

Testing LTL formula translation into Büchi automata

Heikki Tauriainen, Keijo Heljanko*

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

Published online: 2 October 2002 – © Springer-Verlag 2002

Abstract. Model checkers are often used to verify critical systems, and thus a lot of effort should be put on ensuring their reliability. We describe techniques for testing linear temporal logic (LTL) model checker implementations, focusing especially on LTL-to-Büchi automata translation. We propose a randomized testing approach based on the cross-comparison of independent translator implementations, and present methods for test failure analysis. We give experimental results using an automated tool implementing the testing methodology. This method has helped to detect errors in model checking tools such as SPIN. We also propose extending LTL model checkers with a simple counterexample validation algorithm.

Keywords: Verification – Model checking – Temporal logic – Software testing

1 Introduction

Model checkers are often used to verify properties of safety- or business-critical systems. Therefore, it is important that we can trust the model checking results obtained from the verification of a system. This places high demands on the reliability of the model checker implementation. To ensure the correctness of model checking results, it would be a tremendous advantage if the model checking tool had itself been verified or proved to be correct. However, full verification of complex software of this kind, especially when implemented in an imperative general-purpose programming language such as C,

is still out of reach of current software verification techniques. Nevertheless, even methods for only partially improving the robustness of model checkers would still be welcome.

Model checking of linear temporal logic (LTL) properties can be done using the automata-theoretic approach [19]. 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.

We propose a method for testing and, by employing the methodology in an LTL model checker implementation, hopefully improve the robustness of a central part of the implementation. We focus on the translation of LTL formulas into Büchi automata, which seems to be among the most difficult steps in automata-theoretic LTL model checking to implement correctly. The main reason for the difficulty seems to be due to the fact that the most advanced implementations contain several optimization methods, which may be nontrivial to implement. These optimizations are nevertheless required to obtain good model checking performance.

We propose a testing method based on using random input to test several independent implementations against each other. This work can be seen as the summary of the results presented in [15, 16]. The main contributions are:

- Test methods for LTL-to-Büchi translators are presented. These are based on simple automata-theoretic techniques and the LTL model checking procedure.
- We describe how to find an incorrect implementation in the presence of test failures. This method uses a *witness* (concrete evidence on the test failure) to prove one of the implementations incorrect. The method is based on model checking an LTL formula directly in

* The financial support of Academy of Finland (Projects 43963 and 47754) and the Tekniikan Edistämissäätiö Foundation is gratefully acknowledged.

the witness by applying computation tree logic (CTL) model checking techniques (an idea, to our knowledge, first presented in an extended version of [10]).

- We discuss an additional application of the previous technique in LTL model checking tools to validate the generated counterexamples automatically at runtime.
- We present experimental results obtained from an automated testing tool incorporating the test methods presented in this work. The tool uses randomly generated input to test several independently implemented LTL-to-Büchi translators against each other.

The organization of this article is as follows. We start in Sect. 2 by presenting the required definitions of LTL, automata theory, and LTL model checking. In Sect. 3, we describe the test methods devised for testing LTL-to-Büchi translators. The techniques that can be used to analyze test failures to find an incorrect implementation are presented in Sect. 4. We discuss our testbench implementation in Sect. 5, and continue with experimental results in Sect. 6. Conclusions with some directions for future work are presented in Sect. 7.

2 Preliminaries

In this section, we shall review the definitions of the basic theoretical concepts on which this work is based.

2.1 Linear temporal logic

Throughout the text, we use AP to denote a finite nonempty set of atomic propositions. The set of (propositional) linear temporal logic formulas is defined as follows:

- All atomic propositions $p \in AP$ are linear temporal logic formulas.
- If φ and ψ are linear temporal logic formulas, then so are $\neg\varphi$, $(\varphi \vee \psi)$, $X\varphi$ and $(\varphi \cup \psi)$.
- There are no other linear temporal logic formulas.

We use the following traditional semantics for linear temporal logic formulas. Let φ be a linear temporal logic formula, and let ξ be an infinite sequence over 2^{AP} (the set of subsets of AP). Denote by ξ^i the infinite subsequence of ξ that begins at the $(i+1)^{\text{th}}$ element of ξ ($i \geq 0$). The \models relation between infinite sequences over 2^{AP} and linear temporal logic formulas is defined inductively as follows:

- For an atomic proposition $p \in AP$, $\xi \models p$ iff p is contained in the first element of the sequence ξ .
- $\xi \models \neg\varphi$ iff not $(\xi \models \varphi)$. In this case, we write $\xi \not\models \varphi$.
- $\xi \models (\varphi \vee \psi)$ iff $\xi \models \varphi$ or $\xi \models \psi$.
- $\xi \models X\varphi$ iff $\xi^1 \models \varphi$.
- $\xi \models (\varphi \cup \psi)$ iff $\exists i \geq 0 : \xi^i \models \psi$ and $\forall 0 \leq j < i : \xi^j \models \varphi$.

If $\xi \models \varphi$ holds for an infinite sequence $\xi \in (2^{AP})^\omega$, then ξ is called a *model* of the formula φ . For convenience, we denote the set of all models $\{\xi \in (2^{AP})^\omega \mid \xi \models \varphi\}$ by \mathcal{L}_φ and call \mathcal{L}_φ the *language induced by φ* .

2.2 Büchi automata

The class of languages induced by LTL formulas is a subclass of the languages that can be represented with nondeterministic finite-state automata on infinite words called *Büchi automata* (the ω -regular languages; see, for example, [17]). This property has been used to devise the automata-theoretic approach to LTL model checking [19] and therefore presents a need for translating LTL formulas into Büchi automata. We use the following definition.

A Büchi automaton is a tuple $A = \langle \Sigma, Q, \Delta, Q^0, F \rangle$, where:

- Σ is the *alphabet*,
- Q is the finite set of *states*,
- $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*,
- $Q^0 \subseteq Q$ is the set of *initial states*, and
- $F \subseteq Q$ is the set of *accepting states*.

An *execution* of A over an infinite sequence $\xi = \langle x_0, x_1, x_2, \dots \rangle \in \Sigma^\omega$ is an infinite sequence of states $r = \langle q_0, q_1, q_2, \dots \rangle \in Q^\omega$ such that $q_0 \in Q^0$, and for all $i \geq 0$, $(q_i, x_i, q_{i+1}) \in \Delta$.

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

The automaton *accepts* an infinite word $\xi \in \Sigma^\omega$ if and only if there is an accepting execution of A on ξ . If A has no accepting executions on ξ , then A *rejects* ξ .

The set of infinite sequences $\xi \in \Sigma^\omega$ that are accepted by the automaton A is called the language *accepted* (or alternatively, *recognized*) by A , and it is denoted by \mathcal{L}_A .

By basic results of automata theory (see, for example, [18]), two Büchi automata $A_1 = \langle \Sigma_1, Q_1, \Delta_1, Q_1^0, F_1 \rangle$ and $A_2 = \langle \Sigma_2, Q_2, \Delta_2, Q_2^0, F_2 \rangle$ can be composed together into another Büchi automaton that accepts precisely those infinite sequences over $\Sigma_1 \cap \Sigma_2$ that are accepted by both of the original automata, that is, the language $\mathcal{L}_{A_1} \cap \mathcal{L}_{A_2}$. The result of the composition is called the *intersection* of A_1 and A_2 ; in the following, we will denote it by $A_1 \otimes A_2$. The intersection automaton can be built by a *synchronous composition* of the two original automata that results in an automaton having $\mathcal{O}(|Q_1| \cdot |Q_2|)$ states in the worst case; see [18] for one possible construction.

2.3 The LTL model checking problem

The goal of LTL model checking is to determine whether a property expressed as an LTL formula is satisfied in a finite model of a system to be verified. The model can be given as a *Kripke structure*, which represents the system as a state-transition graph whose each state is augmented with a set of atomic propositions to encode the properties that hold in the state.

Formally, a Kripke structure is a tuple $M = \langle S, \rho, \pi \rangle$, where:

- S is the finite set of *states*,
- $\rho \subseteq S \times S$ is the *transition relation* that satisfies the constraint $\forall s \in S : \exists s' \in S : (s, s') \in \rho$,
- $\pi : S \rightarrow 2^{AP}$ is the *labeling function* that associates each state with a set of atomic propositions. Semantically, $\pi(s)$ represents the set of propositions that hold in a state $s \in S$.

An infinite *path* in the Kripke structure is an infinite sequence of states $x = \langle s_0, s_1, s_2, \dots \rangle \in S^\omega$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in \rho$.

It is easy to see that every infinite path $x = \langle s_0, s_1, s_2, \dots \rangle \in S^\omega$ of the Kripke structure corresponds to an infinite sequence $\xi = \langle \pi(s_0), \pi(s_1), \pi(s_2), \dots \rangle \in (2^{AP})^\omega$. For convenience, we denote $\pi(x) = \xi$ and call ξ the *temporal interpretation* of x . The LTL model checking problem for Kripke structures can then be stated as follows:

Given a Kripke structure $M = \langle S, \rho, \pi \rangle$, a state $s \in S$ and an LTL formula φ , does there exist an infinite path $x = \langle s_0, s_1, s_2, \dots \rangle \in S^\omega$ (where $s_0 = s$) such that $\pi(x) \models \varphi$ holds for x ?

We call this the *local model checking problem* for LTL. The *global model checking problem* amounts to finding *all* states $s \in S$ that fulfill the previous condition.

We remark that the above formulation of the LTL model checking problem slightly differs from the classic definition found in the literature [19], where the problem is usually presented as a question whether some property ψ holds for *all* infinite paths that begin at some designated state s . The answer to this question is found in [19] by checking whether there exists *any* such path x in the Kripke structure for which the property $\pi(x) \models \neg\psi$ holds; consequently, the property $\pi(x) \models \psi$ then holds for all infinite paths starting from s only if this is *not* the case. However, this corresponds precisely to solving the local model checking problem as defined above for the formula $\varphi = \neg\psi$.

In the local model checking problem, the given state s fixes a set of infinite paths that need to be considered when searching for the answer to the problem. By treating this set of paths as a language $\mathcal{L}_{M,s} = \{\pi(x) \mid x \text{ is an infinite path of } M \text{ starting from } s\} \subseteq (2^{AP})^\omega$, the local LTL model checking problem can be compactly expressed as the question whether $\mathcal{L}_{M,s} \cap \mathcal{L}_\varphi \neq \emptyset$.

Furthermore, the Kripke structure $M = \langle S, \rho, \pi \rangle$ (with a designated state $s \in S$) can be seen as a Büchi automaton that recognizes the language $\mathcal{L}_{M,s}$; indeed, it is easy to check that the automaton $A_{M,s} = \langle \Sigma, Q, \Delta, Q^0, F \rangle$, where $\Sigma = 2^{AP}$, $Q = S$, $Q^0 = \{s\}$, $F = Q$ and $\Delta = \{(s, a, s') \mid (s, s') \in \rho, \pi(s) = a\}$ accepts an infinite sequence $\xi \in (2^{AP})^\omega$ if and only if ξ is the temporal interpretation $\pi(x)$ of some infinite path x starting from the state s in the Kripke structure M .

We now know that both the LTL formula φ and, by choosing a state $s \in S$, the Kripke structure M can be transformed into Büchi automata A_φ and $A_{M,s}$ that recognize the languages \mathcal{L}_φ and $\mathcal{L}_{M,s}$, respectively. There-

fore, the solution for the local LTL model checking problem in state s can be found by first constructing the Büchi automaton $A_{M,s} \otimes A_\varphi$ that accepts the language $\mathcal{L}_{M,s} \cap \mathcal{L}_\varphi$ and then checking whether this automaton accepts any infinite sequence over 2^{AP} . This test, usually called the automaton *emptiness check*, can be done with an algorithm that checks whether the intersection automaton can reach from any of its initial states a state cycle passing through some accepting state (see, for example, [9]). The time required for this check is linear in the size of the intersection automaton. The memory requirements of model checking can often be improved by using *on-the-fly* model checking techniques [3, 4, 7, 11], which are able to detect LTL property violations without constructing the intersection automaton $A_{M,s} \otimes A_\varphi$ explicitly by combining the construction of the automata $A_{M,s}$ and A_φ with an emptiness checking algorithm. However, all actual on-the-fly LTL model checking tools we know of still isolate the construction of A_φ (i.e., the LTL-to-Büchi translation phase) into a separate program module.

A straightforward way to solve the global LTL model checking problem is to solve the local model checking problem separately for each state s of the Kripke structure. This procedure requires a total of $|S|$ compositions of the formula automaton A_φ with some automaton $A_{M,s}$ built from the Kripke structure; each of the individual compositions results in an automaton with $\mathcal{O}(|Q| \cdot |S|)$ states in the worst case.

It is possible to improve the global model checking procedure so that the formula automaton A_φ needs to be composed with the Kripke structure only once, while still retaining the $\mathcal{O}(|Q| \cdot |S|)$ worst-case upper bound for the number of states in the result of the composition. Basically, the composition can be done more efficiently by sharing the common substructures that would emerge in the individual “local” compositions. Essentially, the modified composition corresponds to intersecting the formula automaton with the Büchi automaton A_M that can be obtained from any $A_{M,s}$ by expanding its initial state set to cover all states of the automaton. A more detailed description of this simple idea, together with the changes that it requires to the emptiness check (which has to be performed globally, too) can be found in [15].

3 Test methods for LTL-to-Büchi translators

This section introduces testing methods for LTL-to-Büchi translators. After a brief introduction to the general testing methodology in Sect. 3.1, Sect. 3.2 focuses on test methods based on the analysis of Büchi automata constructed by the tested implementations using an LTL formula as input. Although these methods are in principle powerful in detecting errors in the implementations, they have some practical disadvantages that make their full-fledged implementation difficult. Therefore, in Sect. 3.3 we propose alternative, more easily implementable test-

ing methods that make use of the LTL model checking procedure, thus using both an LTL formula and a Kripke structure as input. However, due to their greater dependency on the input used for the tests, these test methods do not share the full power of the Büchi automata analysis techniques described in Sect. 3.2; the methods have nevertheless been found to work well in practice [14–16].

3.1 Principles of the testing methodology

Clearly, the methods for testing the correctness of LTL-to-Büchi translation algorithm implementations should be as powerful as possible for detecting errors in the implementations. Preferably, the test methods should also be simple and easy to implement to reduce the effort needed for ensuring the reliability of a tool that is itself used for testing the translators.

The test methods to be presented here are not intended for *proving* the correctness of LTL-to-Büchi translator implementations. In part, this is a consequence of the need for input, i.e., LTL formulas and Kripke structures. Therefore, the tests should be seen as simple methods for detecting inconsistencies in the implementations. The key idea of the tests is to validate the results given by an LTL-to-Büchi translator with those obtained using other implementations, with the intuitive assumption that independent implementations are unlikely to fail the formula translation in exactly the same way on every LTL formula. Errors in the translator implementations should therefore give rise to inconsistent test results leading to test failures. We note that the principle of validating the results of several implementations against each other to detect software faults is similar to the approach used in *N*-version programming methodology [1, 2]. Our main goal, however, is in testing individual LTL-to-Büchi translator implementations instead of combining several implementations into a fault-tolerant software package.

All tests to be presented need LTL formulas to be given as input for the translators; additionally, the tests of Sect. 3.3 also need Kripke structures to be used for running the LTL model checking procedure. However, the tests are not dependent on any particular kind of LTL formulas or Kripke structures, which helps to generate test cases automatically using even simple randomly generated formulas and structures. We believe that the strategy of using randomly generated input is adequate for testing several independent implementations and is an acceptable compromise between testing efficiency and the amount of effort that would otherwise be needed for choosing a representative sample of input formulas and Kripke structures to be used in the tests.

3.2 Analysis of Büchi automata

The semantics of LTL gives rise to a pair of correctness tests that can be applied to the Büchi automata constructed by LTL-to-Büchi translator implementations. In

particular, we shall repeatedly make use of the following facts concerning the relationship between the languages \mathcal{L}_φ and $\mathcal{L}_{\neg\varphi}$ that comprise the models of a given LTL formula φ and its negation $\neg\varphi$, respectively:

Let φ be an LTL formula, and let $\xi \in (2^{AP})^\omega$. By the semantics of LTL, $\xi \in \mathcal{L}_\varphi$ if and only if $\xi \notin \mathcal{L}_{\neg\varphi}$. This implies that:

1. The languages \mathcal{L}_φ and $\mathcal{L}_{\neg\varphi}$ are disjoint, i.e., $\mathcal{L}_\varphi \cap \mathcal{L}_{\neg\varphi} = \emptyset$.
2. Each sequence $\xi \in (2^{AP})^\omega$ belongs to either of the languages \mathcal{L}_φ or $\mathcal{L}_{\neg\varphi}$, and thus $\mathcal{L}_\varphi \cup \mathcal{L}_{\neg\varphi} = (2^{AP})^\omega$.

Let A_φ and $A_{\neg\varphi}$ be two Büchi automata constructed from the LTL formulas φ and $\neg\varphi$ using an LTL-to-Büchi translator. As a requirement for the correctness of the implementation, we expect A_φ and $A_{\neg\varphi}$ to accept the languages \mathcal{L}_φ and $\mathcal{L}_{\neg\varphi}$, respectively, which are known to be complementary to each other. Consequently, there should not exist any infinite sequence over 2^{AP} that is accepted by both A_φ and $A_{\neg\varphi}$; nor should any infinite sequence over 2^{AP} be rejected by both of these automata.

The following tests focus on checking for a complementary relationship between the languages accepted by A_φ and $A_{\neg\varphi}$. However, we note that this check is still not sufficient for proving these languages to be equivalent to the languages \mathcal{L}_φ and $\mathcal{L}_{\neg\varphi}$; for example, the check would not detect the error in an LTL-to-Büchi translation algorithm implementation that by mistake *negates* each input formula before the translation but then performs the translation itself correctly. We shall return to the issue of validating the languages at the end of this section after first describing the tests.

Checking the automata A_φ and $A_{\neg\varphi}$ for common accepting inputs can be done by investigating the *intersection* of the two automata as described in Sect. 2.3: assuming that A_φ and $A_{\neg\varphi}$ are correct, the automaton $A_\varphi \otimes A_{\neg\varphi}$ should accept precisely the intersection of the languages \mathcal{L}_φ and $\mathcal{L}_{\neg\varphi}$, which is known to be empty, as seen above. However, the existence of an input accepted by the intersection automaton implies that at least one of the automata A_φ or $A_{\neg\varphi}$ does not correctly recognize the expected language, so the LTL-to-Büchi translator (or at least one of several translators) used for constructing the automata must have an error. We thus obtain the test procedure shown in Fig. 1.

A successful emptiness check of $A_\varphi \otimes A_{\neg\varphi}$ is not sufficient for showing that the languages accepted by A_φ and $A_{\neg\varphi}$ are complementary: for instance, the test fails to reveal any errors in an LTL-to-Büchi translator that “cheats” by always constructing an empty Büchi automaton regardless of the formula given as input for the translator.

Proving that the languages accepted by A_φ and $A_{\neg\varphi}$ are complementary to each other requires another test based on the fact that no infinite sequence over 2^{AP} should be rejected by both of the automata. In principle, this can be checked for the automata A_φ and $A_{\neg\varphi}$ by first

Test 1: Emptiness check for the intersection of two Büchi automata

Input: LTL formula φ

1. Compute the Büchi automata A_φ and $A_{\neg\varphi}$ using an LTL-to-Büchi translator implementation (or two different implementations).
2. Compute the intersection automaton $A_\varphi \otimes A_{\neg\varphi}$.
3. Check $A_\varphi \otimes A_{\neg\varphi}$ for emptiness. If the intersection automaton accepts any input, then either A_φ or $A_{\neg\varphi}$ does not correctly recognize the language \mathcal{L}_φ or $\mathcal{L}_{\neg\varphi}$, respectively. This suggests that the translation of at least one of the formulas into a Büchi automaton has failed.

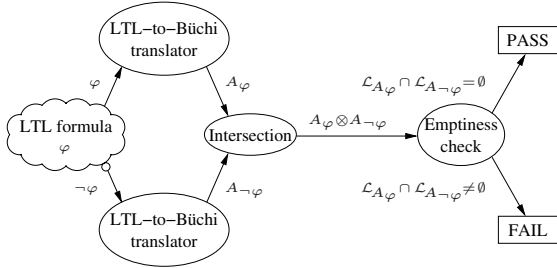


Fig. 1. Emptiness check for the intersection of two Büchi automata

constructing the *union* $A_\varphi \cup A_{\neg\varphi}$ of A_φ and $A_{\neg\varphi}$ (another Büchi automaton, see, for example, [18]) and then checking whether the union automaton accepts the *universal language* $(2^{AP})^\omega$. The existence of an input rejected by the union automaton then suggests that the translation of φ or $\neg\varphi$ into a Büchi automaton was not performed correctly.

Unfortunately, checking the universality of a non-deterministic Büchi automaton is a PSPACE-complete problem in the size of the automaton [18], and it cannot be solved directly with techniques similar to those used in the automaton emptiness check. In principle, it is possible to reduce the test into an emptiness check for Büchi automata: instead of checking the union automaton itself for universality, we could alternatively check its *complement* automaton $\overline{A_\varphi \cup A_{\neg\varphi}}$ (a Büchi automaton accepting precisely those infinite sequences *not* accepted by $A_\varphi \cup A_{\neg\varphi}$) for emptiness. However, this approach requires using a Büchi automata complementation procedure, which will (not surprisingly) cause an exponential ($2^{\mathcal{O}(n \log n)}$) blow-up in the size n of the union automaton [13]. Since the automata A_φ , $A_{\neg\varphi}$ and the union automaton may already have $2^{\mathcal{O}(|\varphi|)}$ states in the length $|\varphi|$ of the formula [20], we suspect that a straightforward implementation of the automaton universality check may prove impractical for all but the shortest LTL formulas. (This is also the reason why we chose not to implement Test 2 at all in the LTL-to-Büchi translator testing tool described in Sect. 5, in

Test 2: Universality check for the union of two Büchi automata

Input: LTL formula φ

1. Compute the Büchi automata A_φ and $A_{\neg\varphi}$ using some LTL-to-Büchi translator implementation (or two different implementations).
2. Compute the union of A_φ and $A_{\neg\varphi}$.
3. Use a Büchi automata complementation procedure to compute the complement of $A_\varphi \cup A_{\neg\varphi}$.
4. Check $\overline{A_\varphi \cup A_{\neg\varphi}}$ for emptiness. If this automaton accepts any input word, then either A_φ or $A_{\neg\varphi}$ does not correctly recognize the language \mathcal{L}_φ or $\mathcal{L}_{\neg\varphi}$, respectively. This suggests that the translation of at least one of the formulas into a Büchi automaton has failed.

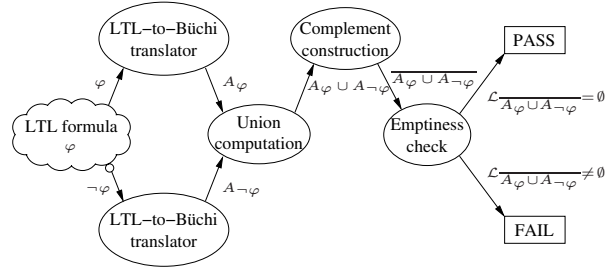


Fig. 2. Universality check for the union of two Büchi automata

trying to keep all algorithms used in the tool as simple as possible to ensure its reliability.) We nevertheless illustrate the steps that would be needed in the test in Fig. 2.

Complementation of Büchi automata is implicit also in LTL model checking, but it is usually avoided by complementing the property formula. For example, even though searching for violations of a property φ in some designated state s of a Kripke structure M actually corresponds to checking the emptiness of the language $\mathcal{L}_{M,s} \cap \overline{\mathcal{L}_\varphi}$ (i.e., the Büchi automaton $A_{M,s} \otimes \overline{A_\varphi}$), the automaton $\overline{A_\varphi}$ can be replaced with the equivalent automaton $A_{\neg\varphi}$, which can be constructed directly from the negated formula. However, a similar approach cannot be used in Test 2 to replace the Büchi automaton $A_\varphi \cup A_{\neg\varphi}$ with an automaton obtained directly from the formula $\neg(\varphi \vee \neg\varphi)$, since this breaks the test's dependency on the automata A_φ and $A_{\neg\varphi}$. Even the result that an implementation translates the formula $\neg(\varphi \vee \neg\varphi)$ correctly into an empty automaton does not necessarily imply that the automata constructed from the formulas φ and $\neg\varphi$ using the same implementation would together accept the universal language. Therefore, a Büchi automata complementation construction is essential for reducing Test 2 correctly to an emptiness check for Büchi automata.

As noted in the beginning of this section, checking the automata A_φ and $A_{\neg\varphi}$ constructed using some LTL-to-Büchi translator for no common accepting or rejecting inputs is not sufficient to show the equivalence between the

languages recognized by A_φ and $A_{\neg\varphi}$ and the languages \mathcal{L}_φ and $\mathcal{L}_{\neg\varphi}$, respectively. However, performing the automata intersection emptiness and the automata union universality tests on Büchi automata constructed using *different* (independent) LTL-to-Büchi translation algorithm implementations can help to increase the confidence in the correctness of the tested implementations. Intuitively, there should only be a small possibility that two (or more) independent LTL-to-Büchi translators have implementation errors that cause them to generate automata recognizing the same language that nevertheless does not correspond to the correct language \mathcal{L}_φ , regardless of the input formula φ . Therefore, using several implementations in the two tests allows validating the automata constructed by the implementations against each other. More specifically, repeating these tests for appropriate pairs of Büchi automata constructed by the implementations will prove that all automata constructed from the same LTL formula accept the same language. However, since it may be unlikely in practice that any of the tested implementations has been proved to be free of errors (by some other means, perhaps), the tests will not formally allow making the conclusion that the automata are then correct even in the total absence of test failures.

Figure 3 illustrates different types of errors that may arise as a result of the incorrect translation of complementary LTL formulas into Büchi automata and how the errors relate to the tests described in this section.

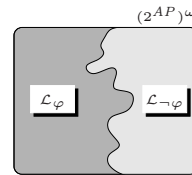
3.3 Using the LTL model checking procedure

The main practical disadvantage of Test 2 is the complexity of the universality test for Büchi automata, both in theoretical sense as well as in terms of implementation difficulty. Unfortunately, this test is clearly necessary to make it possible to detect all types of errors shown in Fig. 3, since the automata intersection emptiness check is not by itself capable of detecting errors in cases where the languages accepted by the automata are disjoint but their union is not the universal language $(2^{AP})^\omega$.

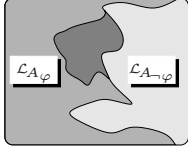
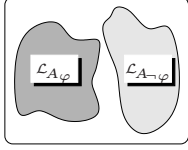
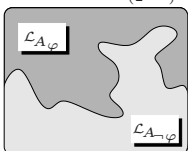
In this section, we describe two additional tests that make use of the automata-theoretic LTL model checking procedure described in Sect. 2.3. Due to their inherent dependency on the input used in the tests (the Kripke structures), these tests may sometimes fail to detect the inconsistencies in a Büchi automaton constructed from some LTL formula. However, the advantage of these methods is their suitability for implementation as straightforward emptiness checks of intersections of Büchi automata; this is a direct consequence of using the LTL model checking procedure, which does not require more complex operations.

3.3.1 Model-checking result cross-comparison

By the semantics of LTL, the truth of an LTL formula φ is well-defined in any Kripke structure M . This fact



Actual relationship between \mathcal{L}_φ and $\mathcal{L}_{\neg\varphi}$

Relationship between languages recognized by two automata A_φ and $A_{\neg\varphi}$	Error	Detectable by
	The languages recognized by the automata are not disjoint	Test 1 (emptiness check of $A_\varphi \otimes A_{\neg\varphi}$)
	The union of the languages recognized by the automata is not the universal language	Test 2 (universality check of $A_\varphi \cup A_{\neg\varphi}$)
	The languages recognized by the automata are complementary but still incorrect	Repeating Tests 1 and 2 on automata constructed using independent implementations [†]

[†] May still fail to detect an error if none of the participating automata generators has been proved to be correct

Fig. 3. Classification of errors resulting from the incorrect translation of either of two complementary LTL formulas φ and $\neg\varphi$ into Büchi automata A_φ and $A_{\neg\varphi}$

makes it possible to test the correctness of LTL-to-Büchi translation algorithm implementations by simply model checking some formula φ in M several times using different LTL-to-Büchi translators for constructing the automaton A_φ required in the model checking process, and finally checking whether the obtained model checking results agree. Assuming that no errors are made in other steps of the model checking procedure (intersecting the formula automaton with the Kripke structure and checking the intersection automaton for emptiness), inconsistent model checking results then imply that at least one of the used LTL-to-Büchi translators failed to perform the formula translation correctly.

The test procedure is shown in Fig. 4. We note that for testing purposes it is not necessary to use models of “real” systems (which tend to be usually very large) as input

Test 3: Model checking result cross-comparison test

Input: LTL formula φ , Kripke structure $M = \langle S, \rho, \pi \rangle$

1. Convert the formula φ into Büchi automata A_φ^i using each LTL-to-Büchi translation algorithm implementation i .
2. Compute the intersection automata $A_M \otimes A_\varphi^i$ (where A_M the automaton described at the end of Sect. 2.3).
3. Perform a global emptiness check on each intersection automaton $A_M \otimes A_\varphi^i$ to obtain sets S_i of states $s \in S$ for which there exists an infinite path x starting from s such that $\pi(x)$ is accepted by A_φ^i .
4. Test whether $S_i = S_j$ for all i, j ($i \neq j$). If this does not hold, one of the LTL-to-Büchi translation algorithms must have failed to translate the formula φ correctly into a Büchi automaton.

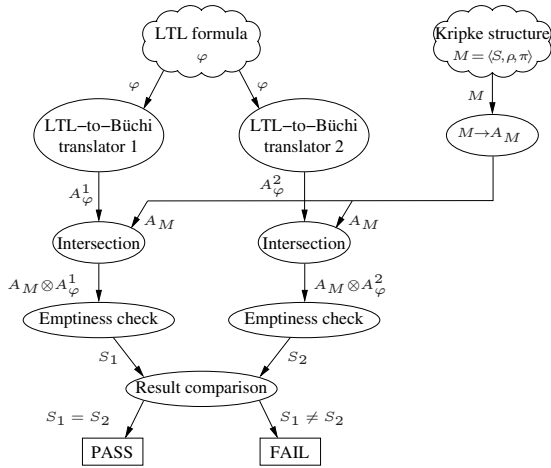


Fig. 4. Model checking result cross-comparison test

for the model checking process, which allows us to reduce the memory and the time requirements for performing tests in practice. Clearly, it is also useful to perform the model checking *globally* (in the sense of the definition in Sect. 2.3) to maximize the amount of data available for comparison tests.

An unsuccessful model checking result cross-comparison test reveals that the languages recognized by some two Büchi automata A_φ^1 and A_φ^2 constructed from the same LTL formula φ are not equivalent. However, the inequivalence of the languages accepted by the automata does not guarantee a test failure: whether any errors are detected in practice depends on the input (the LTL formula φ and the Kripke structure M) used in the model checking procedure. Furthermore, the cross-comparison approach necessitates the use of at least *two* different (independent) implementations for formula translation.

Admittedly, the cross-comparison test does not actually provide any formal evidence that the languages accepted by the automata A_φ^i are equivalent to the expected language \mathcal{L}_φ in case no failures are detected. The

practical value of the test is again justified by the intuitive assumption that independent implementations are unlikely to fail the formula translation in the same way on every input formula. However, as will be discussed in Sect. 4, a *failed* comparison test can always be used to find at least one Büchi automaton that is definitely incorrect, even if none of the participating automata recognizes the language \mathcal{L}_φ accurately.

Using the LTL model checking procedure for testing the correctness of LTL-to-Büchi translators presents the additional need for generating Kripke structures to be used in the model checking process. While it is easy to come up with simple methods for building Kripke structures automatically, it is difficult to estimate how the choice of structures may affect the possibility of finding errors in the implementations. Naturally, increasing the number of states in the structures helps to obtain more data for comparison tests from a single Kripke structure, but the characteristics of Kripke structures that are “optimal” for finding errors in the implementations are likely to depend even on the details of each tested LTL-to-Büchi translator. However, the proposed test methods are independent of any specific way of generating the input, which makes it easy to perform tests even with random formulas and Kripke structures.

3.3.2 Model checking result consistency check

Repeating the model checking result cross-comparison test using another LTL formula or Kripke structure gives an opportunity for further comparison of the implementations. In particular, repeating the test for the negation $\neg\varphi$ of some previously tested LTL formula φ while keeping the Kripke structure fixed provides an additional correctness check for each LTL-to-Büchi translator.

Let $M = \langle S, \rho, \pi \rangle$ be a Kripke structure. A global model check of a formula φ in the structure should find the set S_φ of states $s \in S$ for which there exists an infinite path x (starting from s) through the structure such that $\pi(x) \models \varphi$ holds. Similarly, repeating the global model checking procedure for the negated formula $\neg\varphi$ should result in another set of states $S_{\neg\varphi} \subseteq S$ comprising the initial states of all infinite paths in M that satisfy the formula $\neg\varphi$.

Because each state of the Kripke structure is required to have at least one successor, there is at least one infinite path beginning from each state of the structure. Therefore, by the semantics of LTL, if no infinite path beginning from a state $s \in S$ has the property φ ($s \notin S_\varphi$), then *all* paths must satisfy the property $\neg\varphi$ (that is, $s \in S_{\neg\varphi}$) and vice versa, and thus $S_\varphi \cup S_{\neg\varphi} = S$. If this fact does not hold for the obtained model checking results, the translation of either (or even both) of the formulas φ and $\neg\varphi$ into a Büchi automaton has been performed incorrectly. The test steps are illustrated in Fig. 5. (We remark that due to the possibility that there may exist more than

Test 4: Model checking result consistency check

Input: LTL formula φ , Kripke structure $M = \langle S, \rho, \pi \rangle$

1. Construct the automata A_φ and $A_{\neg\varphi}$ from the formulas φ and $\neg\varphi$ using some LTL-to-Büchi translator.
2. Construct the intersection automata $A_M \otimes A_\varphi$ and $A_M \otimes A_{\neg\varphi}$ (where A_M is the automaton described at the end of Sect. 2.3).
3. Perform a global emptiness check on the intersection automaton $A_M \otimes A_\varphi$ to obtain the set S_φ of states $s \in S$ for which there exists an infinite path x starting from s such that $\pi(x)$ is accepted by A_φ .
4. Repeat Step 3 for the intersection automaton $A_M \otimes A_{\neg\varphi}$. Denote the answers in this case by $S_{\neg\varphi}$.
5. Test whether there exists a state $s \in S$ which is not contained in either of the sets S_φ or $S_{\neg\varphi}$. If such a state exists, the model checking results are inconsistent. This suggests that either the translation of φ or $\neg\varphi$ into a Büchi automaton has failed.

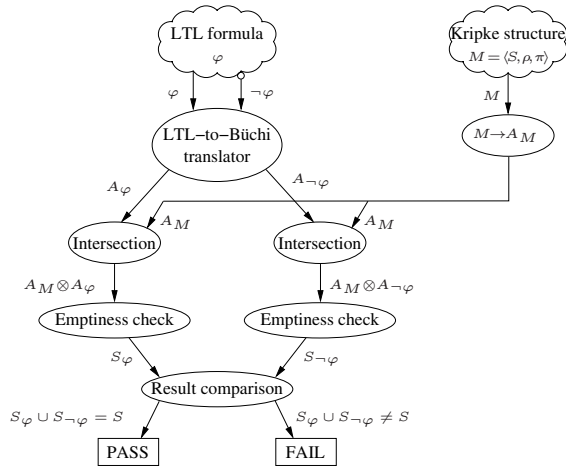


Fig. 5. Model checking result consistency check

one infinite path beginning at the state $s \in S$, it is not necessarily an error if $S_\varphi \cap S_{\neg\varphi} \neq \emptyset$.)

A failed model checking result consistency check implies the existence of an infinite sequence over 2^{AP} that is not accepted by either of the automata A_φ and $A_{\neg\varphi}$ constructed from the formulas φ and $\neg\varphi$, respectively. These errors are of exactly the same type as those that can be detected by Test 2 (applied to the automaton $A_\varphi \cup A_{\neg\varphi}$) described in Sect. 3.2 (see Fig. 3). Therefore, the model checking result consistency check may add to the power of an automated testing tool if the tool only implements Test 1, which is not capable of detecting any of such errors. In addition, unlike the model checking result cross-comparison test, the result consistency check is applicable even when there is only a single LTL-to-Büchi translator available.

Assuming that an automated LTL-to-Büchi translator testing tool implements also Test 3 (the model check-

ing result cross-comparison test), it can be shown that Test 4 then needs to be applied only to automata A_φ and $A_{\neg\varphi}$ generated using the *same* translator. More precisely, performing Test 4 on model checking results obtained using different LTL-to-Büchi translators will not reveal any errors that cannot be detected by Test 3 (between all implementations) combined with a separate result consistency check for each implementation [15].

Finally, we wish to point out that performing the model checking result consistency test requires constructing Büchi automata that can be readily used as input for the tests described in Sect. 3.2. Therefore, all tests described in the current section can be combined together with only a little effort.

4 Test failure analysis

All tests described in the previous section are very similar in the way they are performed. Basically, each test involves first constructing Büchi automata from LTL formulas and from Kripke structures. This phase is followed by pairwise composition of the automata, and finally the results of the composition are checked for expected properties specific to each test.

As a consequence of the need for using several Büchi automata (possibly generated by different LTL-to-Büchi translators), test failures do not directly indicate which automata (i.e., which LTL-to-Büchi translator implementations) involved in the tests actually caused the failures. Even though Test 1 (the Büchi automata intersection emptiness test), Test 2 (the automata union universality test) and Test 4 (the model checking result consistency check) can be used for testing even a single LTL-to-Büchi translator, the mere occurrence of test failures is insufficient information for determining which one of the automata generated by the translator is incorrect. The situation seems even worse with Test 3 (the model checking result cross-comparison test) due to the need for always using different implementations for formula translation.

A method which could be suggested to find an incorrect implementation from many implementations is to search for patterns in the occurrence of failures. For example, if a translator sometimes fails a test against all other tested translators, which in turn pass all tests against each other, it is intuitively likely that there is an error in that one translator. The effectiveness of the approach might be improved by increasing the number of independent LTL-to-Büchi translators taking part in the tests. Unfortunately, this simple approach is not feasible in case there are only a few implementations available or if the implementations are not mutually independent (e.g., if the tested implementations are only different versions of some translator). For this reason, we suggest using an alternative technique that is able to find a *provably* incorrect Büchi automaton by analyzing a single test failure.

Each test involves checking the emptiness of some Büchi automaton as one of its last steps. We know that the nonemptiness of a Büchi automaton $A = \langle \Sigma, Q, \Delta, Q^0, F \rangle$ implies that there exists an infinite sequence $\xi \in \Sigma^\omega$ accepted by the automaton. Such sequences (called *witnesses*) are essential to our approach for distinguishing a faulty LTL-to-Büchi translator from several translators, as the witnesses give the necessary evidence for proving the incorrectness of an implementation. In practice, a witness can be constructed as a side result of the emptiness check that is performed on some nonempty Büchi automaton; see, for example, [9] for details. Even though the witness is an *infinite* sequence, it can always be represented with two *finite* strings over the alphabet Σ : a finite *prefix* followed by a *cycle* that repeats itself indefinitely.

The role of the witness in each of the four presented tests is as follows:

- The failure of Test 1 can be confirmed by exhibiting a witness $\xi \in (2^{AP})^\omega$ that is accepted by the automaton $A_\varphi \otimes A_{\neg\varphi}$ constructed as the intersection of two Büchi automata A_φ and $A_{\neg\varphi}$ corresponding to two complementary LTL formulas φ and $\neg\varphi$, respectively. The witness (that should not be accepted by both automata, by the semantics of LTL) can be constructed during the emptiness check of $A_\varphi \otimes A_{\neg\varphi}$.
- In principle, Test 2 can be performed by checking the emptiness of the automaton $\overline{A_\varphi \cup A_{\neg\varphi}}$ constructed from the union of two Büchi automata. In case the emptiness check for the automaton $\overline{A_\varphi \cup A_{\neg\varphi}}$ fails, the witness found during the check is an infinite sequence over 2^{AP} that is in this case incorrectly *rejected* by one of the automata A_φ and $A_{\neg\varphi}$.
- An inconsistency detected in Test 3 reveals a state s and an infinite path x starting from s (in a Kripke structure $M = \langle S, \rho, \pi \rangle$) such that $\pi(x)$ is accepted by one but rejected by another Büchi automaton constructed from the same LTL formula φ . In this case, the witness is the sequence $\pi(x)$, and it can be constructed during the emptiness check whose result suggests that $\mathcal{L}_{M,s} \cap \mathcal{L}_\varphi \neq \emptyset$.
- A failure in Test 4 gives rise to a witness that corresponds to an infinite sequence rejected by both automata constructed from two complementary LTL formulas (see Sect. 3.3.2). The witness can be obtained as the temporal interpretation of any infinite path starting from a state in which the model checking results are inconsistent.

Each of the above cases considers a pair of Büchi automata, another one of which is definitely known to erroneously accept or reject the witness. Assuming that we know (or can determine) whether the formula φ holds in the witness ξ , we can then definitively distinguish an incorrect automaton from the two automata. Table 1 shows the various possibilities.

For example, when analyzing a failure in Test 1, the witness ξ is an infinite sequence over 2^{AP} accepted by two supposedly complementary Büchi automata A_φ and $A_{\neg\varphi}$ constructed from some formula φ and its negation, respectively. If we find that $\xi \models \varphi$ holds, then the automaton constructed for the negation of the formula is definitely incorrect, since it should reject ξ . However, if $\xi \not\models \varphi$, then the automaton constructed for the positive formula erroneously accepts ξ , and therefore it must be incorrect.

Determining the incorrect automaton from two candidates as shown in Table 1 also reveals a faulty LTL-to-Büchi translation algorithm implementation. However, we are not allowed to assume anything about the correctness of the other implementation. Namely, all that we know about the automaton constructed by that implementation is that it accepts or rejects *one particular witness* as expected. It is possible that the automaton might still not recognize the correct language, but this cannot be confirmed by the analysis.

Using the witness to separate the definitely incorrect implementation from two candidates requires knowledge of whether the formula φ holds in the witness. This can be determined by model checking the formula φ separately in the witness, which can be seen as the temporal interpretation of an infinite path in a Kripke structure whose states are connected into a sequence that ends in a loop (see Fig. 6).

It might seem questionable to use another model checking procedure to determine whether the property φ holds in the witness. Obviously, applying, for ex-

Table 1. Finding an incorrect Büchi automaton using a witness ξ

Test	Büchi automata	Reason for test failure	Incorrect automaton	
			$\xi \models \varphi$	$\xi \not\models \varphi$
1	$A_\varphi, A_{\neg\varphi}$	Both A_φ and $A_{\neg\varphi}$ accept ξ	$A_{\neg\varphi}$	A_φ
2	$A_\varphi, A_{\neg\varphi}$	Neither A_φ nor $A_{\neg\varphi}$ accepts ξ	A_φ	$A_{\neg\varphi}$
3	A_φ^1, A_φ^2	A_φ^1 accepts ξ , but A_φ^2 rejects ξ	A_φ^2	A_φ^1
4	$A_\varphi, A_{\neg\varphi}$	Neither A_φ nor $A_{\neg\varphi}$ accepts ξ	A_φ	$A_{\neg\varphi}$

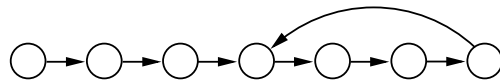


Fig. 6. A Kripke structure whose states are connected into a sequence ending in a loop (state labels omitted)

ample, the full automata-theoretic LTL model checking procedure to the witness requires constructing a Büchi automaton from the formula φ ; unfortunately, we cannot use any of the tested LTL-to-Büchi translators for this purpose if we are uncertain about their reliability.

However, the simple structure of the witnesses permits model checking the formula φ in the witness *directly* using a restricted LTL model checking algorithm designed for Kripke structures that share their shape with the witnesses. This is supported by the fact that in such structures, the semantics of LTL essentially coincides (see an extended version of [10]) with the semantics of *computation tree logic* (see, for example, [9]), for which simple model checking algorithms are available. For example, we can model check the LTL formula φ in the witness by applying a textbook algorithm [9] to the CTL formula $\tau(\varphi)$, where the mapping τ is defined inductively as follows¹:

$$\begin{aligned} \tau(p) &= p \text{ for all atomic propositions } p \in AP \\ \tau(\neg\varphi) &= \neg\tau(\varphi) \\ \tau((\varphi \vee \psi)) &= (\tau(\varphi) \vee \tau(\psi)) \\ \tau(\mathbf{X}\varphi) &= \mathbf{EX}\tau(\varphi) \\ \tau((\varphi \mathbf{U}\psi)) &= \mathbf{E}(\tau(\varphi) \mathbf{U}\tau(\psi)) \end{aligned}$$

More specifically, it is straightforward to show by structural induction on the formula that $\xi \models \varphi$ iff $M, s_0 \models \tau(\varphi)$, where s_0 is the state without any predecessors in the Kripke structure that forms the witness. (Alternatively, one can use a direct LTL model checking algorithm for witnesses, such as the one described in [15].)

We point out that a direct witness model checking algorithm can have more general application even in real LTL model checking tool implementations. For example, such an algorithm can easily be used to validate counterexamples (examples of computation paths that violate a given LTL property) that a model checking tool might find during the analysis of some system to be verified. Integrating a counterexample validation algorithm into a model checker provides the tool with basic capabilities of detecting automatically some of the situations that might result in model checking errors, more specifically, by preventing the tool from suggesting incorrect counterexamples for the verified properties.

5 A randomized testbench for LTL-to-Büchi translators

We have implemented most of the test methods discussed in Sect. 3, excluding Test 2 (the universality check for the union of two Büchi automata, see Sect. 3.2), into

¹ Actually, the formula transformation is independent of the particular path quantifier used in the CTL formulas due to the fact that the formulas will be model checked in restricted Kripke structures, each state of which has exactly one successor. Thus the **A** quantifier could well be used instead of the **E** quantifier.

a software tool (a “testbench”) for automatic testing of LTL-to-Büchi translation algorithm implementations. The testbench (an extended version of those described in [14, 16]) also includes an implementation of a restricted LTL model checking algorithm used in the analysis of test results to determine which of the tested LTL-to-Büchi translators failed, as described in Sect. 4. The C++ source code for the program can be obtained from <URL: <http://www.tcs.hut.fi/~htauriai/lbtt/>>.

The testbench uses randomized algorithms for generating LTL formulas and Kripke structures needed as input for the tests. The overall goal in the design of the algorithms was to keep them simple. We did not consider the precise distribution of the output generated by the algorithms, for example, that the random LTL formulas should be distributed “uniformly” in some probabilistic sense. (However, see [15] for a description on how the parameters of the formula generation algorithm can be adjusted to ensure that each generated formula will have the same expected number of occurrences of each operator.) After all, since none of the described tests can be used to prove exhaustively the correctness of an LTL-to-Büchi translator, a “poor” sample of input will even in the worst case only result in a suboptimal test failure rate. The algorithms used in the testbench are shown as examples in Figs. 7 and 8. We point out the following features of the algorithms (for a more detailed discussion, see [15]):

```

1 function RandomFormula ( $n$  : Integer): LtlFormula
2 if  $n = 1$  then begin
3    $p :=$  random symbol in  $AP \cup \{\top, \perp\}$ ;
4   return  $p$ ;
5 end
6 else if  $n = 2$  then begin
7    $op :=$  random unary operator;
8    $\varphi :=$  RandomFormula(1);
9   return  $op \varphi$ ;
10 end
11 else begin
12    $op :=$  random operator in the set of all available
      operators;
13   if  $op$  is unary then begin
14      $\varphi :=$  RandomFormula( $n - 1$ );
15     return  $op \varphi$ ;
16   end
17   else begin (*  $op$  is binary *)
18      $x :=$  random integer in the interval  $[1, n - 2]$ ;
19      $\varphi :=$  RandomFormula( $x$ );
20      $\psi :=$  RandomFormula( $n - x - 1$ );
21     return  $(\varphi \ op \ \psi)$ ;
22   end;
23 end;
24 end;

```

Fig. 7. Pseudocode for the LTL formula generation algorithm

```

1 function RandomGraph( $n$  : Integer,  $d$  : Real  $\in$  [0.0, 1.0],
    $t$  : Real  $\in$  [0.0, 1.0])
   : KripkeStructure
2    $S := \{s_0, s_1, \dots, s_{n-1}\}$ ;
3    $NodesToProcess := \{s_0\}$ ;
4    $UnreachableNodes := \{s_1, s_2, \dots, s_{n-1}\}$ ;
5    $\rho := \emptyset$ ;
6   while  $NodesToProcess \neq \emptyset$  do begin
7      $s :=$  a random node in  $NodesToProcess$ ;
8      $NodesToProcess := NodesToProcess \setminus \{s\}$ ;
9      $\pi(s) := \emptyset$ ;
10    for all  $P \in AP$  do
11      if RandomNumber(0.0, 1.0)  $< t$  then
12         $\pi(s) := \pi(s) \cup \{P\}$ ;
13    if  $UnreachableNodes \neq \emptyset$  then begin
14       $s' :=$  a random node in  $UnreachableNodes$ ;
15       $UnreachableNodes :=$ 
16         $UnreachableNodes \setminus \{s'\}$ ;
17       $NodesToProcess :=$ 
18         $NodesToProcess \cup \{s'\}$ ;
19       $\rho := \rho \cup \{(s, s')\}$ ;
20    end;
21    for all  $s' \in S$  do
22      if RandomNumber(0.0, 1.0)  $< d$  then begin
23         $\rho := \rho \cup \{(s, s')\}$ ;
24        if  $s' \in UnreachableNodes$  then begin
25           $UnreachableNodes :=$ 
26             $UnreachableNodes \setminus \{s'\}$ ;
27           $NodesToProcess :=$ 
28             $NodesToProcess \cup \{s'\}$ ;
29        end;
30      end;
31    if there is no edge  $(s, s')$  in  $\rho$  for any  $s' \in S$  then
32       $\rho := \rho \cup (s, s)$ ;
33    end;
34    return  $\langle S, \rho, \pi \rangle$ ;
35  end;

```

Fig. 8. Pseudocode for the Kripke structure generation algorithm

- The LTL formula generation algorithm (Fig. 7) follows an approach similar to the one introduced in [5] by generating formulas with a parse tree having a given number n of nodes. The behavior of the algorithm can be altered by restricting the set of operators to be used in the formulas. Besides the basic LTL operators \neg , \vee , \times , and \cup , the testbench supports several other common operators that can be defined using these basic operators, such as the logical connectives \wedge (conjunction), \rightarrow (implication), and \leftrightarrow (equivalence) and the temporal operators \diamond (eventually), \square (always), and R (release, the dual of \cup). In addition, the Boolean constants \top (true) and \perp (false) can be used in place of atomic propositions.
- The random Kripke structure generation algorithm (Fig. 8) generates structures with a connected com-

ponent consisting of a given number n of states with the constraint that every state of the structure must have at least one successor. The behavior of the structure generation algorithm can be adjusted with the parameter d that approximates the probability of having a transition between any pair of states in the structure, together with the parameter t that gives the probability with which any of the atomic propositions is considered to hold in any state of the structure.

- As an alternative to graph-like Kripke structures, the testbench also allows using randomly generated *paths*, i.e., Kripke structures in which the states are connected into a linear sequence ending in a transition back to some earlier state of the sequence (see Fig. 6). This permits using the *restricted witness model checking* algorithm as another algorithm in Test 3 (the model checking result cross-comparison check), which makes it possible to apply all tests even to a single translation algorithm implementation.

After generating an LTL formula and a Kripke structure, the testbench invokes every tested LTL-to-Büchi translator in turn to obtain Büchi automata from the formula and its negation. The testbench then performs Test 1, Test 3, and Test 4 on the automata, repeating Tests 1 and 3 for each appropriate pair of automata generated by two different implementations. The test procedure can then be repeated using a different LTL formula or a different Kripke structure.

Should a test result in a failure, the testbench can be instructed to use the internal model checking algorithm for witnesses to find an incorrect Büchi automaton. To justify its answer, the testbench also shows a proof whether the formula holds in the witness or not, and, if an automaton incorrectly accepts the witness, an accepting execution that the automaton has when given the witness as input.

6 Experimental results

The test methods were experimented on the following independent LTL-to-Büchi translator implementations:

ÅSA+ An LTL-to-Büchi translator derived from Mauno Rönkkö's C++ class library [12] implementing the translation algorithm presented in [7].

SPIN v3.x.x The LTL-to-Büchi translator algorithm implemented in the model checker SPIN [8] by Gerard J. Holzmann. The implementation is originally based on the translation algorithm in [7], but it includes many improvements to the basic algorithm, some of which are described in [6].

Tests 3 and 4 have been applied to testing this implementation already since its version 3.3.3. The tests have been able to find errors in various versions of the tool [14, 16]. To illustrate the effect of testing on improving the robustness of the tool, we included versions 3.3.3, 3.3.9, 3.4.3 and 3.4.7 of the tool in the

experiments. We however note that this might have created a bias in the results, making SPIN perhaps seem more unreliable than the other implementations. We emphasize that the behavior of SPIN v3.4.7 was consistent with all the other implementations in our tests.

LTL2AUT An LTL-to-Büchi translator written by the authors of [5]. The implementation includes three different translation algorithms: the previously mentioned GPVW algorithm [7], the GPVW+ algorithm based on some improvements proposed already in [7], and the LTL2AUT algorithm of [5]. We tested the implementation using all the available algorithms.

PROD The LTL-to-Büchi translator included in the Pr/T net reachability analyzer PROD [21]. The implementation is based on the algorithm presented in [19]. The version of the tool used for the tests was from 21 February 2001 (PROD 3.3.09). The testing method has also helped to improve a previous version of this implementation.

We used the same testing environment and testing strategy as in [15]. The tests were performed by running each of the translators on randomly generated fixed-size (5–12 parse tree nodes) LTL formulas and their negations using 1000 random formulas of each parse tree size with at most five different atomic propositions in each formula. For Tests 3 and 4 we used Kripke structures generated with the algorithm shown in Fig. 8 using parameters $n = 50$, $d = 0.1$, and $t = 0.5$.

Because the PROD implementation supported only a subset of the logical and temporal operators that could have been used with the other implementations, we restricted the operator set to $\{\neg, \vee, \wedge, \rightarrow, \text{U}, \diamond, \square\}$. The formula generation parameters were adjusted for each parse tree size such that each generated formula had the same expected number of occurrences of each different operator [15].

The test results are shown in Tables 2 and 3. Each cell of Table 2 contains two numbers giving the failure rates in Test 1 (the Büchi automata intersection emptiness check) and Test 3 (the model checking result cross-comparison test) between a pair of implementations. Every table cell, excluding the diagonal cells of the matrices corresponding to each formula parse tree size, gives the number of failed tests among a maximum of 2000 (in the bottom row, 16 000) tests performed. (Because the matrices are symmetric, the numbers above the diagonal of each matrix have been omitted.) Since Test 3 is not relevant when both automata involved in the test are generated by the same implementation, the diagonal cells report only the failure rates in Test 1, which could be performed only half as many times on each implementation against itself.

Because ÅSA+, LTL2AUT (all three variants), PROD and version 3.4.7 of the SPIN model checker did not fail any of the tests between each other, the results of these

Table 2. Failure rates for Tests 1 and 3

Random formula parse tree size	Implementation	Number of test failures $\langle \text{Test 1 failures} \rangle / \langle \text{Test 3 failures} \rangle$			
		[Diagonal cells: 1000 tests; other cells: 2000 tests]			
		Å/L/P/7 [†]	SPIN v3.3.3	SPIN v3.3.9	SPIN v3.4.3
5	Å/L/P/7	0 / 0			
	S3.3.3	21 / 4	20 / -		
	S3.3.9	0 / 0	21 / 4	0 / -	
	S3.4.3	0 / 0	21 / 4	0 / 0	0 / -
6	Å/L/P/7	0 / 0			
	S3.3.3	32 / 8	31 / -		
	S3.3.9	0 / 0	32 / 8	0 / -	
	S3.4.3	0 / 0	32 / 8	0 / 0	0 / -
7	Å/L/P/7	0 / 0			
	S3.3.3	49 / 9	48 / -		
	S3.3.9	0 / 0	49 / 9	0 / -	
	S3.4.3	0 / 0	49 / 9	0 / 0	0 / -
8	Å/L/P/7	0 / 0			
	S3.3.3	66 / 14	61 / -		
	S3.3.9	0 / 1	65 / 13	0 / -	
	S3.4.3	0 / 0	66 / 14	0 / 1	0 / -
9	Å/L/P/7	0 / 0			
	S3.3.3	69 / 16	69 / -		
	S3.3.9	0 / 0	69 / 16	0 / -	
	S3.4.3	0 / 0	69 / 16	0 / 0	0 / -
10	Å/L/P/7	0 / 0			
	S3.3.3	73 / 19	66 / -		
	S3.3.9	0 / 1	71 / 18	0 / -	
	S3.4.3	0 / 0	73 / 19	0 / 1	0 / -
11	Å/L/P/7	0 / 0			
	S3.3.3	87 / 19	83 / -		
	S3.3.9	0 / 2	87 / 17	0 / -	
	S3.4.3	0 / 0	87 / 19	0 / 2	0 / -
12	Å/L/P/7	0 / 0			
	S3.3.3	101 / 31	91 / -		
	S3.3.9	0 / 3	102 / 32	0 / -	
	S3.4.3	0 / 1	102 / 30	0 / 2	0 / -
TOTAL	Å/L/P/7	0 / 0			
	S3.3.3	498 / 120	469 / -		
	S3.3.9	0 / 7	496 / 117	0 / -	
	S3.4.3	0 / 1	499 / 119	0 / 6	0 / -

[†]ÅSA+, LTL2AUT, PROD, SPIN v3.4.7

algorithms are shown in the tables under the common name “Å/L/P/7”. The independence of the implementations increases the confidence in their reliability. However, all previous versions of SPIN occasionally failed the test against the other tested implementations.

For example, version 3.4.3 of SPIN was involved in a single model checking result cross-comparison failure on the LTL formula $\square\square(p_4 \wedge (p_2 \text{U} (\neg\neg p_3 \wedge \diamond p_4)))$. Analyzing the failure with the witness model checking algorithm confirmed an error in the implementation: it is easy to check that the formula φ holds in the infinite se-

Table 3. Failure rates for Test 4

Random formula parse tree size	Å/L/P/7 [†] SPIN v3.3.3 SPIN v3.4.3	SPIN v3.3.9
5	0	0
6	0	0
7	0	0
8	0	1
9	0	0
10	0	1
11	0	1
12	0	2
TOTAL	0	5

[†]ÅSA+, LTL2AUT, PROD, SPIN v3.4.7

quence $\xi = \langle \{p_1, p_3, p_4\}, \{p_1, p_3, p_4\}, \{p_1, p_3, p_4\}, \dots \rangle$; on this input, however, the automaton generated by the SPIN v3.4.3 implementation can only loop through its nonaccepting single initial state indefinitely, thus rejecting the input. This confirms that the automaton generated by the translator is incorrect. While it can be argued that the randomly generated formula, being equivalent to the more simple formula $\Box(p_4 \wedge (p_2 \cup p_3))$ (which SPIN v3.4.3 does not translate to an automaton that causes a test failure), has redundancy in it, the faults detected in any implementation should nevertheless be always repaired to remove the possibility that they might cause model checking errors. The results show that the error behind the example case has been fixed in SPIN v3.4.7.

The failure rates in Test 1 are higher than those of Test 3, which suggests that Test 3 might be less efficient in detecting errors in practice, perhaps due to its need for more input, the properties of which may seriously affect the efficiency of the testing procedure. However, Test 1 is by itself not capable of detecting all types of errors (see Fig. 3), as evidenced also in practice by the example above. Other experiments [15] have observed the error rates to be quite close to each other when using a different set of formula operators; it is clear, however, that the effect of the formula generation parameters on the test failure rate always depends also on the implementations participating in the tests and cannot therefore be estimated easily in any general way.

Test 4 (the model checking result consistency test) had significantly worse performance than either of the two other tests and could detect errors only in SPIN v3.3.9. However, because of the many factors that may all affect the tests, it is still recommended to use all the available tests to maximize testing efficiency, since each of these three tests is capable of detecting errors that cannot be detected by the two other tests. This view is further supported by the fact that all these tests are straightforward to implement into a common framework. How-

ever, if Test 2 were also available, Tests 3 and 4 would no longer be needed, since Test 2 finds all errors detectable by Test 4. In addition, Tests 1 and 2 could be used together to prove the equivalence of languages accepted by two automata constructed using different translators, thus making also Test 3 redundant.

We refer to [15] for further experimental results, together with more detailed discussion on the different types of failures encountered in the tests.

7 Conclusions

Model checkers are often used to verify systems which are critical in either a safety or business sense. Since these tools are used to reason about the properties of such systems, it is essential that their user is able to rely on the verification results reported by the tool.

We have presented methods that can be used for testing implementations of LTL-to-Büchi translation algorithms, which form one of the core components of any LTL model checker based on the automata-theoretic model checking approach [19]. The test methods are based on simple automata-theoretic techniques and the LTL model checking procedure. Most of the methods (excluding Test 2 described in Sect. 3.2) are easy to implement using basic model checker implementation techniques.

Test failure analysis has been presented. It uses a witness (i.e., an infinite sequence on which the test failure exists) to prove one of the implementations incorrect. This analysis employs a special, simple-to-implement model checking procedure to check whether an LTL formula holds in the witness.

We have combined the test methods and failure analysis to an automated testbench. This testbench uses random formulas and Kripke structures to test several independent LTL-to-Büchi translator implementations against each other. The testbench implementation was able to uncover errors in several versions of the SPIN model checker. We have obtained similar results with other model checkers, too; of the implementations we have tested, the LTL2AUT [5] and the ÅSA [12] translators have been the only implementations in which no errors have ever been detected using the testbench.

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, the emptiness checking and partial order reduction algorithms). However, we think that the LTL-to-Büchi translation algorithm, including all the possibly used optimizations, is one of the most difficult algorithms to implement.

In the future, we would like to extend the approach to also test nondeterministic finite automata (NFA) generated from safety LTL formulas using the approaches presented in [10]. These (safety) LTL-to-NFA translation algorithm implementations would need a different

emptiness checking subroutine. The model checking results obtained could be then compared against the results obtained using the model checking procedure employing the LTL-to-Büchi translation algorithm for the same formula.

We propose that LTL model checkers should be extended with a counterexample validation algorithm as described at the end of Sect. 4. The implementation of this algorithm is quite simple, as it can be done, for example, by using a straightforward implementation of a CTL model checker. The running time overhead of such an algorithm should be 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. 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.

Admittedly, simple random testing is not adequate for *proving* the correctness of any LTL-to-Büchi translator. As can be seen in the experiments made with various versions of SPIN, the effectiveness of random testing in finding errors decreases rapidly as the implementation improves. Increasing the number of tests or the size of the formulas and Kripke structures used in the tests might improve the odds of finding errors, but the fact that no amount of testing is sufficient to prove the absolute correctness of an implementation makes this approach somewhat unappealing.

Therefore, the testing techniques are probably best suited for assisting in the development of a new translator to test its robustness before releasing the implementation, in the hope of detecting some of the remaining errors and omissions in the implementation. The test methods might also be applicable to regression testing when making optimizations or other improvements to a translation algorithm implementation, in order to test whether the implementation seems to preserve its reliability between different releases.

Acknowledgements. We would like to thank Mauno Rönkkö, Kimmo Varpaaniemi, Marco Daniele, and Gerard J. Holzmann for helping us to integrate their respective LTL-to-Büchi translation algorithm implementations in the experiments made in this work. We are also grateful to the anonymous referees of the SPIN'2000 Workshop and STTT, whose feedback was very important in improving this work.

References

- Avizienis, A.: The N -version approach to fault-tolerant software. *IEEE Trans Software Eng* SE-11(12):1491–1501, 1985
- Avizienis, A., Chen, L.: On the implementation of N -version programming for software fault tolerance during program execution. In: Proc. 1st IEEE International Computer Software and Applications Conference (COMPSAC 77), pp. 149–155, 1977
- Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. *Formal Methods Syst Design* 1:275–288, 1992
- Couvreur, J.-M.: On-the-fly verification of linear temporal logic. In: Proc. World Congress on Formal Methods in the Development of Computing Systems (FM'99), volume I, Lecture Notes in Computer Science, vol. 1708. Springer, Berlin Heidelberg New York, 1999, pp. 253–271
- Daniele, M., Giunchiglia, F., Vardi, M.Y.: Improved automata generation for linear temporal logic. In: Proc. 11th International Conference on Computer Aided Verification (CAV'99), Lecture Notes in Computer Science, vol. 1633. Springer, Berlin Heidelberg New York, 1999, pp. 249–260. See also: “Software packages” at <http://www.cs.rice.edu/CS/Verification/>
- Etesami, K., Holzmann, G.: Optimizing Büchi automata. In: Proc. 11th International Conference on Concurrency Theory (CONCUR'2000), Lecture Notes in Computer Science, vol. 1877. Springer, Berlin Heidelberg New York, 2000, pp. 153–167
- Gerth, R., Peled, D., Vardi, M.Y., Wolper P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proc. 15th Workshop Protocol Specification, Testing, and Verification, pp. 3–18. Chapman and Hall, London, 1995
- Holzmann, G.: The model checker SPIN. *IEEE Trans Software Eng* 23(5):279–295, 1997
- Clarke, E. Jr., Grumberg, O., Peled, D.: Model checking. MIT, Boston, Mass., USA, 1999
- Kupferman, O., Vardi, M. Y.: Model checking of safety properties. In: Proc. 11th International Conference on Computer Aided Verification (CAV'99), Lecture Notes in Computer Science, vol. 1633. Springer, Berlin Heidelberg New York, 1999, pp. 172–183. See also an extended version at <http://www.cs.rice.edu/vardi/papers/>
- Latvala, T., Heljanko, K.: Coping with strong fairness. *Fundam Inform* 43(1–4):175–193, 2000
- Rönkkö, M.: A distributed object oriented implementation of an algorithm converting a LTL formula to a generalised Buchi automaton, 1999. Available only on the WWW. See Mauno Rönkkö's homepage at <http://www.abo.fi/~mauno.ronkko/>
- Safra, S.: On the complexity of ω -automata. In: Proc. 29th IEEE Symposium on Foundations of Computer Science, pp. 319–327. IEEE Computer, New York, 1988
- Tauriainen, H.: A randomized testbench for algorithms translating linear temporal logic formulae into Büchi automata. In: Proc. Workshop Concurrency, Specification and Programming 1999 (CS&P'99), pp. 251–262. Warsaw University, 1999
- Tauriainen, H.: Automated testing of Büchi automata translators for linear temporal logic. Technical Report A66, Laboratory for Theoretical Computer Science, Helsinki University of Technology, 2000. Available at <http://www.tcs.hut.fi/Publications/reports/A66abstract.html>
- Tauriainen, H., Heljanko, K.: Testing SPIN's LTL formula conversion into Büchi automata with randomly generated input. In: Proc. 7th International SPIN Workshop on Model Checking of Software (SPIN'2000), Lecture Notes in Computer Science, vol. 1885. Springer, Berlin Heidelberg New York, 2000, pp. 54–72
- Thomas, W.: Languages, automata and logic. In: Rozenberg, G., Salomaa, A. (eds.), *Handbook of formal languages*, volume 3, pp. 385–455. Springer, Berlin Heidelberg New York, 1997
- Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: *Logics for concurrency: structure versus automata*, Lecture Notes in Computer Science, vol. 1043. Springer, Berlin Heidelberg New York, 1996, pp. 238–265
- Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. 1st IEEE Symposium on Logic in Computer Science (LICS'86), pp. 332–344. IEEE Computer, New York, 1986
- Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. *Inf Comput* 115(1):1–37, 1994
- Varpaaniemi, K., Heljanko, K., Lilius, J.: PROD 3.2 - An advanced tool for efficient reachability analysis. In: Proc. 9th International Conference on Computer Aided Verification (CAV'97), Lecture Notes in Computer Science, vol. 1254. Springer, Berlin Heidelberg New York, 1997, pp. 472–475