680 | 849/7669 |
| | SPIN 3.3.8 | 4/3841 | 849/7680 | 849/7669 |
| | SPIN 3.3.9 | 1/4000 | 1/7998 | 1/7987 |
| | SPIN 3.3.10 | 0/3987 | 0/7985 | — |
| (3) | ÅSA | 0/3998 | — | 0/7983 |
| | SPIN 3.3.7 | 0/3811 | 624/7647 | 623/7635 |
| | SPIN 3.3.8 | 0/3811 | 624/7647 | 623/7635 |
| | SPIN 3.3.9 | 0/4000 | 0/7998 | 0/7985 |
| | SPIN 3.3.10 | 0/3985 | 0/7983 | — |
| (4) | ÅSA | 0/3999 | — | 0/7988 |
| | SPIN 3.3.7 | 4/3777 | 747/7582 | 747/7572 |
| | SPIN 3.3.8 | 4/3777 | 747/7582 | 747/7572 |
| | SPIN 3.3.9 | 0/4000 | 64/7999 | 64/7989 |
| | SPIN 3.3.10 | 0/3989 | 0/7988 | — |

**Table 3.** Result cross-comparison statistics (random paths)

| Formula symbol set | Implementation | Total number of consistency check failures | Total number of result cross-comparison failures / number of comparisons performed | |
|---|---|---|---|---|
| (1) | | | ÅSA | SPIN 3.3.10 |
| | ÅSA | 0/3999 | — | 0/7980 |
| | SPIN 3.3.7 | 0/3897 | 926/7793 | 926/7775 |
| | SPIN 3.3.8 | 0/3897 | 918/7793 | 918/7775 |
| | SPIN 3.3.9 | 0/4000 | 0/7999 | 0/7981 |
| | SPIN 3.3.10 | 0/3981 | 0/7980 | — |
| (2) | ÅSA | 0/4000 | — | 0/7989 |
| | SPIN 3.3.7 | 6/3850 | 923/7700 | 923/7690 |
| | SPIN 3.3.8 | 6/3850 | 921/7700 | 920/7690 |
| | SPIN 3.3.9 | 0/4000 | 0/8000 | 0/7989 |
| | SPIN 3.3.10 | 0/3989 | 0/7989 | — |
| (3) | ÅSA | 0/4000 | — | 0/7984 |
| | SPIN 3.3.7 | 51/3820 | 825/7666 | 825/7650 |
| | SPIN 3.3.8 | 50/3820 | 822/7666 | 819/7650 |
| | SPIN 3.3.9 | 0/4000 | 0/8000 | 0/7984 |
| | SPIN 3.3.10 | 0/3984 | 0/7984 | — |
| (4) | ÅSA | 0/3999 | — | 0/7991 |
| | SPIN 3.3.7 | 60/3820 | 898/7666 | 899/7659 |
| | SPIN 3.3.8 | 60/3820 | 891/7666 | 892/7659 |
| | SPIN 3.3.9 | 0/4000 | 76/7999 | 76/7992 |
| | SPIN 3.3.10 | 0/3992 | 0/7991 | — |

was the first to pass all result cross-comparison checks with the reference implementation. However, in these tests this version was slightly more unstable than its immediate predecessor due to the occasional automaton generation failures. This may be a result of some new optimizations made in the newer version to reduce the size of the generated automata, which reminds that extreme care should always be taken when making optimizations into an implementation in order to retain its stability and correctness. This reveals another possible application for the test procedure as a regression testing method to be used in the development of new versions of an LTL-to-Büchi algorithm implementation.

## 6 Conclusions

We have presented a random testing method for LTL-to-Büchi conversion algorithm implementations. The approach was quite effective in uncovering errors in the SPIN model checker, and this has been our experience also with other model checkers on which we have run a smaller set of tests. (Of the four independent implementations we have tested, the reference implementation ÅSA has been the only one in which no errors have ever been detected using this method.)

This work can be seen as the continuation of the work in [13]. We have improved the methodology presented there by using counterexample validation algorithms (essentially, a global CTL model checker) to decide which one of several disagreeing implementations is incorrect. Also the use of random paths is first introduced here. We present a larger set of experimental results with up-to-date SPIN versions. New to this work is additionally the idea of using a counterexample validation algorithm as the last step of an LTL model checker.

In the future, we would like to extend the approach to also test nondeterministic finite automata generated from (syntactic) safety LTL formulas using the approaches presented in [8]. These (safety) LTL-to-NFA conversion algorithm implementations would need a different (more simple) emptiness checking subroutine, but the results could then be compared against the results obtained from the LTL-to-Büchi conversion tests for the same formula.

We will also continue using the test program on other LTL-to-Büchi converter implementations based on different conversion algorithms than the one used here. For example, it would be interesting to try the test program on the LTL2AUT implementation of [3]. The testbench implementation itself could also be still improved with some of the methods for the direct comparison of Büchi automata as described in the beginning of Sect. 2 (the emptiness check for the synchronous composition of two automata, however without the universality test for the union of the automata).

A very surprising result in the experiments was that using randomly generated paths as input reachability graphs for the testing procedure resulted in a slightly higher failure rate in the tested implementations. However, since there are so many factors affecting the test procedure (details of the input generation algorithms, the particular combination of values chosen for the test parameters, even the tested implementations themselves), it is impossible to say anything

conclusive about which one of these graph generation methods might be "better" for uncovering errors in the implementations. The relationship between the test parameters and the failure rates would certainly make an interesting issue to investigate in the future.

We propose that LTL model checkers should be extended with a counterexample validation algorithm as described in Sect. 4. The implementation of this algorithm is quite straightforward, as it can be done based on even a straightforward implementation of a CTL model checker. The running time overhead of such an algorithm should be quite negligible, as it is linear both in the size of the formula and the length of the counterexample. Such an algorithm would increase the confidence in the counterexamples provided by the model checker implementation, and could hopefully help in finding some of the yet unknown implementation errors in the tool. Of course, separate validation of counterexamples only helps in detecting false negatives but not false positives; however, it would still be a step in the right direction.

There are several other places in LTL model checkers which could probably be improved with random testing but which we have not covered here (for example, emptiness checking and partial order reduction algorithms). However, we think that the LTL-to-Büchi conversion algorithm, including all the possibly used optimizations, is one of the most difficult algorithms to implement, and thus should also be regression tested whenever new versions are implemented.

An optimal situation would of course be that the implementation of a model checker were fully verified (and still have adequate performance). Until that is the case, we aim at a more humble goal: to validate a part of the model checker using random testing.

## Acknowledgements

The random testing method has been used on SPIN starting from version 3.3.3, on which the first preliminary tests were made in the summer of 1999. More extensive tests have been done on versions 3.3.8, 3.3.9 and 3.3.10 since January 2000. We would like to thank Gerard J. Holzmann for creating new and improved versions of SPIN during the period this work was done.

We would also like to thank Tommi Junttila for critical comments on this work and Mauno Rönkkö for creating a reliable reference implementation. We are also grateful to the anonymous referees whose feedback was very important in improving this work.

## References

[1] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL$^*$. In *Proceedings of 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 388–397. IEEE Computer Society Press, 1995.

[2] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[3] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, pages 249–260. Springer, 1999. LNCS 1633.

[4] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of 15th Workshop Protocol Specification, Testing, and Verification*, pages 3–18, 1995.

[5] G. Holzmann. On-the-fly, LTL model checking with Spin. <URL: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`>.

[6] G. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[7] E. Clarke Jr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[8] O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *Proceedings of 11th International Conference on Computer Aided Verification (CAV'99)*, pages 172–183. Springer, 1999. LNCS 1633.

[9] J. Lilius. ÅSA: The Åbo System Analyser, September 1999. <URL: `http://www.abo.fi/%7Ejolilius/mc/aasa.html`>.

[10] M. Rönkkö. A distributed object oriented implementation of an algorithm converting a LTL formula to a generalised Buchi automaton, 1998. <URL: `http://www.abo.fi/%7Emauno.ronkko/ASA/ltlalg.html`>.

[11] S. Safra. *Complexity of automata on infinite objects*. PhD thesis, The Weizmann Institute of Science, 1989.

[12] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[13] H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae into Büchi automata. In *Proceedings of the Workshop Concurrency, Specification and Programming 1999 (CS&P'99)*, pages 251–262. Warsaw University, September 1999.

[14] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, pages 238–265, 1996. LNCS 1043.

[15] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Computer Society Press, 1986.

[16] K. Varpaaniemi, K. Heljanko, and J. Lilius. PROD 3.2 - An advanced tool for efficient reachability analysis. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, pages 472–475. Springer, June 1997. LNCS 1254.