Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 66
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 66
Espoo 2000 HUT-TCS-A66

AUTOMATED TESTING OF BÜCHI AUTOMATA TRANSLATORS FOR LINEAR TEMPORAL LOGIC

Heikki Tauriainen

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 66
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 66
Espoo 2000 HUT-TCS-A66

AUTOMATED TESTING OF BÜCHI AUTOMATA TRANSLATORS FOR LINEAR TEMPORAL LOGIC

Heikki Tauriainen

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-5275-9
ISSN 1457-7615

Picaset Oy
Helsinki 2000

ABSTRACT: The formal verification of finite-state reactive and concurrent systems against temporal logical requirements can be done by model checking, which has the advantage of being well suited for automation. However, reasoning about the correctness of systems using automated techniques places high demands for ensuring the reliability of the model checking tools themselves.

This work describes testing methods for detecting implementation errors in a specific class of algorithms required in the automata-theoretic model checking procedure for propositional linear temporal logic (LTL). These algorithms translate temporal requirements into Büchi automata that are used in the model checking process. Most of the test methods can be easily integrated into an automatic testing tool for translation algorithm implementations. Experimental results using a randomized tool for testing the correctness of several implementations included in real model checkers are presented. This testing has proved to be an effective method for finding implementation errors in the translators.

This work also presents a restricted LTL model checking algorithm designed to work in a very simple subclass of systems, on which the analysis of test failures is based. This algorithm helps to automatically confirm the incorrectness of a translation algorithm implementation.

KEYWORDS: Model checking, linear temporal logic, Büchi automata, algorithm testing

CONTENTS

1 INTRODUCTION

The goal of *verification* is to show that a given hardware or software system conforms to its specifications and cannot behave in ways that might lead into unexpected, undesirable or even critical situations. *Formal* verification methods try to achieve this goal by *proving* that unintended behaviour in the system is theoretically impossible. For example, these techniques can be used in hardware system design to check the correctness of hardware specifications before actually building the system. This may even reduce the overall production costs by removing (or decreasing) the possibility that design errors will need to be fixed in the finished product.

Model checking [1, 21] is one of the techniques applied especially in the formal verification of reactive and concurrent systems and their specifications, e.g. data communications protocols. This technique operates on a *model* built from the original specifications of the system to be verified. Basically, the model is a (possibly abstracted) representation of the original system, and its behaviour reflects the system's behaviour in light of a given property to be verified. Verification then proceeds with checking whether this formal model has the given property.

Also the properties to be verified need to be stated in a form that supports expressing requirements on the system model in terms of the chosen modelling formalism. Model checking makes wide use of various *temporal logics* for expressing these requirements as *formulae* of a chosen temporal logic. The requirements concern the system's behaviour as time passes (e.g., the relative order of events observed in the system), and they may include temporal concepts such as "always", "eventually" or "infinitely often". Three commonly used temporal logics in model checking are the computation tree logic (CTL), the linear temporal logic (LTL), and the full branching time logic (CTL*). (For a detailed review of all these logics, see e.g. [11].)

Different logics have different expressive power, which affects the nature of the properties that can be expressed in the logic. The variety in the expressiveness of different logics results in a large number of model checking techniques, some of which may be applicable only to certain logics. This work concentrates on model checking *propositional linear temporal logic* [20] with techniques based on the general *automata-theoretic approach* to model checking due to Vardi and Wolper [31, 30].

Like any other complex task that requires high preciseness, model checking specifications of real systems is made easier through the use of *automated tools* for performing the task. Model checking techniques translate quite readily into general verification *procedures* that can in principle be easily automated. Practical tools with abilities for model checking various temporal logics include the model checker SPIN [10] designed for the verification of protocols, the PROD tool [33, 34] for the analysis of systems modelled as Predicate/Transition nets [7], and the SMV hardware system model checker [18] based on symbolic verification techniques (see e.g. [11]).

Clearly, the correctness of the results given by any software tool that is used to reason about the properties of some system (e.g., another piece of software) is highly dependent on the correctness of the tool implementation itself. Proving the tool implementation correct using automated techniques would certainly be very desirable. However, model checking tools are often complex pieces of software themselves, and their full verification is still out of reach of current algorithmic verification techniques in practice. In order to alleviate the unavoidable *state explosion problem* (see e.g. [29]) that makes model checking of complex systems difficult in practice, model checking tools have to use many nontrivial techniques for performing their task in a memory-efficient way. Unfortunately, these techniques may increase the complexity of the model checking tools themselves, which makes them more prone to implementation errors.

Testing and simulation are common methods for examining the behaviour and reliability of systems whose detailed analysis may otherwise be too complicated. Even very informal testing techniques can be of valuable help in uncovering errors in software. Since it may be difficult to prove implementations of model checking algorithms correct automatically, testing can offer a simple approach applicable to improving the robustness of implementations of the algorithms used in model checking tools.

LTL model checking tools based on the automata-theoretic approach usually employ a translation of linear temporal logic properties into finite-state automata over infinite words (Büchi automata). In comparison to the other phases of LTL model checking, the formula translation phase can be relatively hard to implement. Errors in the implementation of a translation algorithm may create a source of model checking errors that may degrade the reliability of the tool. This work describes methods that can be used for testing this phase of the model checking procedure for LTL (referred to as *LTL-to-Büchi translation*). The testing methods can be automated into a software package for testing real implementations of translation algorithms—even those used in real model checking tools. This work describes an implementation of some of these methods into a randomized testbench for LTL-to-Büchi translators (an extended version of the one described in [26, 27]), together with experimental results of tests made on several independent translation algorithm implementations. To improve the capabilities of detecting errors in the translators, the testbench makes use of an LTL model checking algorithm for a restricted class of system models.

The following two chapters introduce the formalisms used to represent the system models and the properties to be verified. Chapter 4 reviews the automata-theoretic model checking procedure for linear temporal logic, which forms the core of several test methods for LTL-to-Büchi translation algorithms. The test methods themselves are described in Chap. 5. The chapter also includes a description of an LTL model checking algorithm for the restricted class of models that arises in the analysis of test results. The algorithm enhances the power of the test methods by providing a way to confirm the failure of a particular LTL-to-Büchi translator. Chapter 6 presents the results of applying some of the test methods to several real LTL-to-Büchi translation algorithm implementations. The work ends with some conclusions in Chap. 7. The four appendices contain details on some issues mentioned only briefly in the text, such as the correctness proof of the restricted LTL model checking algorithm described in Chap 5.

2 STATE-TRANSITION MODELS OF SYSTEMS

Model checking techniques traditionally assume the system description to be given as a finite *state-transition graph*. The system is thought to have a (unique) *state* at each instant of its operation, and it operates in discrete steps by making *transitions* from a state to another state. The model of the system is built by exhaustively enumerating all the possible states that the system can ever visit during its operation (called the *state space* of the system). Transitions are then added between the states to represent all the possible ways in which the system can change its state during operation.[1] Because the operation of the system may vary according to its inputs, it can behave in many different ways. Each of these ways is individually called a *behaviour* (or equivalently, an *execution*) of the system. The model built from the system captures *all* these possibilities such that any actual operation observed in the system can be represented as an execution of the model.[2] An execution can be described as a sequence of states the system visits during its operation, or, alternatively, as a sequence of transitions the system makes when moving from one state to another. In this work, executions of the system will always be treated as sequences of states. The system is assumed to have a unique *initial state* where it begins its operation. In addition, the models are assumed to have a finite state space.

Model checking a given property in the system requires the ability to distinguish between the executions of the system with respect to the property. This is done by augmenting each individual system state with information describing the characteristics of the state. The characteristics of any system execution are then determined by the characteristics of the states occurring in the execution. The information associated with the states can be expressed in temporal logic by using a set of *atomic propositions*, each of which is given a fixed truth value in each individual system state. The propositions acquire their semantics from the original system specification and the property to be checked in the model.

An additional assumption concerning the executions of the system is that every individual execution of the system is always infinite. This is a reasonable assumption about a reactive system (e.g., a server protocol) that should continue responding to its inputs indefinitely. We therefore deny the possibility of any finite terminating behaviours in the system model to simplify the discussion. If the system has any finite behaviours, they can be interpreted as infinite behaviours in which the system will stay forever in the final state of the terminating behaviour once reaching it. A system model can be augmented with extra transitions and states in order to make it satisfy this requirement. Additional atomic propositions might also need to be introduced

[1]In practice, the system specifications are often given in a more high-level notation, using e.g. Pr/T nets [7], process algebras (e.g. CCS [19]) or various tool-specific specification languages (e.g. PROMELA [9]). The Kripke structures described here can be considered low-level semantical interpretations of the system descriptions; if necessary, they can be built even automatically from various high-level specifications.

[2]In the discussion, the system is often identified with its model. Therefore, we will often speak of executions of the system when actually referring to executions of the model.

if the goal is to check for the reachability of these states.

Formally, the system models are defined as node-labelled directed graphs, called *Kripke structures* in the model checking context. In order to exclude from the graph any state sequences that cannot be extended into infinite ones by repeatedly appending states to the end of the sequence, every state of the structure is required to have at least one successor. This means that the transition relation is total.

Let AP denote a given nonempty finite set of atomic propositions describing the properties of the system states.

Definition 1 (Kripke structures) *A Kripke structure is a quadruple* $M = \langle S, \rho, s^0, \pi \rangle$, *where*

- S *is a finite set of* states,
- $\rho \subseteq S \times S$ *is a* transition relation *that satisfies the condition* $\forall s \in S : \exists s' \in S : (s, s') \in \rho$,
- $s^0 \in S$ *is the* initial state, *and*
- $\pi : S \mapsto 2^{AP}$ *is a* labelling function *that associates each individual 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* $\langle s_0, s_1, s_2, \dots \rangle \in S^{\omega}$ [3] *such that* $(s_n, s_{n+1}) \in \rho$ *for all* $n \geq 0$. ■

In order to reason about the properties of the executions of the system model, it is useful to consider only those paths that begin in the initial state of the Kripke structure. These are the paths that correspond to the executions of the system. Given a Kripke structure $M = \langle S, \rho, s^0, \pi \rangle$, the set $\{\langle s_0, s_1, s_2, \dots \rangle \in S^{\omega} \mid s_0 = s^0 \text{ and } (s_i, s_{i+1}) \in \rho \text{ for all } i \geq 0\}$ is called the set of *executions* (or *behaviours*) of the structure.

The state-labelling function π can be used to project any path in the Kripke structure onto an infinite sequence of labels of the states in the sequence. These sequences of state labels can be considered infinite *words* whose "letters" are subsets of AP, and the set of all letters (2^{AP}) is called the *alphabet*. Moreover, any subset $\mathcal{L}$ of $(2^{AP})^{\omega}$ can be considered a *language* of infinite words over the alphabet 2^{AP}. In particular, we will denote by $\mathcal{L}_M$ the set of words corresponding to the executions of a given Kripke structure M, and we say that $\mathcal{L}_M$ is *generated* by M. This language analogy will be used later in Chap. 4 when discussing automata-theoretic model checking of linear temporal properties in Kripke structures.

The following example demonstrates the different concepts described above.

[3]For any nonempty set X, X^{ω} denotes the set of all infinite sequences that can be constructed from the elements of X.

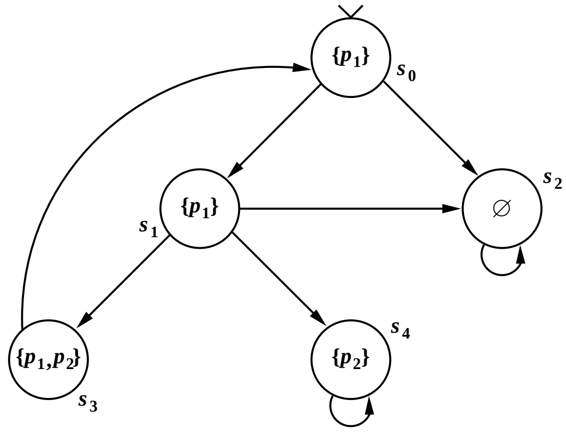


Fig. 2.1: A simple Kripke structure

Example 1 *Let* $AP = \{p_1, p_2\}$ *and let* $M = \langle S, \rho, s^0, \pi \rangle$ *be the Kripke structure defined as follows:*

$$\begin{aligned}
S &= \{s_0, s_1, s_2, s_3, s_4\},\\
\rho &= \{(s_0, s_1), (s_0, s_2), (s_1, s_2), (s_1, s_3), (s_1, s_4),\\
&\quad (s_2, s_2), (s_3, s_0), (s_4, s_4)\},\\
s^0 &= s_0,\\
\pi(s_0) &= \{p_1\},\\
\pi(s_1) &= \{p_1\},\\
\pi(s_2) &= \emptyset,\\
\pi(s_3) &= \{p_1, p_2\}, \text{ and}\\
\pi(s_4) &= \{p_2\}.
\end{aligned}$$

This Kripke structure can be depicted as the node-labelled directed graph shown in Fig. 2.1. The states $s \in S$ *correspond to the nodes of the graph, the transitions* $(s, s') \in \rho$ *correspond to the directed arcs between nodes, and the function* π *gives a label for each node of the graph.*

Two executions of M *are*

$$x_1 = \langle s_0, s_1, s_3, s_0, s_1, s_3, \dots \rangle \quad \text{and} \quad x_2 = \langle s_0, s_2, s_2, s_2, \dots \rangle.$$

They correspond to the infinite sequences of state labels

$$\begin{aligned}
\xi_{x_1} &= \langle \pi(s_0), \pi(s_1), \pi(s_3), \pi(s_0), \pi(s_1), \pi(s_3), \dots \rangle\\
&= \langle \{p_1\}, \{p_1\}, \{p_1, p_2\}, \{p_1\}, \{p_1\}, \{p_1, p_2\}, \dots \rangle
\end{aligned}$$

and

$$\begin{aligned}
\xi_{x_2} &= \langle \pi(s_0), \pi(s_2), \pi(s_2), \pi(s_2), \dots \rangle\\
&= \langle \{p_1\}, \emptyset, \emptyset, \emptyset, \dots \rangle,
\end{aligned}$$

respectively. These sequences of state labels also belong to the language $\mathcal{L}_M$ *generated by the structure.*

The path $\langle s_0, s_1, s_2 \rangle$ *is not an execution, because it is finite. The infinite path* $\langle s_1, s_3, s_0, s_1, s_4, s_4, s_4, \dots \rangle$ *is not an execution either, since it does not begin in the initial state* s_0.

The infinite sequence $\xi = \langle \{p_2\}, \{p_1\}, \{p_2\}, \{p_1\}, \{p_2\}, \{p_1\}, \dots \rangle$ *does not belong to the language* $\mathcal{L}_M$*, since* M *has no execution corresponding to* ξ. ■

3 LINEAR TEMPORAL LOGIC

This work concentrates on testing model checking algorithms used in the verification of *propositional linear temporal logic*. This logic, introduced by Pnueli [20], is an extension of ordinary propositional logic with *temporal operators*, and it can be used to express qualitative temporal properties of individual executions of a Kripke structure. This chapter presents the formal definition and semantics of LTL. The semantics of LTL can also be given an interpretation on Kripke structures, which provides a way to state a temporal requirement concerning *all* executions of the structure. This extension then leads to the model checking problem for LTL, which will be discussed in Chap. 4.

The set of linear temporal logic formulae is defined inductively as follows. As before, AP denotes a finite nonempty set of atomic propositions.

Definition 2 (Linear temporal logic) *The set of linear temporal logic formulae consists of the finite-length strings that can be obtained by the application of the following rules:*

- *All atomic propositions* $p \in AP$ *are LTL formulae.*
- *If* φ *is an LTL formula, then* $\neg\varphi$ *is an LTL formula.*
- *If* φ *and* ψ *are LTL formulae, then* $(\varphi \lor \psi)$ *is an LTL formula.*
- *If* φ *is an LTL formula, then* $\mathsf{X}\,\varphi$ *is an LTL formula.*
- *If* φ *and* ψ *are LTL formulae, then* $(\varphi\,\mathsf{U}\,\psi)$ *is an LTL formula.*

■

The semantics of linear temporal logic formulae are defined over infinite sequences of subsets of AP as follows.

Definition 3 (Semantics of LTL) *Let* $\xi = \langle y_0, y_1, y_2, \dots\rangle \in (2^{AP})^\omega$ *be an infinite sequence of subsets of AP, and let* φ *be a linear temporal logic formula. Let* ξ^i *denote the infinite subsequence of* ξ *beginning at the* i^{th} *successor of* y_0 *in the sequence. That is,* $\xi^0 = \xi = \langle y_0, y_1, y_2, \dots\rangle$, $\xi^1 = \langle y_1, y_2, \dots\rangle$, $\xi^2 = \langle y_2, \dots\rangle$, *and so forth.*

We use the notation $\xi \models \varphi$ *to say that the sequence* ξ satisfies *(or alternatively, is a* model of*) the formula* φ, *and the notation* $\xi \not\models \varphi$ *is used to say that* ξ *does* not *satisfy* φ. *The relation* $\models$ *between the infinite sequences over subsets of* AP *and LTL formulae is given by the following conditions:*

- $\xi \models p$ *iff* $p \in y_0$, *the first element of the sequence* ξ.
- $\xi \models \neg\varphi$ *iff* $\xi \not\models \varphi$.
- $\xi \models (\varphi \lor \psi)$ *iff* $\xi \models \varphi$ or $\xi \models \psi$.
- $\xi \models \mathsf{X}\,\varphi$ *iff* $\xi^1 \models \varphi$.
- $\xi \models (\varphi\,\mathsf{U}\,\psi)$ *iff there exists* $i \geq 0$ *such that* $\xi^i \models \psi$, *and for all* $0 \leq j < i$, $\xi^j \models \varphi$.

If φ *has no models in* $(2^{AP})^\omega$, *we say that* φ *is an* unsatisfiable *formula. Conversely, if* $\neg\varphi$ *has no models in* $(2^{AP})^\omega$, φ *is called a* valid *LTL formula.*

■

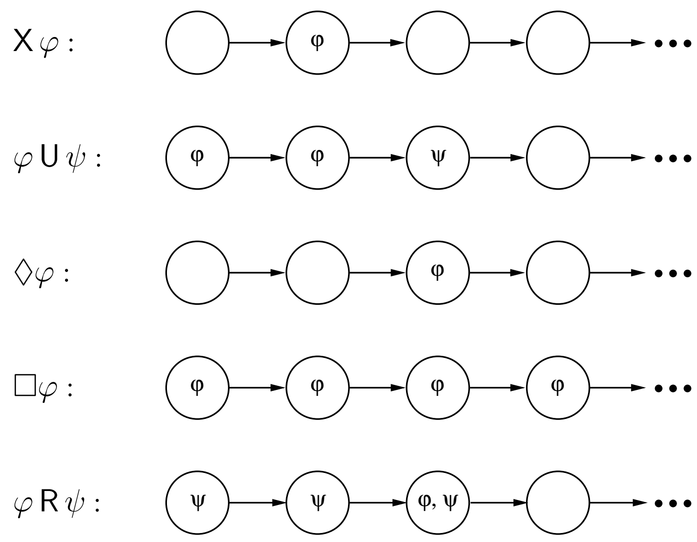


Fig. 3.1: Illustration of the semantics of the temporal operators X, U, $\Diamond$, $\Box$ and R

Other logical connectives and Boolean constants can be defined as abbreviations in the usual way: $\top \stackrel{def}{\equiv} (p \vee \neg p)$ for an arbitrary atomic proposition $p \in AP$ (Boolean constant "true"), $\bot \stackrel{def}{\equiv} \neg\top$ (Boolean constant "false"), $(\varphi \wedge \psi) \stackrel{def}{\equiv} \neg(\neg\varphi \vee \neg\psi)$ (conjunction), $(\varphi \rightarrow \psi) \stackrel{def}{\equiv} (\neg\varphi \vee \psi)$ (implication), and $(\varphi \leftrightarrow \psi) \stackrel{def}{\equiv} \big((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)\big)$ (equivalence).

Definition 3 implies that the satisfiability of an LTL formula in an infinite sequence $\xi \in (2^{AP})^\omega$ depends only on the first subset of AP in the sequence, if the formula does not contain any X or U operators. However, the satisfiability of formulae containing these *temporal operators* may depend on some other part of the sequence or even the entire sequence.

Intuitively, the subsequences ξ^i of a sequence ξ represent "the state of the world" at discrete consecutive time steps corresponding to the increasing values of the index i. Therefore, the sequence ξ^i can be thought of following ξ^0 "i steps later" in the future. This analogy can help to understand the temporal interpretation of the X and U operators. In the following, we shall give an informal description of the semantics of these operators, together with the definitions of a few commonly used other temporal operators that can be defined using these basic operators. See also Fig. 3.1 for an illustration.

The temporal formula $\mathsf{X}\,\varphi$ is satisfied in an infinite sequence ξ if the formula φ holds in the infinite subsequence ξ^1 of ξ, i.e., "at the next time step in the future". The X operator is called the "Next time" operator.

The formula $(\varphi\,\mathsf{U}\,\psi)$ holds in an infinite sequence ξ if and only if ψ holds "now or some time in the future" (i.e., in some infinite subsequence ξ^i of ξ, where $i \geq 0$), and φ holds "until" ψ becomes true (i.e., in all infinite subsequences of the original sequence beginning at some nonnegative index less than i). Therefore, U is called the "Until" operator. However, φ needs not hold in any subsequence of ξ if ψ already holds in ξ^0 (i.e., in the whole sequence ξ).

New operators can again be defined in terms of the U operator: two commonly used operators are $\Diamond\varphi \stackrel{def}{\equiv} (\top\,\mathsf{U}\,\varphi)$, expressing that φ *eventually* holds in a sequence, and $\Box\varphi \stackrel{def}{\equiv} \neg\Diamond\neg\varphi$, which is used to say that φ *always* holds in a sequence. We will also use the dual of the U operator R called the "Release" operator, which is defined by $(\varphi \,\mathsf{R}\, \psi) \stackrel{def}{\equiv} \neg(\neg\varphi \,\mathsf{U}\, \neg\psi)$. Intuitively, the formula $(\varphi \,\mathsf{R}\, \psi)$ is true in an infinite sequence ξ if and only if ψ either holds "forever" in the sequence (i.e., in all infinite subsequences of ξ), or if both φ and ψ hold "at the same time now or somewhere in the future" (i.e., in some infinite subsequence ξ^i of ξ with $i \geq 0$), and ψ holds also in all "earlier" subsequences (i.e., in all subsequences ξ^j with $0 \leq j < i$). In this case, φ "releases" ψ in the sequence, so ψ need not remain true any longer after the first true occurrence of φ.

Example 2 *Let $AP = \{p_1, p_2\}$, and let $\xi = \langle y_0, y_1, y_2, \dots\rangle = \langle \{p_1\}, \{p_1\}, \{p_1, p_2\}, \{p_1\}, \{p_1\}, \{p_1, p_2\}, \dots\rangle$ be an infinite sequence over 2^{AP}. We show that this sequence satisfies the LTL formula $\Box\Diamond p_2$, by the direct application of the semantics of LTL. This formula corresponds to the property "p_2 is always eventually true in the sequence", or, in other words, "p_2 is true in the sequence* infinitely often*".*

The formula is first rewritten using the basic temporal operators:

$$\begin{array}{ll}
\xi \models \Box\Diamond p_2 & \textit{iff} \\
\xi \models \Box(\top \,\mathsf{U}\, p_2) & \textit{iff} \\
\xi \models \neg\Diamond\neg(\top \,\mathsf{U}\, p_2) & \textit{iff} \\
\xi \models \neg\big(\top \,\mathsf{U}\, \neg(\top \,\mathsf{U}\, p_2)\big) & \textit{iff} \\
\xi \not\models \top \,\mathsf{U}\, \neg(\top \,\mathsf{U}\, p_2). &
\end{array}$$

By the semantics of LTL, $\xi \models \top \,\mathsf{U}\, \neg(\top \,\mathsf{U}\, p_2)$ if there exists an $i \geq 0$ such that $\xi^i \models \neg(\top \,\mathsf{U}\, p_2)$, and for all $0 \leq j < i$, $\xi^j \models \top$. Therefore, $\xi \not\models \top \,\mathsf{U}\, \neg(\top \,\mathsf{U}\, p_2)$ holds only if this is not *the case. This can occur in two ways:*

(1) There is no $i \geq 0$ such that $\xi^i \models \neg(\top \,\mathsf{U}\, p_2)$ is true, that is, $\xi^i \models \top \,\mathsf{U}\, p_2$ holds for all $i \geq 0$.

(2) For all $i \geq 0$ such that $\xi^i \models \neg(\top \,\mathsf{U}\, p_2)$ there exists a $0 \leq j < i$ for which $\xi^j \not\models \top$.

In fact, the case (1) holds in the given sequence. First of all, we note that for all $i \geq 0$, $\xi^i \models \top$:

$$\begin{array}{llll}
\xi^i \models \top & & & \textit{iff} \\
\xi^i \models p_1 \vee \neg p_1 & & & \textit{iff} \\
\xi^i \models p_1 & \text{or} & \xi^i \models \neg p_1 & \textit{iff} \\
\xi^i \models p_1 & \textit{or} & \xi^i \not\models p_1, &
\end{array}$$

which is trivially true, since either $p_1 \in y_i$ or $p_1 \notin y_i$ for all subsets $y_i \in 2^{AP}$ and $i \geq 0$.

We then show that for all $i \geq 0$, $\xi^i \models \top \,\mathsf{U}\, p_2$. We note that for all $k \geq 0$, $\xi^{k+3} = \xi^k$ in the given sequence. By definition, $\xi^i \models \top \,\mathsf{U}\, p_2$ if and only if there exists an $i' \geq i$ such that $\xi^{i'} \models p_2$ and for all $i \leq j < i'$, $\xi^j \models \top$. It has already been shown that $\xi^j \models \top$ for all $j \geq 0$. We also know that $\xi^2 = \langle \{p_1, p_2\}, \{p_1\}, \{p_1\}, \{p_1, p_2\}, \{p_1\}, \{p_1\}, \dots\rangle \models p_2$. Because $\xi^{k+3} = \xi^k$ for all $k \geq 0$, it follows that $\xi^{2+3k} \models p_2$ for all $k \geq 0$, and therefore for all $i \geq 0$ there must exist an $i' \geq i$ such that $\xi^{i'} \models p_2$, so $\xi \models \Box\Diamond p_2$. ■

Let $M = \langle S, \rho, s^0, \pi \rangle$ be a Kripke structure. Since the labelling function π maps every path in M into an infinite sequence of subsets of AP, π gives the *temporal interpretation* of any path in the structure. More precisely, the semantics of LTL are interpreted on Kripke structures as follows:

Definition 4 (LTL semantics in Kripke structures) *Let $x = \langle s_0, s_1, s_2, \ldots \rangle \in S^\omega$ be an infinite path in a Kripke structure M, and let φ be an LTL formula. We say that the path x* satisfies *φ, denoted $x \models \varphi$, if and only if the infinite sequence $\xi_x = \langle \pi(s_0), \pi(s_1), \pi(s_2), \ldots \rangle$ satisfies the formula φ.*

We say that the Kripke structure M satisfies the LTL formula φ if and only if all paths $x \in \{\langle s_0, s_1, s_2, \ldots \rangle \in S^\omega \mid s_0 = s^0$ and $(s_i, s_{i+1}) \in \rho$ for all $i \geq 0\}$ (i.e., all executions of M) satisfy φ. We denote this by $M \models \varphi$. ■

The latter part of this definition considers only the *executions* of the structure and therefore does not require anything of the paths that do not begin in the initial state of M. Therefore, even if $M \models \varphi$ is true, M may still contain paths x for which $x \models \varphi$ does not hold. We nevertheless use phrases like "the formula φ holds in M" to mean that φ holds in all *executions* of M. This should not give rise to any confusion, since we are usually not interested in paths that are not executions.

The semantics of LTL imply that an execution x of a Kripke structure M satisfies an LTL formula φ if and only if it does not satisfy its negation $\neg\varphi$. However, this "symmetry" does not apply to the satisfiability of the formula in the whole structure M. Since $M \models \varphi$ holds if and only if *all* behaviours of M satisfy φ, even a single execution satisfying $\neg\varphi$ (i.e., a *counter-example* for φ) is sufficient to show that $M \not\models \varphi$ is true. However, this does not generally imply that $M \models \neg\varphi$ would then hold, since this is again a statement over *all* executions of the structure. Therefore, although it is not possible that $M \models \varphi$ and $M \models \neg\varphi$ hold at the same time, it may be that *neither* of these properties holds in the structure. This occurs if M has several paths beginning in its initial state, some of which satisfy the property φ, while others satisfy the negated property $\neg\varphi$.

The following example demonstrates interpreting the semantics of LTL on the executions of a Kripke structure.

Example 3 *The sequence ξ of the previous example corresponds to the execution $x_1 = \langle s_0, s_1, s_3, s_0, s_1, s_3, \ldots \rangle$ of the Kripke structure M given in Example 1 (see also Fig. 2.1). We showed in the previous example that the LTL formula $\Box\Diamond p_2$ holds in this execution.*

However, $M \not\models \Box\Diamond p_2$, because M also has the execution $x_2 = \langle s_0, s_2, s_2, s_2, \ldots \rangle$, and the formula does not hold in the sequence $\xi_{x_2} = \langle \pi(s_0), \pi(s_2), \pi(s_2), \pi(s_2), \ldots \rangle = \langle \{p_1\}, \emptyset, \emptyset, \emptyset \ldots \rangle$. This is because p_2 is never true in this execution, which can again be shown using the semantics of LTL as in the previous example.

(As a matter of fact, the executions x_1 and x_2 together show that $M \models \neg\Box\Diamond p_2$ does not hold either, since $x_1 \not\models \neg\Box\Diamond p_2$, but $x_2 \models \neg\Box\Diamond p_2$.) ■

4 AUTOMATA-THEORETIC LTL MODEL CHECKING

This chapter introduces the model checking problem for linear temporal logic and reviews its automata-theoretic solution, which creates the need for translating LTL formulae into Büchi automata. Since the model checking procedure for LTL forms a significant basis for the testing techniques for LTL-to-Büchi translation algorithm implementations (to be described in Chap. 5), this chapter includes a fairly detailed description of the general model checking procedure.

4.1 THE LTL MODEL CHECKING PROBLEM

In short, LTL model checking tells whether all behaviours of a given system model satisfy a given LTL property. For example, one might be interested in confirming that the system will always return to some "safe" state after performing some operation, regardless of the outcome of the operation. (For example, a data communications protocol could be checked for the property that it will always recover from lost messages, assuming that no message can be lost infinitely many times.) If the system is found to have an execution violating the desired property, the system has an error and needs to be modified in order to prevent the occurrence of the undesired behaviour.

Definition 4 of the previous chapter gives a way to interpret the semantics of LTL on the executions of Kripke structures. The model checking problem for linear temporal logic can then be stated as follows.

Problem 1 (The LTL model checking problem) *Given a Kripke structure M and a linear temporal logic formula φ, does $M \models \varphi$ hold?*

In the LTL model checking problem, linear temporal logic formulae express requirements concerning all executions of a Kripke structure. Alternatively, since a single counter-example is sufficient for proving an LTL property false in the set of executions of the structure, the problem can be solved by checking whether the structure has an execution satisfying the *negation* of the same property. By the semantics of LTL, the nonexistence of such an execution implies that the property itself is true in all executions. However, the very naive approach of checking each execution of the structure in turn for the satisfiability (or unsatisfiability) of some property (e.g., by the direct application of the semantics of LTL) is not generally feasible, since the number of executions contained in the structure may be infinite.

To find more practical methods for solving the model checking problem for LTL, it is useful to rephrase the problem as a question about the relationship between languages [31]. As mentioned already in Chap. 2, the Kripke structure M can be seen as generating a language $\mathcal{L}_M$ that consists of the infinite words over state labels (chosen from the set 2^{AP}) corresponding to the executions of the model. Since also the models of an LTL formula φ are infinite sequences of subsets of AP, the set of all models of the formula can actually be considered another language $\mathcal{L}_\varphi$ of infinite words over 2^{AP}. Therefore, the model checking problem can be stated as the question whether the language $\mathcal{L}_M$ generated by the executions of the system model is contained in the language $\mathcal{L}_\varphi$ corresponding to the models of the LTL formula, that is, whether $\mathcal{L}_M \subseteq \mathcal{L}_\varphi$. Since a single system execution satisfies an LTL formula φ if and only if it does not satisfy its negation $\neg\varphi$, the formula φ is satisfied in all executions of the system if and only if M has no execution satisfying $\neg\varphi$. Therefore, the problem reduces to the question whether *no* word in $\mathcal{L}_M$ belongs to the language $\mathcal{L}_{\neg\varphi}$ corresponding to the negation of the LTL formula. Finding an answer to this question amounts to checking whether the intersection $\mathcal{L}_M \cap \mathcal{L}_{\neg\varphi}$ of the languages $\mathcal{L}_M$ and $\mathcal{L}_{\neg\varphi}$ is empty.

In general, however, the model checking problem for linear temporal logic is known to be PSPACE-complete in the size of the formula [2], which inevitably limits the practical applicability of model checking as one of the reasons behind the state explosion problem. However, current computer technology has made LTL model checking possible even in real-world problems, and the benefits of model checking in uncovering errors in system specifications justify the need for solving this complex problem.

4.2 AUTOMATA-THEORETIC APPROACH TO LTL MODEL CHECKING

The study of formal languages is closely connected with the theory of *automata*. Analogously to the view of Kripke structures as models of systems, automata can be considered "models" of languages, and their properties can be used for proving properties of the languages corresponding to the automata. Since the LTL model checking problem can be stated as a question about the relationship between two languages, the problem can be solved by using automata-theoretic techniques. This general approach to model checking is due to Vardi and Wolper [31]; its specific application to linear temporal logic is discussed in [30].

4.2.1 Büchi Automata

The connection between LTL model checking and automata theory arises from the fact that the language $\mathcal{L}_\varphi$ consisting of the models of a linear temporal logic formula φ can be represented as a nondeterministic finite automaton over infinite words—a finite state-transition system, whose behaviours generate all the models of the formula. These state-transition systems are traditionally called *Büchi automata*. (More formally, any language corresponding to the set of models of some LTL formula belongs to the class of *ω-regular* languages, and each such language is recognizable by a nondeterministic Büchi automaton. See e.g. [28].)

Instead of thinking of a Büchi automaton as generating all models of an LTL formula, the automaton can intuitively be seen as a "machine" with the ability to tell from any infinite word over the alphabet 2^{AP} whether it belongs to the language $\mathcal{L}_\varphi$ corresponding to the models of the formula. Therefore, Büchi automata can be used to test the behaviours of a system for the satisfiability of linear temporal logic properties.

We use the following definition for Büchi automata.

Definition 5 (Büchi automata) *A Büchi automaton is a 5-tuple* $A = \langle \Sigma, Q, \Delta, q^0, \mathcal{F} \rangle$*, where*

- Σ *is a finite* alphabet*,*
- Q *is a finite set of* states*,*
- $\Delta \subseteq Q \times 2^{\Sigma} \times Q$ *is the* transition relation[1]*,*
- $q^0 \in Q$ *is the* initial state*, and*
- $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ *is a finite set of* acceptance conditions*, where* $F_i \subseteq Q$ *for all* $1 \leq i \leq n$*.*

An execution *of* A *over an infinite word* $\xi = \langle y_0, y_1, y_2, \ldots \rangle \in \Sigma^{\omega}$ *is an infinite sequence of states* $\langle q_0, q_1, q_2, \ldots \rangle \in Q^{\omega}$ *such that* $q_0 = q^0$*, and for all* $i \geq 0$*,* $(q_i, \sigma_i, q_{i+1}) \in \Delta$ *for some* $\sigma_i \subseteq \Sigma$ *such that* $y_i \in \sigma_i$*.*

Let $r = \langle q_0, q_1, q_2, \ldots \rangle \in Q^{\omega}$ *be an execution of* A *over an infinite word* $\xi \in \Sigma^{\omega}$*. Let* $\inf(r) \subseteq Q$ *be the set of states occurring infinitely many times in* r*. We say that the execution* fulfils *the acceptance condition* $F_i \in \mathcal{F}$ *if and only if* $\inf(r) \cap F_i \neq \emptyset$*. If this holds for all acceptance conditions* $F_i \in \mathcal{F}$*, we say that* r *is an* accepting *execution of* A *over* ξ*.*

The automaton accepts *an infinite word* $\xi \in \Sigma^{\omega}$ *if and only if it has an accepting execution over* ξ*. If the automaton has no accepting executions over* ξ*, it* rejects ξ*.* ■

Büchi automata $A = \langle \Sigma, Q, \Delta, q^0, \mathcal{F} \rangle$ with $|\mathcal{F}| \neq 1$ are sometimes called *generalized* Büchi automata to distinguish them from automata with only one acceptance condition. It can be shown (see e.g. [11]) that all Büchi automata with any nonzero number of acceptance conditions are equally expressive, so such a distinction is not used here.

The language $\mathcal{L}_A$ accepted by the Büchi automaton A consists of the set of infinite words over Σ accepted by the automaton A. If the automaton represents a linear temporal logic formula φ, we will refer to the automaton as A_{φ} and to the language accepted (or *recognized*) by the automaton as $\mathcal{L}_{A_{\varphi}}$.

Büchi automata can be seen as edge-labelled directed graphs. The nodes of the graph are the elements of the set Q, and the arcs between the graph nodes are given by the transition relation such that there is an arc from state $q \in Q$ to another state $q' \in Q$ if and only if $(q, \sigma, q') \in \Delta$ for some $\sigma \subseteq \Sigma$. The arc label σ is a *set* of alphabet symbols, each of which can cause the automaton to move from the state q to the state q'.

Definition 5 allows a Büchi automaton to have several arcs beginning in a state such that the labels of these arcs are not disjoint (i.e., they contain a common symbol $a \in \Sigma$). Therefore, the automaton can have many executions on a given word. It is sufficient that *any* of these executions is accepting for the automaton to accept the word. This nondeterminism is actually an

[1] The "labels" associated with the transitions are defined over 2^{Σ} instead of Σ for convenience. This makes it possible to combine all alphabet symbols on which the automaton can move from a state to another state into the same transition; see the definition of the executions of a Büchi automaton.

essential requirement for Büchi automata to be able to express all linear temporal properties: it can be shown that deterministic Büchi automata (with at most one execution on any input word) are not as expressive as nondeterministic Büchi automata (see e.g. [30]).

We will not discuss here how to actually obtain a Büchi automaton from a linear temporal logic formula. This phase in LTL model checking is a somewhat nontrivial task in itself and may even be difficult to handle correctly and efficiently in practice, which is suggested by the experiments made in this and earlier work [26, 27] with practical implementations. Even the theoretical question of LTL formula translation into Büchi automata has still gained research interest with new and improved translation algorithms aimed at efficient minimization of the number of states and transitions in the constructed automata presented year after year. The early algorithmic techniques for LTL-to-Büchi translation [31] were related to *tableau methods* for LTL (e.g. [35, 16, 12]). Most of the recent algorithms focus on the direct construction of automata. These algorithms try to use the syntactic structure of the LTL formula efficiently to guide the automaton construction in order to minimize the size of the result. This basic approach was presented in [8], and further improvements have been proposed later both inside and around the basic conversion phase [5, 24, 6].

Not all languages corresponding to the models of LTL formulae have concise representations as Büchi automata. The translation of a linear temporal logic formula φ into a Büchi automaton may in the worst case require an automaton with $2^{\mathcal{O}(|\varphi|)}$ states, where $|\varphi|$ denotes the length of φ [32].

Example 4 *As an example of a Büchi automaton representing a language defined by an LTL formula, we give a Büchi automaton for the formula* $\Box\Diamond p_2$ *from Example 2. Let* $AP = \{p_1, p_2\}$*, and let* $A_{\Box\Diamond p_2} = \langle \Sigma, Q, \Delta, q^0, \mathcal{F} \rangle$ *be the Büchi automaton, where*

$$\begin{aligned}
\Sigma &= 2^{AP} = \big\{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\big\}, \\
Q &= \{q_0, q_1, q_2\}, \\
\Delta &= \Big\{\big(q_0, \big\{\{p_2\}, \{p_1, p_2\}\big\}, q_1\big), \big(q_0, \big\{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\big\}, q_2\big), \\
&\qquad \big(q_1, \big\{\{p_2\}, \{p_1, p_2\}\big\}, q_1\big), \big(q_1, \big\{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\big\}, q_2\big), \\
&\qquad \big(q_2, \big\{\{p_2\}, \{p_1, p_2\}\big\}, q_1\big), \big(q_2, \big\{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\big\}, q_2\big)\Big\}, \\
q^0 &= q_0, \text{ and} \\
\mathcal{F} &= \big\{\{q_0, q_1\}\big\}.
\end{aligned}$$

The automaton is shown in Fig. 4.1. The states associated with the only acceptance condition of the automaton are marked with a double circle.

Clearly, no execution of the automaton can visit the state q_0 *infinitely often, since the automaton has no transitions with* q_0 *as the target state. Therefore, all accepting executions of the automaton must visit the state* q_1 *infinitely often. This can happen only if the automaton executes an infinite number of transitions with the label* $\big\{\{p_2\}, \{p_1, p_2\}\big\}$*, so any word accepted by the automaton must contain an infinite number of symbols* $\{p_2\}$ *or* $\{p_1, p_2\}$*. From this it follows that the word must have an infinite number of suffixes beginning with either of these symbols, and by the semantics of LTL,* p_2 *holds in any such suffix. Therefore, the automaton accepts an*

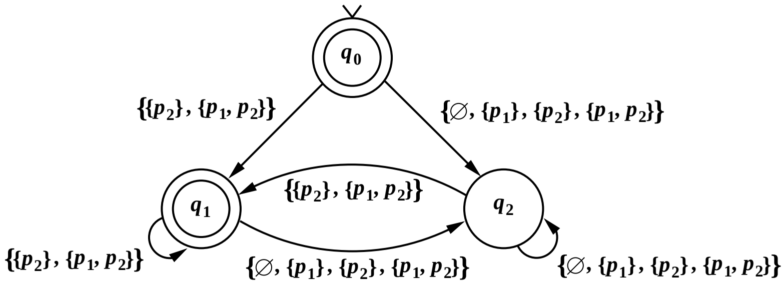


Fig. 4.1: A Büchi automaton for the LTL formula $\Box\Diamond p_2$

infinite sequence over 2^{AP} only if p_2 holds infinitely often in the sequence, which corresponds to the LTL property $\Box\Diamond p_2$.

Conversely, given any infinite sequence over 2^{AP} having the property $\Box\Diamond p_2$, the automaton can first move from the state q_0 to the state q_2, and then loop between q_2 and q_1 indefinitely by moving from q_2 to q_1 whenever "reading" either of the symbols $\{p_2\}$ or $\{p_1, p_2\}$. The automaton can then remain in q_1 until it "reads" a symbol other than $\{p_2\}$ or $\{p_1, p_2\}$, which forces it to return to state q_2. The fact that the input sequence satisfies the LTL property $\Box\Diamond p_2$ guarantees that the automaton will visit the state q_1 infinitely often, so the input is accepted.

Therefore, the automaton accepts an infinite sequence over 2^{AP} if and only if the sequence satisfies the LTL property $\Box\Diamond p_2$. ■

4.2.2 Kripke Structures as Büchi Automata

In the end of Chap. 2, the executions of a Kripke structure were identified with a language $\mathcal{L}_M$ of infinite words over 2^{AP}. Since also Büchi automata are representations for languages, any Kripke structure can further be identified with a Büchi automaton that accepts the language $\mathcal{L}_M$. Informally, a given Kripke structure can be transformed into an equivalent Büchi automaton over the alphabet $\Sigma = 2^{AP}$ by simply copying the label of each state of the Kripke structure onto every arc leaving the state. In addition, all executions of the automaton are trivially accepting, and therefore no explicit acceptance conditions are required. (This is equivalent to having one acceptance condition including all states of the automaton.)

More precisely, we have the following lemma.

Lemma 1 *Let $M = \langle S, \rho, s^0, \pi\rangle$ be a Kripke structure. Define the Büchi automaton $A_M = \langle \Sigma, Q, \Delta, q^0, \mathcal{F}\rangle$, where*

$$\begin{aligned}
\Sigma &= 2^{AP}, \\
Q &= S, \\
\Delta &= \big\{(s, \sigma, s') \in Q \times 2^{\Sigma} \times Q \mid (s, s') \in \rho,\ \text{and } \sigma = \{\pi(s)\}\big\}, \\
q^0 &= s^0, \text{ and} \\
\mathcal{F} &= \emptyset.
\end{aligned}$$

Let $\xi = \langle y_0, y_1, y_2, \ldots\rangle \in (2^{AP})^\omega$ be an infinite word over subsets of AP.

The automaton A_M accepts the word ξ if and only if the Kripke structure M has an execution $x = \langle s_0, s_1, s_2, \ldots\rangle \in S^\omega$ such that $y_i = \pi(s_i)$ for all $i \geq 0$.

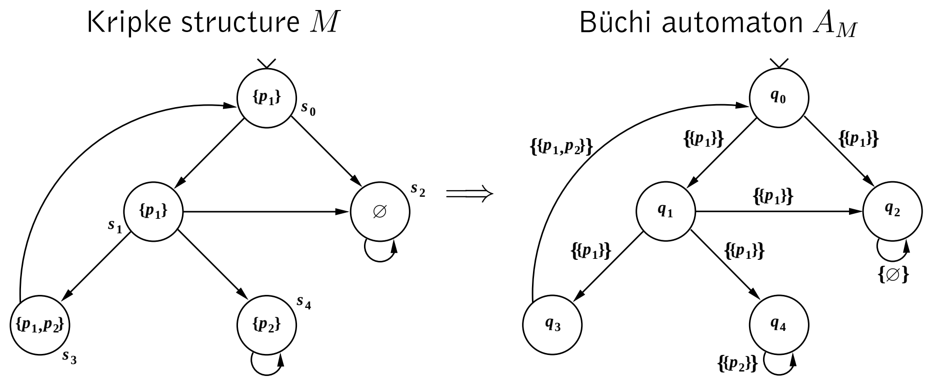


Fig. 4.2: Converting a Kripke structure into a Büchi automaton

Proof: ($\Rightarrow$) Assume that M has an execution $x = \langle s_0, s_1, s_2, \dots \rangle \in S^\omega$. Thus, $s_0 = s^0$, and for all $i \geq 0$, $(s_i, s_{i+1}) \in \rho$. This execution corresponds to the word $\xi = \langle y_0, y_1, y_2, \dots \rangle \in (2^{AP})^\omega$, where $y_i = \pi(s_i)$ for all $i \geq 0$. By the definition of A_M, $q^0 = s^0$, and for all $i \geq 0$, $\big(s_i, \{y_i\}, s_{i+1}\big) \in \Delta$. Therefore, x is an execution of A_M over ξ. Because $\forall F \in \mathcal{F} : \inf(r) \cap F \neq \emptyset$ holds trivially (since $\mathcal{F}$ is empty), x is an accepting execution, and A_M accepts the word ξ.

($\Leftarrow$) Conversely, assume that A_M accepts the word $\xi = \langle y_0, y_1, y_2, \dots \rangle \in (2^{AP})^\omega$. Therefore, it has an execution $r = \langle q_0, q_1, q_2, \dots \rangle \in Q^\omega$ on ξ, where $q_0 = q^0$, and for all $i \geq 0$, $(q_i, \sigma_i, q_{i+1}) \in \Delta$ for some $\sigma_i \subseteq \Sigma$ such that $y_i \in \sigma_i$. By definition of A_M, this can be the case only if for all $i \geq 0$, $(q_i, q_{i+1}) \in \rho$, and $\sigma_i = \{\pi(q_i)\}$. Because $y_i \in \sigma_i$, it follows that $y_i = \pi(q_i) = \pi(s_i)$, and since $q_0 = q^0 = s^0$, it follows that r is an execution of M. $\square$

Example 5 *Using the construction in the above lemma, we can construct an equivalent Büchi automaton A_M for the Kripke structure $M = \langle S, \rho, s^0, \pi \rangle$ defined in Example 1. Let $AP = \{p_1, p_2\}$ as in the previous examples, and let $A_M = \langle \Sigma, Q, \Delta, q^0, \mathcal{F} \rangle$, where*

$$\begin{aligned}
\Sigma &= 2^{AP} = \big\{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\big\}, \\
Q &= \{q_0, q_1, q_2, q_3, q_4\}, \\
\Delta &= \Big\{\big(q_0, \{\{p_1\}\}, q_1\big), \big(q_0, \{\{p_1\}\}, q_2\big), \big(q_1, \{\{p_1\}\}, q_2\big), \\
&\quad \big(q_1, \{\{p_1\}\}, q_3\big), \big(q_1, \{\{p_1\}\}, q_4\big), \big(q_2, \{\emptyset\}, q_2\big), \\
&\quad \big(q_3, \{\{p_1, p_2\}\}, q_0\big), \big(q_4, \{\{p_2\}\}, q_4\big)\Big\}, \\
q^0 &= q_0, \text{ and} \\
\mathcal{F} &= \emptyset.
\end{aligned}$$

By Lemma 1, this automaton accepts an infinite sequence over 2^{AP} if and only if the sequence is the temporal interpretation of an execution of M. Figure 4.2 illustrates the conversion. ■

4.2.3 Synchronous Product

Any two Büchi automata A_1 and A_2 corresponding to two languages $\mathcal{L}_{A_1}$ and $\mathcal{L}_{A_2}$ can be combined together into another Büchi automaton that accepts precisely the language $\mathcal{L}_{A_1} \cap \mathcal{L}_{A_2}$ (see e.g. [32]). This composition is

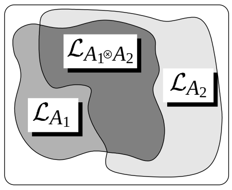


Fig. 4.3: The languages $\mathcal{L}_{A_1}$, $\mathcal{L}_{A_2}$ and $\mathcal{L}_{A_1 \otimes A_2}$

called the *synchronous product* of A_1 and A_2 (denoted in the following by $A_1 \otimes A_2$). Since the language accepted by the product automaton corresponds to the intersection of the languages accepted by the automata from which it was constructed, the product automaton can also be simply called the *intersection* of two Büchi automata. See Fig. 4.3 for an illustration.

The construction of the synchronous product, together with the proof that it has the required properties, are given in the following lemma.

Lemma 2 *Let $A_1 = \langle \Sigma_1, Q_1, \Delta_1, q_1^0, \mathcal{F}_1 \rangle$ and $A_2 = \langle \Sigma_2, Q_2, \Delta_2, q_2^0, \mathcal{F}_2 \rangle$ be two Büchi automata, where Q_1 and Q_2 are disjoint, and $\mathcal{F}_1 = \{F_1^1, F_2^1, \ldots, F_n^1\}$ and $\mathcal{F}_2 = \{F_1^2, F_2^2, \ldots, F_m^2\}$ for some $n, m \geq 0$. Define the Büchi automaton $A = \langle \Sigma, Q, \Delta, q^0, \mathcal{F} \rangle$, where*

$$
\begin{aligned}
\Sigma &= \Sigma_1 \cup \Sigma_2, \\
Q &= Q_1 \times Q_2, \\
\Delta &= \Big\{ \big((q_1, q_2), \sigma, (q_1', q_2')\big) \in Q \times 2^{\Sigma} \times Q \mid \\
&\qquad (q_1, \sigma_1, q_1') \in \Delta_1, (q_2, \sigma_2, q_2') \in \Delta_2, \text{ and } \sigma = \sigma_1 \cap \sigma_2 \neq \emptyset \Big\}, \\
q^0 &= (q_1^0, q_2^0), \text{ and} \\
\mathcal{F} &= \{F_1^1 \times Q_2, F_2^1 \times Q_2, \ldots, F_n^1 \times Q_2, \\
&\qquad Q_1 \times F_1^2, Q_1 \times F_2^2, \ldots, Q_1 \times F_m^2\}.
\end{aligned}
$$

Assume that $\Sigma_1 \cap \Sigma_2 \neq \emptyset$, and let $\xi = \langle y_0, y_1, y_2, \ldots \rangle \in (\Sigma_1 \cap \Sigma_2)^\omega$ be an infinite word over $\Sigma_1 \cap \Sigma_2$.

The automaton A accepts the word ξ if and only if both A_1 and A_2 accept ξ. Moreover, A will not accept any word in $(\Sigma_1 \cup \Sigma_2)^\omega \setminus (\Sigma_1 \cap \Sigma_2)^\omega$.

Proof: ($\Rightarrow$) Assume that A_1 and A_2 both accept ξ. Then, A_1 and A_2 have executions $r_1 = \langle q_0^1, q_1^1, q_2^1, \ldots \rangle \in Q_1^\omega$ and $r_2 = \langle q_0^2, q_1^2, q_2^2, \ldots \rangle \in Q_2^\omega$, respectively, such that $q_0^1 = q_1^0$, $q_0^2 = q_2^0$, and for all $i \geq 0$ and $j \in \{1, 2\}$, $(q_i^j, \sigma_i^j, q_{i+1}^j) \in \Delta_j$ for some $\sigma_i^j \subseteq \Sigma_j$ such that $y_i \in \sigma_i^j$. By definition of A, $q^0 = (q_1^0, q_2^0) = (q_0^1, q_0^2)$, $\big((q_i^1, q_i^2), \sigma_i^1 \cap \sigma_i^2, (q_{i+1}^1, q_{i+1}^2)\big) \in \Delta$ for all $i \geq 0$, and $y_i \in \sigma_i^1 \cap \sigma_i^2$. Therefore, A has the execution $r = \langle q_0, q_1, q_2, \ldots \rangle \in (Q_1 \times Q_2)^\omega$ such that $q_i = (q_i^1, q_i^2) \in Q_1 \times Q_2$ for all $i \geq 0$.

Since r_1 is an accepting execution of A_1, there must for all acceptance conditions $F_i^1 \in \mathcal{F}_1$ ($1 \leq i \leq n$) exist a state $q_{k_i}^1 \in F_i^1$ that occurs infinitely often in r_1. Therefore, for each acceptance condition $F_i^1 \times Q_2 \in \mathcal{F}$ ($1 \leq i \leq n$), there are infinitely many indices $j \geq 0$ such that (in the execution r) $q_j \in F_i^1 \times Q_2$, so r intersects each of the acceptance conditions $F_i^1 \times Q_2 \in \mathcal{F}$ an infinite number of times. Because Q_2 is finite, there must for each acceptance condition exist a state $(q^1_{k_i}, q^2_{k_i}) \in F^1_i \times Q_2$ $(1 \leq i \leq n)$ that by itself occurs infinitely many times in the execution r, and therefore the acceptance condition $F^1_i \times Q_2$ is fulfilled in r for all $1 \leq i \leq n$.

A similar argument shows that also all acceptance conditions of the form $Q_1 \times F^2_j \in \mathcal{F}$ are fulfilled in the execution r. Therefore, r fulfils all acceptance conditions in $\mathcal{F}$. It follows that r is an accepting execution of A, so A accepts ξ.

($\Leftarrow$) Assume then that A accepts ξ. Therefore, it has an execution $r = \langle (q^1_0, q^2_0), (q^1_1, q^2_1), (q^1_2, q^2_2), \ldots \rangle \in (Q_1 \times Q_2)^\omega$, where $(q^1_0, q^2_0) = (q^0_1, q^0_2)$, and for all $i \geq 0$, $\big((q^1_i, q^2_i), \sigma_i, (q^1_{i+1}, q^2_{i+1})\big) \in \Delta$ for some $\sigma_i \subseteq \Sigma$ such that $y_i \in \sigma_i$. It follows directly from the definition of Δ that A_1 and A_2 have executions $r_1 = \langle q^1_0, q^1_1, q^1_2, \ldots \rangle \in Q^\omega_1$ and $r_2 = \langle q^2_0, q^2_1, q^2_2, \ldots \rangle \in Q^\omega_2$ on input ξ, respectively.

Because r is an accepting execution of A, there exists for each acceptance condition $F^1_i \times Q_2 \in \mathcal{F}$ $(1 \leq i \leq n)$ a state $(q^1_{k_i}, q^2_{k_i}) \in F^1_i \times Q_2$ that occurs infinitely often in r. Therefore, the state $q^1_{k_i}$ occurs in r_1 an infinite number of times. Similarly, we can find also for each acceptance condition $Q_1 \times F^2_j \in \mathcal{F}$ $(1 \leq j \leq m)$ a state $q^2_{k_j} \in F^2_j$ occurring infinitely often in r_2. Therefore, r_1 and r_2 are accepting executions of A_1 and A_2, respectively, so both automata accept ξ.

Finally, it is easy to see from the definition of A that A cannot even have an execution over a word $\xi \in (\Sigma_1 \cup \Sigma_2)^\omega \setminus (\Sigma_1 \cap \Sigma_2)^\omega$, and therefore it can neither accept any word in this set. Any word in this set would have to contain a symbol $a \in (\Sigma_1 \setminus \Sigma_2) \cup (\Sigma_2 \setminus \Sigma_1)$; however, for all transitions $(q, \sigma, q') \in \Delta$ of A, $\sigma \subseteq \Sigma_1 \cap \Sigma_2$, so $a \notin \sigma$ for all transitions of A. $\square$

Intuitively, the synchronous product of two Büchi automata captures all the "legal" *synchronous behaviours* that the original automata can have on any input word. Here, a "legal" synchronous behaviour corresponds to a parallel execution of the original automata such that at each step of the execution, the labels on the transitions chosen by the automata at that step share at least one common element.

Example 6 *We compute the synchronous product of the Büchi automaton A_M from Example 5 with the Büchi automaton $A_{\Box\Diamond p_2}$ from Example 4 (see the definitions for these automata from pages 15 and 13, respectively).*

Since A_M and $A_{\Box\Diamond p_2}$ both have the same alphabet 2^{AP}, their synchronous product $A = A_M \otimes A_{\Box\Diamond p_2} = \langle \Sigma, Q, \Delta, q^0, \mathcal{F} \rangle$ will have the same alphabet, so $\Sigma = 2^{AP} = \big\{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\big\}$.

Since A_M contains 5 states and $A_{\Box\Diamond p_2}$ contains 3 states, A has $5 \cdot 3 = 15$ states, so

$$\begin{aligned} Q \;=\; & \{(q_0, q'_0), (q_1, q'_0), (q_2, q'_0), (q_3, q'_0), (q_4, q'_0), \\ & (q_0, q'_1), \ldots, (q_4, q'_1), \\ & (q_0, q'_2), \ldots, (q_4, q'_2)\}, \end{aligned}$$

where the first element of each pair is a state of A_M and the second element is a state of $A_{\Box\Diamond p_2}$ (for clarity, primes are added to the states of $A_{\Box\Diamond p_2}$ to distinguish them from the states of A_M).

The initial state q^0 of A is (q_0, q'_0).

The transition relation Δ can be constructed as follows. Beginning in state (q_0, q_0'), we find the transitions starting from this state by checking whether any transition (q_0, σ, q) of A_M can be synchronized with any transition (q_0', σ', q') of $A_{\Box\Diamond p_2}$ (here, q and q' can be any states of A_M and $A_{\Box\Diamond p_2}$, respectively). The transitions of A_M are

$$\big(q_0, \{\{p_1\}\}, q_1\big) \quad \text{and} \quad \big(q_0, \{\{p_1\}\}, q_2\big),$$

and the transitions of $A_{\Box\Diamond p_2}$ are

$$\big(q_0', \{\{p_2\}, \{p_1, p_2\}\}, q_1'\big) \quad \text{and} \quad \big(q_0', \{\emptyset, \{p_1\}, \{p_2\}, \{p_1, p_2\}\}, q_2'\big).$$

By definition of the synchronous product, two transitions (one of which is always chosen from A_M, the other from $A_{\Box\Diamond p_2}$) can be synchronized if the set intersection of the labels associated with the transitions is nonempty. Because $\{\{p_1\}\} \cap \{\{p_2\}, \{p_1, p_2\}\} = \emptyset$, the transition $\big(q_0', \{\{p_2\}, \{p_1, p_2\}\}, q_1'\big)$ of $A_{\Box\Diamond p_2}$ cannot be synchronized with either of the transitions of A_M. However, the other transition can be synchronized with either of A_M's transitions, so Δ contains the two transitions

$$\Big((q_0, q_0'), \{\{p_1\}\}, (q_1, q_2')\Big) \quad \text{and} \quad \Big((q_0, q_0'), \{\{p_1\}\}, (q_2, q_2')\Big).$$

By repeating this test for the other 14 states of A, we can construct the full transition relation Δ, eventually obtaining the Büchi automaton depicted in Fig. 4.4.

Finally, the acceptance conditions $\mathcal{F}$ are determined by the acceptance conditions of A_M and $A_{\Box\Diamond p_2}$. Because A_M does not have any acceptance conditions, but $A_{\Box\Diamond p_2}$ has one acceptance condition $\{q_0', q_1'\}$, A will have one acceptance condition, so

$$\begin{aligned}\mathcal{F} \;=\; \big\{\{&(q_0, q_0'), (q_1, q_0'), (q_2, q_0'), (q_3, q_0'), (q_4, q_0'),\\ &(q_0, q_1'), (q_1, q_1'), (q_2, q_1'), (q_3, q_1'), (q_4, q_1')\}\big\}.\end{aligned}$$

As before, the states belonging to the only acceptance condition of A are marked with a double circle in Fig. 4.4.

By Lemma 2, we know that the automaton A will accept an infinite word over 2^{AP} if and only if the word is accepted by both A_M and $A_{\Box\Diamond p_2}$. As an example of such an accepting execution, we can take the sequence

$$r = \langle (q_0, q_0'), (q_1, q_2'), (q_3, q_2'), (q_0, q_1'), (q_1, q_2'), (q_3, q_2'), (q_0, q_1'), \ldots \rangle$$

By Lemma 2, this sequence corresponds to the accepting executions

$$r_{A_M} = \langle q_0, q_1, q_3, q_0, q_1, q_3, \ldots \rangle \text{ and } r_{A_{\Box\Diamond p_2}} = \langle q_0', q_2', q_2', q_1', q_2', q_2', q_1', \ldots \rangle$$

of A_M and $A_{\Box\Diamond p_2}$, respectively. (All executions of A_M are accepting.)

By Lemma 1, r_{A_M} corresponds to some execution in the Kripke structure M from which A_M was originally constructed (cf. Example 5, page 15). In addition, by collecting the labels of the transitions in r we obtain the temporal interpretation $\langle \{p_1\}, \{p_1\}, \{p_1, p_2\}, \{p_1\}, \{p_1\}, \{p_1, p_2\}, \ldots \rangle$ of this system execution. As seen in Example 2 (page 8), we know that the LTL formula $\Box\Diamond p_2$ holds in this execution. ■

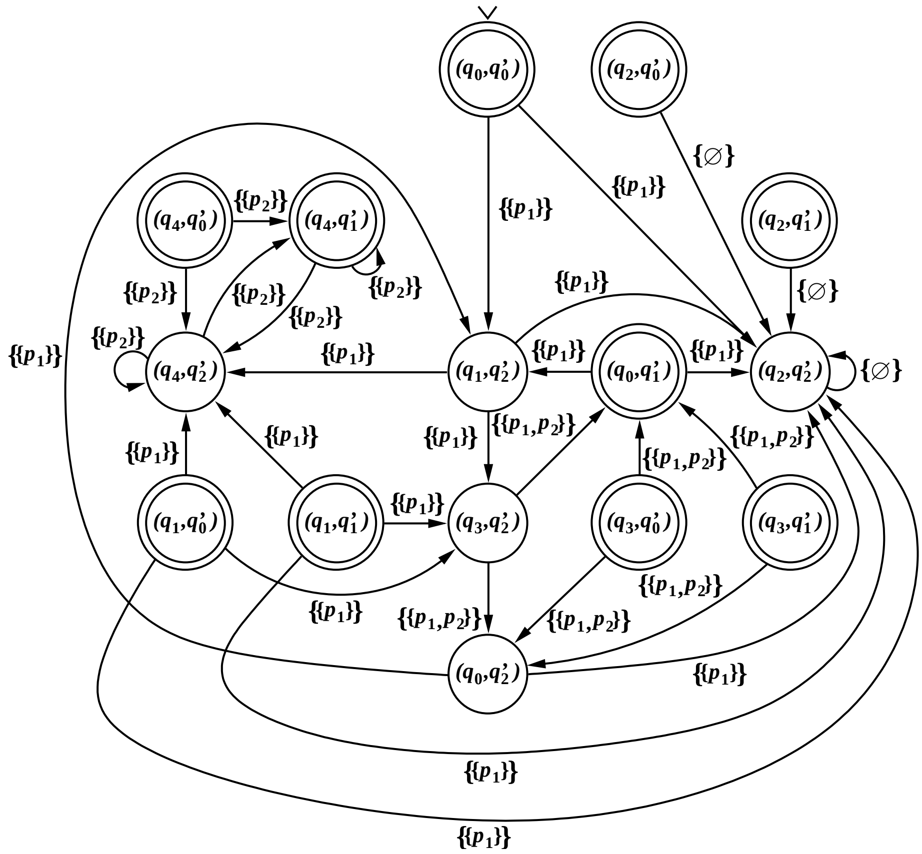


Fig. 4.4: Synchronous product of two Büchi automata

4.2.4 Solving the LTL Model Checking Problem Using Büchi Automata

Using the synchronous product, we can (for any Kripke structure M and any Büchi automaton $A_{\neg\varphi}$ corresponding to the negation of a given LTL formula) now construct a Büchi automaton that recognizes the language $\mathcal{L}_M \cap \mathcal{L}_{\neg\varphi}$. As mentioned previously, checking this language for emptiness corresponds to checking whether the language $\mathcal{L}_M$ is contained in $\mathcal{L}_\varphi$, and therefore the language can be used to solve the model checking problem for LTL. This result follows directly from the previous two lemmata and can be stated as the following theorem.

Theorem 1 *Let M be a Kripke structure and let $A_{\neg\varphi}$ be a Büchi automaton that accepts the exact set of infinite sequences over 2^{AP} satisfying the linear temporal logic formula $\neg\varphi$. Let A_M be the Büchi automaton obtained from the Kripke structure M using the construction described in Lemma 1.*

The Büchi automaton $A_M \otimes A_{\neg\varphi}$ accepts no input word over 2^{AP} if and only if $M \models \varphi$.

Proof: By Lemma 1, the automaton A_M accepts an infinite word over 2^{AP} if and only if the word is the temporal interpretation of some execution of M.

Lemma 2 shows that the synchronous product $A_M \otimes A_{\neg\varphi}$ of A_M and $A_{\neg\varphi}$ is a Büchi automaton that has an accepting execution on an infinite word over 2^{AP} if and only if the word is accepted by both A_M and $A_{\neg\varphi}$, i.e., if and only

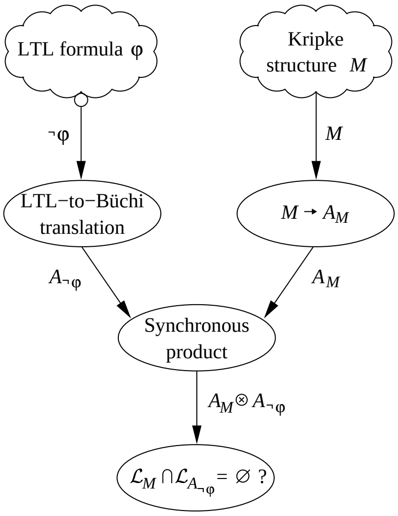


Fig. 4.5: Automata-theoretic model checking procedure for LTL

if it is the temporal interpretation of some execution of the Kripke structure M, and it has the linear temporal property $\neg\varphi$. Therefore, $A_M \otimes A_{\neg\varphi}$ has no accepting executions if and only if no execution of M satisfies the property $\neg\varphi$. By the semantics of LTL, this holds if and only if all executions of M satisfy the property φ, i.e., if and only if $M \models \varphi$. $\square$

By Theorem 1, the model checking problem can now be solved by first converting the system model M into a corresponding Büchi automaton A_M, then intersecting this automaton with the Büchi automaton $A_{\neg\varphi}$ constructed from the *negation* of some linear temporal logic formula φ, and finally checking the resulting automaton for the existence of accepting executions. As stated in the proof, any accepting execution of the product automaton corresponds to an execution of M that satisfies the property $\neg\varphi$. Therefore, any accepting execution of the product automaton has a corresponding system execution that provides a *counter-example* for the property $M \models \varphi$.

In LTL model checking, checking the emptiness of the language $\mathcal{L}_M \cap \mathcal{L}_{\neg\varphi}$ requires constructing the property automaton for the negated formula $\neg\varphi$. This corresponds to finding an answer to the question whether *all* the system executions satisfy the property φ. However, in some cases we may be only interested to know whether *any* system execution satisfies the property φ individually. From Theorem 1 it follows that this problem is equivalent to model checking the LTL property $\neg\varphi$ in the system. In terms of languages, this problem corresponds to checking the emptiness of the language $\mathcal{L}_M \cap \mathcal{L}_\varphi$ and can be solved using the synchronous product of the system and a Büchi automaton constructed from the property φ itself by following the same steps as above. As a matter of fact, this variant of the LTL model checking problem will be used in the tests for LTL-to-Büchi translation algorithm implementations in Sect. 5.1.2.

The automata-theoretic model checking procedure for linear temporal logic is summarized in Fig. 4.5.

4.2.5 Checking the Existence of Accepting Executions

Solving the LTL model checking problem with Büchi automata still requires checking whether the synchronous product $A_M \otimes A_{\neg\varphi}$ of the system automaton and the property automaton has any accepting executions. This phase is often called the *emptiness check*, since the nonexistence of an accepting execution in the product automaton implies that the language accepted by the automaton is empty. The emptiness check can be done by using the graph-theoretical concept of *maximal strongly connected components* (MSCCs) of a graph. We shall first give a brief description of this concept.

Informally, a subset of nodes of a (finite) graph (a *component* of the graph) is *strongly connected* if and only if there exists a path in the graph with zero or more arcs between any two nodes in the subset. (It is usually assumed that every node is reachable from itself by an empty path.) The strongly connected component is *maximal* if any proper superset of graph nodes covering the strongly connected component is not strongly connected itself. A strongly connected component is called *nontrivial* if there exists a path with at least one arc between any two nodes of the component.

Clearly, the MSCCs of the graph must be disjoint. Namely, if there existed two unequal maximal strongly connected components, whose intersection were nonempty, the union of these components would form another strongly connected component. However, this would be in contradiction with the maximality of the original strongly connected components. Since each node of the graph belongs to some maximal strongly connected component (which may be a trivial MSCC), the union of all MSCCs covers the entire graph.

The executions of a Büchi automaton are in a special relation to the nontrivial MSCCs of the automaton (see e.g. [11]). By definition, each execution of a Büchi automaton $A = \langle \Sigma, Q, \Delta, q^0, \mathcal{F} \rangle$ is an infinite sequence $r \in Q^\omega$. Since Q is finite but r is infinite, there must be at least one state that occurs infinitely many times in r, and therefore $\inf(r) \neq \emptyset$. In addition, no state in $Q \setminus \inf(r)$ occurs in r infinitely often. From this it follows that the execution can be divided into a finite prefix of states of Q followed by an infinite subsequence of states in $\inf(r)$. By definition of $\inf(r)$, each state in $\inf(r)$ still occurs infinitely often in this subsequence. Then, any two states in $\inf(r)$ must be *reachable* from each other in the Büchi automaton by a path with at least one arc, since otherwise these states could not both belong to $\inf(r)$. Because of this property, $\inf(r)$ is actually a *nontrivial strongly connected component* of the automaton. Therefore, there exists also a (unique) nontrivial *maximal* strongly connected component that includes $\inf(r)$.

According to Definition 5, any accepting execution r of A contains (for all acceptance conditions $F_i \in \mathcal{F}$, $1 \leq i \leq n$) a state $q_i \in F_i$ that occurs infinitely often in the execution, and therefore the states q_i ($1 \leq i \leq n$) also belong to $\inf(r)$. From the above discussion, we see that the accepting execution r will eventually remain within some unique nontrivial maximal strongly connected component $C \subseteq Q$ of the automaton. Since this component includes $\inf(r)$, it also holds that for all acceptance conditions $F_i \in \mathcal{F}$ ($1 \leq i \leq n$), $C \cap F_i \neq \emptyset$. Therefore, the MSCC intersects all acceptance conditions of the automaton.

On the other hand, if the automaton has a nontrivial maximal strongly connected component C that is reachable from the initial state of the automaton, and C intersects all acceptance conditions of the automaton, the automaton then has an accepting execution. Namely, we can in this case construct such an execution by first following some path from the initial state of the automaton to some state in C and then extending the path with an infinitely repeating state cycle. The states in the cycle must be chosen so that the cycle contains a state from each acceptance condition of the automaton, and there is an arc from the last state of the cycle back to its first state. The construction of this cycle is possible, because C intersects all acceptance conditions, and all states of C are reachable from each other by the strongly connectedness property. The cycle may actually be constructed in many ways, since the particular order in which the different acceptance conditions are encountered in the cycle is not relevant to Büchi acceptance.

The previous discussion shows that a Büchi automaton has an accepting execution if and only if it contains a nontrivial maximal strongly connected component that intersects all the acceptance conditions of the automaton, and the component is reachable from the initial state of the automaton.

Example 7 *We demonstrate the LTL model checking procedure by checking whether the LTL formula $\neg\Box\Diamond p_2$ holds in the Kripke structure M of Example 1 (page 4). That is, we wish to check whether p_2 holds only finitely often in all executions of M. (We already argued in Example 2 that this is not the case; we shall now show this using the systematic LTL model checking procedure.)*

Model checking the LTL formula $\neg\Box\Diamond p_2$ requires first computing the synchronous product of the Büchi automaton A_M (see Example 5) with the Büchi automaton $A_{\neg\neg\Box\Diamond p_2}$ constructed for the negation of the property to be checked. Since the negated property $\neg\neg\Box\Diamond p_2$ is logically equivalent to $\Box\Diamond p_2$ by the semantics of LTL, we shall actually need to construct the automaton $A_{\Box\Diamond p_2}$. We have already done this in Example 4 (page 13). We have also already computed the required product automaton $A_M \otimes A_{\Box\Diamond p_2}$ in Example 6 (see Fig. 4.4). Thus, the only remaining task is to check whether the product automaton has any accepting executions starting from the state (q_0, q'_0).

Using the product state notation from Example 6, the maximal strongly connected components of the product automaton are

$$\begin{array}{rcl@{\qquad\qquad}rcl}
C_1 & = & \{(q_0,q'_0)\} & C_7 & = & \{(q_2,q'_1)\} \\
C_2 & = & \{(q_1,q'_0)\} & C_8 & = & \{(q_3,q'_1)\} \\
C_3 & = & \{(q_2,q'_0)\} & C_9 & = & \{(q_0,q'_1),(q_0,q'_2),(q_1,q'_2),(q_3,q'_2)\} \\
C_4 & = & \{(q_3,q'_0)\} & C_{10} & = & \{(q_2,q'_2)\} \\
C_5 & = & \{(q_4,q'_0)\} & C_{11} & = & \{(q_4,q'_1),(q_4,q'_2)\} \\
C_6 & = & \{(q_1,q'_1)\} & & &
\end{array}$$

(See Sect. 4.2.6 for a discussion on how the MSCCs can be computed in practice.)

We see that the components $C_1, \ldots, C_8$ are trivial and can be discarded. Of the remaining components, we check whether any of them intersects any of the acceptance conditions of the product automaton. (As seen in Example 6, the product automaton has only one acceptance condition, because $\mathcal{F} = \{F\} = \{\{(q_0,q'_0),(q_1,q'_0),(q_2,q'_0),(q_3,q'_0),(q_4,q'_0),(q_0,q'_1),(q_1,q'_1),$

$(q_2, q_1'), (q_3, q_1'), (q_4, q_1')\}\})$. *We see that* C_9, C_{10} *and* C_{11} *are all reachable from the initial state* (q_0, q_0'), *and*

$$\begin{array}{rcll} C_9 \cap F & = & \{(q_0, q_1')\} \neq \emptyset, & \\ C_{10} \cap F & = & \emptyset, & \text{and} \\ C_{11} \cap F & = & \{(q_4, q_1')\} \neq \emptyset. & \end{array}$$

The existence of reachable nontrivial MSCCs that intersect the single acceptance condition now confirms that the property does not hold in the Kripke structure. We can construct an accepting execution for the automaton e.g. with the component C_{11} *by first taking the path* $\langle (q_0, q_0'), (q_1, q_2'), (q_4, q_2') \rangle$ *to reach the component, and then extending the path with the cycle* $\langle (q_4, q_2'), (q_4, q_1'), (q_4, q_2'), (q_4, q_1'), \ldots \rangle$ *that visits the state* $(q_4, q_1') \in F$ *infinitely often. We thus obtain the accepting execution*

$$\langle (q_0, q_0'), (q_1, q_2'), (q_4, q_2'), (q_4, q_1'), (q_4, q_2'), (q_4, q_1'), \ldots \rangle.$$

By Lemma 2, this execution corresponds to the execution $\langle q_0, q_1, q_4, q_4, q_4, \ldots \rangle$ *of the automaton* A_M. *This sequence in turn corresponds to the execution* $\langle s_0, s_1, s_4, s_4, s_4, \ldots \rangle$ *of the Kripke structure* M *(by Lemma 1). It is easy to see that* p_2 *holds infinitely often in this execution, so it is indeed a counter-example for the LTL property* $\neg\Box\Diamond p_2$. ■

4.2.6 Implementation Considerations

This section describes a straightforward way to implement the final steps of the LTL model checking procedure using simple explicit representations for the state space and the Büchi automata. We assume that we already have the Büchi automaton $A_{\neg\varphi}$ corresponding to the negation of a given LTL formula φ and the automaton A_M corresponding to some Kripke structure M.

The construction of the synchronous product (as given in Lemma 2) results in a graph whose number of states always equals the product of the numbers of states in the automata corresponding to the Kripke structure and to the LTL property, respectively. Also the number of transitions in the product is dependent on the numbers of states in these automata, and it is additionally bounded by the size of the alphabet 2^{AP} common to both automata. However, a direct implementation of this construction may produce an automaton having states that are not reachable from its initial state. Clearly, since no execution of any Büchi automaton can visit any unreachable states, these states can be removed from the product without changing the language recognized by the automaton.[2] It is sufficient to compute the product as the minimal set of product states that includes the initial state and is closed under the transition relation. In practice, this can be done using a straightforward algorithm that constructs the product by performing e.g. a depth-first search in the structure. The search starts from the initial state of the structure and then extends the structure with new states as required by the transition relation. Furthermore, the search algorithm can be implemented so that it can operate directly on the Kripke structure M instead of the Büchi automaton

[2]Returning to the product automaton of Example 6, we could use this fact to remove the states (q_1, q_0'), (q_2, q_0'), (q_3, q_0'), (q_4, q_0'), (q_1, q_1'), (q_2, q_1') and (q_3, q_1').

A_M corresponding to the Kripke structure, so an explicit automaton conversion is not required. This is easy to see from the similarity between Kripke structures and the corresponding automata defined in Lemma 1.

Checking the synchronous product for emptiness requires finding the product automaton's maximal strongly connected components reachable from its initial state. These can be computed in linear worst-case time in the size of the product using e.g. the algorithm due to Tarjan [25]. From these components it is easy to discard the trivial ones by checking that each component either contains at least two states, or that the state in each single-state component is connected to itself with an edge. It is also straightforward to check whether any nontrivial MSCC intersects all acceptance sets of the automaton by simply taking the union of the sets of acceptance conditions associated with the states in the MSCC. The component can be discarded if the union comprises only an incomplete set of acceptance conditions, since no accepting execution of the automaton can then stay in that component forever.

If the maximal strongly connected components are computed by using a depth-first search algorithm (such as Tarjan's algorithm) starting from the initial state of the product, the reachability of each component from the initial state is guaranteed. If the automaton has a nontrivial MSCC that intersects all acceptance conditions, we can then construct an actual execution of the product automaton to obtain a counter-example for the property $M \models \varphi$. This execution can be built by first finding a path from the initial state of the product to some state in the MSCC and then extending the path with an accepting cycle in the MSCC. The path from the initial state to the MSCC can be obtained directly from the depth-first search stack used for searching the MSCCs. The construction of a cycle that intersects all acceptance conditions requires an additional search inside the MSCC. This search can be done in quadratic time in the size of the component for any number of acceptance conditions, e.g., by using the techniques presented in [13] or [15].

In practice, it is possible to combine the computation of the synchronous product with the check for accepting executions in the product space. This results in an *on-the-fly* model checking algorithm, whose advantage over the straightforward method described above is that it may be able to find an accepting execution (i.e., a counter-example for the property to be verified) without exploring the full product space. Since a single counter-example is all that is needed for proving an LTL property false, it may therefore be possible to do verification in less space if the property does not hold in the given Kripke structure. Techniques for on-the-fly emptiness checking and counter-example generation have been presented in [3], [4] and [15]. In this work, however, we shall need only small Kripke structures, so extreme memory-efficiency in the search for accepting executions is not of primary importance. For simplicity, we shall therefore keep the product computation and the search for accepting executions separate.

5 TESTING LTL TRANSLATION INTO BÜCHI AUTOMATA

One of the most difficult phases in the automata-theoretic LTL model checking procedure is obtaining a Büchi automaton from an LTL formula. In addition to the relative conceptual complexity of the translation in comparison to the other phases of the LTL model checking procedure, difficulties are caused by the need for an efficient implementation that generates as small automata as possible from the input formulae. The need for small automata arises from the exponential worst-case impact (in the length of the LTL formula) that the size of an automaton may have on the memory requirements of the model checking process [32]. There is no known general procedure to do the translation efficiently in a minimal way, and the best known methods rely on various heuristics in order to minimize the size of the automata [24, 6]. The optimizations made for reducing the size of the automata can therefore increase the complexity of the translation algorithm implementation and make it more prone to errors.

As the previous discussion shows, Büchi automata can be considered to represent languages of infinite words corresponding to the models of LTL formulae, and LTL-to-Büchi translation algorithms provide the tools for systematically constructing the automata from the formulae. However, errors in the implementation of these algorithms may occasionally cause the translation of some formulae to fail. In these cases, the translator produces an *incorrect* Büchi automaton that may in fact recognize a completely different language from the one corresponding to the property. Using such an automaton in the following model checking phases will then invalidate all model checking results regarding the intended property. In some cases the model checking tool may not even provide any evidence to the user that one of the model checking phases may have failed.

The following sections describe methods for testing the LTL-to-Büchi translation phase of LTL model checking. Testing could certainly help in improving the reliability of other parts of model checkers. Of course, actual tools differ very much in their implementation details, so it may be difficult to find general methods suitable for automated testing of different model checkers. However, most of the available implementations include the LTL-to-Büchi translation algorithm in a separate "black box" program module. In addition, the input for this phase—the LTL formula—is not usually dependent on any previous computation, since the formula is usually supplied by the user. Therefore, testing this phase of the model checking procedure can be done using general techniques and can even be automated to some extent, which allows the test methods to be applied to real model checking tools.

The test methods to be presented are based on the direct analysis of Büchi automata obtained using different LTL-to-Büchi translation algorithm implementations, and a more indirect way for testing the correctness of the different translation algorithms through using the entire LTL model checking procedure. The testing involves running the translation algorithm implementations on given LTL formulae and checking the obtained automata for consistency, together with model checking the formulae in given Kripke structures. In practice, these LTL formulae and Kripke structures can even be generated automatically using randomized techniques. An essential part of testing is the *cross-comparison* of different translation algorithm implementations. Basically, this means running several translation algorithm implementations on the same input and then checking whether the results obtained using the different implementations are consistent. Further analysis of contradictory results provides a way to determine which of the implementations had failed.

5.1 TEST METHODS FOR LTL-TO-BÜCHI TRANSLATION

The following subsections describe tests [26, 27] that can be made on the Büchi automata and the model checking results obtained using the LTL-to-Büchi translation algorithms to be tested. In order to automate the tests into a reliable implementation for testing the correctness of LTL-to-Büchi translators, the implementation should be kept as simple as possible. For this reason, the difficulty of implementing each of the presented tests is also considered. Although the tests require input, LTL formulae and Kripke structures, to be used for running the LTL-to-Büchi translation algorithms and the LTL model checking procedure, the tests are independent of any particular kind of formulae or Kripke structures.

5.1.1 Analysis of Büchi Automata

A natural approach for testing the correctness of LTL-to-Büchi translation algorithm implementations is to try to directly analyze the automata generated by the implementations. The analysis methods can be based on the semantics of linear temporal logic.

Let $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$ denote the languages corresponding to the sets of models of a given LTL formula φ and its negation, respectively. By the semantics of LTL, no infinite sequence over 2^{AP} can satisfy both an LTL formula and its negation, and therefore the languages $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$ must be disjoint. On the other hand, any infinite sequence of subsets of AP satisfies either an LTL formula or its negation, again by the semantics of LTL. Therefore, the languages $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$ are in fact *complementary* to each other with respect to the set $(2^{AP})^\omega$, so it must be that for any LTL formula φ, $\mathcal{L}_\varphi \cap \mathcal{L}_{\neg\varphi} = \emptyset$, and $\mathcal{L}_\varphi \cup \mathcal{L}_{\neg\varphi} = (2^{AP})^\omega$.

The fact that $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$ are disjoint provides a partial correctness test for two Büchi automata constructed from the formula φ and from its negation $\neg\varphi$ using an LTL-to-Büchi translation algorithm implementation (or even two different implementations). The correct translation of these formulae should result in two Büchi automata recognizing the languages $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$. Let A_φ and $A_{\neg\varphi}$ denote the Büchi automata obtained from φ and $\neg\varphi$, respectively. The synchronous product of these automata can be used to test the automata for their expected properties. By Lemma 2, the product automaton accepts an infinite word over 2^{AP} if and only if the word is accepted by both A_φ and $A_{\neg\varphi}$, i.e., if and only if the word belongs to the intersection of the languages recognized by the automata. Since the language intersection $\mathcal{L}_\varphi \cap \mathcal{L}_{\neg\varphi}$ is known to be empty, the synchronous composition of A_φ and $A_{\neg\varphi}$

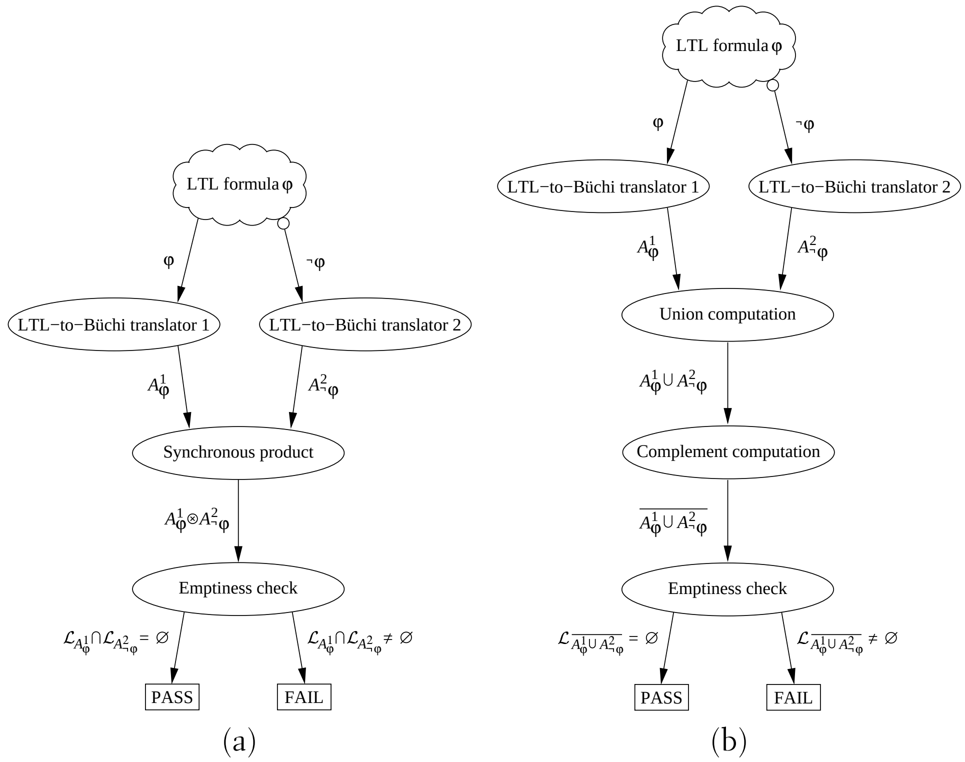


Fig. 5.1: (a) Emptiness check for the intersection of Büchi automata A_φ and $A_{\neg\varphi}$. (b) Universality check for the union of A_φ and $A_{\neg\varphi}$

should therefore have no accepting executions, which can be confirmed by checking the product automaton $A_\varphi \otimes A_{\neg\varphi}$ for emptiness. However, if the product automaton is nonempty, then at least one of the automata A_φ and $A_{\neg\varphi}$ does not correctly recognize the expected language, and therefore the LTL-to-Büchi translation of at least one of these formulae must have failed. In this case, the techniques described in Sect. 4.2.5 can be used to construct an infinite word over 2^{AP} incorrectly accepted by both A_φ and $A_{\neg\varphi}$ by extracting it from an accepting execution of the product automaton. If both automata A_φ and $A_{\neg\varphi}$ were constructed using the same LTL-to-Büchi translation algorithm implementation, the failed emptiness check immediately confirms an error in the implementation.

The above steps are collected below into Test 1. See also Fig. 5.1 (a).

Test 1 (Emptiness check for the intersection of A_φ and $A_{\neg\varphi}$)
Input: an 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 synchronous product $A_\varphi \otimes A_{\neg\varphi}$.*

3. *Check $A_\varphi \otimes A_{\neg\varphi}$ for emptiness. If the product automaton accepts any input word, then either A_φ or $A_{\neg\varphi}$ does not correctly recognize the language $\mathcal{L}_\varphi$ or $\mathcal{L}_{\neg\varphi}$, respectively. This suggests that the translation of at least one of the formulae into a Büchi automaton has failed.*

■

We shall shortly address the question why it can be useful to apply two *different* LTL-to-Büchi translators for constructing the automata A_φ and $A_{\neg\varphi}$ required in Test 1. However, we first introduce another similar consistency check applicable to the automata A_φ and $A_{\neg\varphi}$.

Test 1 is not complete for showing the correctness of an LTL-to-Büchi translation algorithm implementation even on a single LTL formula. For example, it is easy to see that an implementation that "cheats" by always generating an empty automaton (i.e., an automaton that rejects all its inputs) regardless of the input formula would trivially pass this test, since intersecting any automaton with an empty automaton results in an empty product automaton.

In principle, the fact that the *union* of $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$ forms the universal language $(2^{AP})^\omega$ provides another test to be used together with Test 1 in order to confirm that the languages recognized by the automata A_φ and $A_{\neg\varphi}$ are complementary to each other. It can be shown (see e.g. [30]) that any two Büchi automata can be combined into another Büchi automaton that accepts precisely the union of the languages recognized by the original automata. Therefore, it might be possible to check whether the automaton $A_\varphi \cup A_{\neg\varphi}$ accepts the universal language, i.e., that the automaton accepts every input word over 2^{AP}. The existence of an input word *not* accepted by this automaton would again suggest that one of the LTL-to-Büchi translation algorithm implementations has an error.

Unfortunately, the universality test for a Büchi automaton is not as easy to perform in practice as the emptiness check—as a matter of fact, this problem is known to be PSPACE-complete [30]. The language universality test might first be reduced to a language emptiness check, which can be solved using Büchi automata: the fact that the language $\mathcal{L}_\varphi \cup \mathcal{L}_{\neg\varphi}$ is universal implies that its complement $\overline{\mathcal{L}_\varphi \cup \mathcal{L}_{\neg\varphi}}$ must be empty. However, this reduction differs from all previous operations on Büchi automata in that it involves the *complementation* of nondeterministic Büchi automata. Although Büchi automata are closed under complementation, the complementation construction [23] is relatively hard to implement in comparison to the other operations applied to Büchi automata so far. In addition, even the optimal construction may cause an exponential ($2^{\mathcal{O}(n \log n)}$) worst-case blow-up in the size of the automaton. The blow-up is a consequence of the nondeterminism of Büchi automata and cannot in general be avoided [23].

Although the language union universality test was not used in the experiments made in this work on real LTL-to-Büchi translation algorithm implementations, the required steps are collected below in Test 2. The steps are illustrated in Fig. 5.1 (b).

Test 2 (Universality check of the union of A_φ and $A_{\neg\varphi}$)
Input: an 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 $A_{\neg\varphi}$ (see e.g. [30]).*

3. *Using a Büchi automata complementation procedure (see [23]), 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 $\mathcal{L}_\varphi$ or $\mathcal{L}_{\neg\varphi}$, respectively. This suggests that the translation of at least one of the formulae into a Büchi automaton has failed.*

■

Taken together, Tests 1 and 2 are able to show that the languages recognized by two Büchi automata A_φ and $A_{\neg\varphi}$, constructed using the same implementation from some input formula φ, are complementary to each other. Although this is already a valuable result in itself, the tests are not powerful enough to prove the correctness of an LTL-to-Büchi translation algorithm implementation on any input formula even if both of the tests succeed. The tests only confirm that the relationship between the languages recognized by the two automata is as expected; however, this is not sufficient for telling whether the languages correctly correspond to the models of the LTL properties involved. Therefore, these tests may fail to detect some systematic errors in the translation. For example, if an implementation erroneously mixed the names of the atomic propositions in the given input formula such that otherwise independent propositions share the same name, the automaton generated by the implementation would not correctly recognize the models of the original formula.

This problem can be helped by using two or more independent LTL-to-Büchi translators for the formula translation as suggested above. Instead of performing the checks only with automata obtained using a single implementation i (chosen from a set of implementations I), each of the implementations can be used in turn to convert φ and $\neg\varphi$ into Büchi automata. Tests 1 and 2 can then be repeated on each pair of automata A^i_φ and $A^j_{\neg\varphi}$ constructed by any two implementations $i \in I$ and $j \in I$, respectively. Since the LTL formula φ uniquely defines its set of models (i.e., the languages $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$), all Büchi automata constructed from φ ($\neg\varphi$) using the different implementations should accept the same language $\mathcal{L}_\varphi$ ($\mathcal{L}_{\neg\varphi}$). Therefore, no synchronous composition of any two automata A^i_φ and $A^j_{\neg\varphi}$ for some $i, j \in I$ should have any accepting executions, and the same should also hold for the automata $\overline{A^i_\varphi \cup A^j_{\neg\varphi}}$. A successful run of all these tests proves that the languages accepted by the automata A^i_φ ($A^i_{\neg\varphi}$) are equivalent. This would finally prove the correctness of the tested implementations on the formulae φ and $\neg\varphi$, *provided that at least one of the implementations participating in the tests is already known to be correct.*

The lack of an implementation that is known to be correct still leaves a small possibility for a *false positive*, i.e., a case in which all eight possible tests[1] between the automata generated by some two implementations succeed, but the automata are still incorrect. This will occur if the languages recognized by all automata constructed from the same formula are equivalent to each other but still not equivalent to the language corresponding to the models of the formula. (For example, consider two otherwise correct LTL-to-Büchi translator implementations, both of which negate every input formula before

[1] 1. "$A^i_\varphi \cap A^i_{\neg\varphi} = \emptyset$?" 2. "$\overline{A^i_\varphi \cup A^i_{\neg\varphi}} = \emptyset$?" 3. "$A^j_\varphi \cap A^j_{\neg\varphi} = \emptyset$?" 4. "$\overline{A^j_\varphi \cup A^j_{\neg\varphi}} = \emptyset$?" 5. "$A^i_\varphi \cap A^j_{\neg\varphi} = \emptyset$?" 6. "$\overline{A^i_\varphi \cup A^j_{\neg\varphi}} = \emptyset$?" 7. "$A^j_\varphi \cap A^i_{\neg\varphi} = \emptyset$?" 8. "$\overline{A^j_\varphi \cup A^i_{\neg\varphi}} = \emptyset$?"

translation.) Intuitively, the probability of a false positive should decrease if another independent implementation is included in the tests.

However, even if none of the implementations is known to be correct, which is likely to be the case in practice, testing different implementations against each other still increases the intuitive confidence in the correctness of the automata if no failures are detected. This view is based on the assumption that two independent implementations are not likely to fail in the same way on the same input formula (i.e., by generating equivalent but incorrect automata from the formula). Thus, an inconsistency is likely to be detected in at least one of the eight possible tests that can be made on the Büchi automata generated by two different implementations if one of the tested translators has an error.

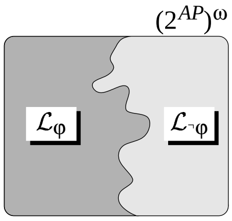
None of the tests can give *false negative* answers, however. This is because a test failure between two automata always implies that at least one of the automata is incorrect:

- A failure in Test 1 between two implementations $i, j \in I$ implies the existence of an input $\xi \in (2^{AP})^\omega$ recognized by both of the automata A^i_φ and $A^j_{\neg\varphi}$. (If necessary, such an input can be constructed by the same techniques used for extracting counter-examples for LTL properties from product automata. See Sect. 4.2.6.) At least one of these automata *must* now be incorrect, since no infinite sequence over 2^{AP} can be a model of φ and $\neg\varphi$ at the same time. Therefore, one of the automata incorrectly accepts ξ. The other automaton also accepts ξ, but this test does not give useful information about the correctness of that automaton: for example, it may be that the automaton accepts every input, although neither of the formulae is actually valid. Distinguishing the certainly incorrect automaton from the two will be discussed later in Sect. 5.2.

- Similarly, a failure in Test 2 between two implementations $i, j \in I$ implies that there exists a sequence $\xi \in (2^{AP})^\omega$ *rejected* by both of the automata A^i_φ and $A^j_{\neg\varphi}$. (Such a sequence could again be constructed during the emptiness check of $\overline{A^i_\varphi \cup A^j_{\neg\varphi}}$.) The techniques used for distinguishing the incorrect automaton in Test 1 can be applied also to this case to determine which one of the automata incorrectly rejects ξ. (As above, nothing can be said about the absolute correctness of the other automaton.)

The different types of tests and the types of errors they are able to detect are summarized in Fig. 5.2. However, because Test 2 is difficult to implement, relying only on Test 1 is likely to reduce the overall efficiency of finding errors in the implementations with the testing procedure. The next subsection discusses some alternative testing methods based on more easily implementable techniques that can help to improve testing efficiency.

The previous tests have the advantage of being independent of the chosen LTL formula φ. Therefore, these tests can use even random LTL formulae that are quite easy to generate automatically. Previous experiments [26, 27] suggest that even simple randomly generated input can be of use in finding



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

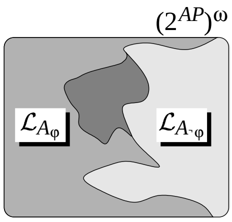
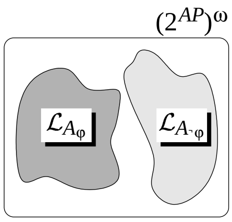
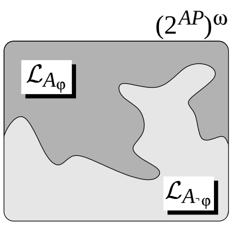
Relationship between languages recognized by two automata	Error	Detectable by
	The languages recognized by the automata are not disjoint	Test 1
	The union of the languages recognized by the automata is not the universal language	Test 2
	The languages recognized by the automata are complementary but still incorrect	Running Tests 1 and 2 using several independent implementations (may still result in a false positive unless a correct implementation is available)

Fig. 5.2: Examples of incorrect relationships between languages accepted by two Büchi automata constructed from φ and $\neg\varphi$ by two LTL-to-Büchi implementations and how to detect them

errors in LTL-to-Büchi translators. The random testing strategy is also the approach taken in this work; the details will be discussed later in Chap. 6.

Limited testing could still be done on the implementations using specially chosen LTL formulae: for example, no Büchi automaton constructed from an *unsatisfiable* LTL formula should have any accepting executions. Another weak but simple consistency check would be to test that the Büchi automaton constructed from a *valid* LTL formula has even one accepting execution. Checks based on these properties could again be implemented by direct application of the emptiness check to the automata. Of course, a clever implementation might be able to detect the validity or unsatisfiability of a formula directly (e.g., from the syntactic structure of the formula), without actually performing translation using more general techniques. Therefore, relying only on such special cases may not be sufficient for testing all parts of the implementation; assessing the coverage of this kind of testing will require taking the implementation details into account. Furthermore, these tests require LTL formulae with known special properties, which makes it more difficult to generate the formulae automatically. Of course, one could simply use a preselected collection of valid or unsatisfiable formulae instead and test only a few selected cases.

This work, however, focuses on finding reasonably general testing methods for LTL-to-Büchi translators. For that reason, no testing methods based on LTL formulae with special properties will be used, since their effectiveness on a particular implementation depends more closely on the details of the implementation. Instead, this work continues using the random input approach, treating the tested implementations simply as "black boxes" without looking at their internal details. (Certainly, choosing a set of test formulae with detailed knowledge about the structure of an implementation may result in more effective tests for that particular implementation, so this kind of testing is not a bad strategy e.g. in the development of a new implementation.)

5.1.2 Using the LTL Model Checking Procedure

The decision of implementing only Test 1 into an automated testing tool is likely to reduce the tool's effectiveness in finding errors in LTL-to-Büchi translation algorithm implementations, unless alternative testing methods are used to remedy this problem. This subsection discusses how the LTL model checking procedure (described in Chap. 4) can be applied to testing the correctness of LTL-to-Büchi translators.

Given a Kripke structure M, the semantics of LTL guarantee that the truth value of any LTL formula is uniquely defined in the structure. Therefore, no matter which methods are used to model check a given LTL formula φ in M, the methods should always give the same answer to the question whether φ holds in M, provided that all the used methods are sound and complete and they are applied correctly. Since the abstract automata-theoretic model checking procedure for LTL is sound and complete (by Theorem 1), the correctness of the model checking results given by an LTL model checking procedure implementation depends on whether all phases of the procedure are free of implementation errors.

Therefore, if there are several LTL-to-Büchi translation algorithm implementations available, each of them can be used in turn to convert a given LTL formula into a Büchi automaton, which is then used to model check the formula in a given Kripke structure. In effect, using different implementations for the formula translation now corresponds to having several "model checking procedures", all of which should give the same answer if none of the LTL-to-Büchi implementations have errors. Inconsistencies in the answers then suggest that some of the LTL-to-Büchi translation algorithms are in error, or that an error occurred during some later phase in the LTL model checking procedure.

Admittedly, applying the full LTL model checking procedure to test the correctness of only one of its phases seems more complicated than the direct analysis of Büchi automata. Furthermore, inconsistencies in the model checking results may not necessarily originate from the LTL-to-Büchi translation phase, but some other phase instead. Therefore, it might seem questionable whether this method is effective and easily implementable enough for uncovering errors particularly in the LTL-to-Büchi translation phase. However, this approach is justified because of the following main reasons:

- All phases of the LTL model checking procedure after the LTL-to-Büchi translation can be integrated into a *common* implementation framework. This helps in trying to isolate the source of model checking result inconsistencies into the formula translation phase that is performed with the tested translation algorithm implementations. In principle, this results in "an LTL model checker with a replaceable LTL-to-Büchi translation module".

- Extreme memory-efficiency is not of primary importance for the purposes of plain testing, since it is not necessary to use real-sized examples of Kripke structures as test cases. Therefore, it may be acceptable to implement the LTL model checking phases with very straightforward algorithms, such as those described in Sect. 4.2.6. In addition, all of these algorithms are also conceptually more simple than the algorithms needed for LTL-to-Büchi translation. Actually, the algorithms for computing the synchronous product of two Büchi automata and checking it for emptiness by examining its nontrivial MSCCs can be implemented as refinements of a basic graph depth-first search.

For testing purposes, the LTL model checking procedure can be simplified slightly. To model check an LTL formula φ in a given Kripke structure M, the formula would normally need to be *negated* first to obtain the Büchi automaton to be used for checking the language $\mathcal{L}_M \cap \mathcal{L}_{A_{\neg\varphi}}$ for emptiness. However, when testing LTL-to-Büchi translators, the actual answer to the question whether the LTL formula φ holds in the Kripke structure is not relevant, since the formula might have been generated randomly (and therefore it may not even represent any "useful" property). It is more important to see whether the LTL model checking procedure *gives the same model checking results* for the formula, whichever of the tested LTL-to-Büchi translation algorithm implementations is used in the model checking process. Therefore, it is not necessary to negate the formula φ before constructing a Büchi automaton. Instead, it is possible to simply convert φ itself into an automaton and then proceed with synchronizing the automaton with the Kripke structure as before. As discussed at the end of Sect. 4.2.4, the resulting product automaton can be used to tell whether *any* of the executions of the Kripke structure satisfies the property φ individually. It is clear that the answer to this question should also remain the same regardless of the methods used for solving this problem—provided that they are sound and complete, of course. This "modified" model checking procedure is actually equivalent to checking whether the LTL formula $\neg\varphi$ holds in the structure.

A simple additional refinement of this testing method allows easily "reusing" a single Kripke structure to obtain more data for comparison. By Lemma 2, every execution of the synchronous composition of two Büchi automata corresponds to two synchronous executions of the original automata (in this case, the system automaton A_M and the property automaton A_φ). These synchronous executions begin in the respective initial states of the structures. However, *changing* the initial state of the Kripke structure also changes the set of executions in the structure. Repeating the LTL model checking procedure in the modified structure then gives a *different set of model checking results*, telling whether any infinite path beginning in the *new* initial state of the Kripke structure has the property φ. This allows a new comparison to be made on the results obtained using the different LTL-to-Büchi translation algorithm implementations: inconsistent results again suggest that some of the implementations may have failed. By considering each state of the Kripke structure in turn as the initial state, the check can be repeated as many times as there are states in the Kripke structure.

Although changing the initial state of the Kripke structure essentially creates a new Kripke structure, it is sufficient to synchronize the system automaton A_M with the property automaton *only once*. Several synchronizations would be required only if also the transition relation of M were changed or if the new product automata contained states (i.e., pairs of states chosen from A_M and A_φ, respectively) not included in the result of any previous synchronization. However, changing the initial state of the Kripke structure does not affect its transition relation, and the definition in Lemma 2 guarantees that the product always contains all possible state pairs, independent of the initial state of A_M. Therefore, the product as defined in Lemma 2 can actually be called the *global synchronous product*, since it includes *all* synchronous executions of A_M and A_φ, no matter in what state A_M begins its execution.[2]

Performing all the emptiness checks in the global synchronous product requires changes also in the emptiness checking phase. For example, if the MSCCs of the product are computed using Tarjan's algorithm, the emptiness check can be performed "globally" by simply restarting the MSCC search algorithm in every state (q, q^0), where q is a state of A_M, and q^0 is the initial state of A_φ. During each run of Tarjan's algorithm, each nontrivial MSCC can then be checked for accepting executions as described in Sect. 4.2.5. Let (q, q^0) be the state in which the MSCC algorithm was most recently restarted. If the search finds an accepting execution, the (modified) Kripke structure then has an execution that begins in the state corresponding to the state q of A_M, and this execution satisfies the property φ.

[2]However, see Appendix A for notes on the practical implementation of the global synchronous product and the following emptiness check.

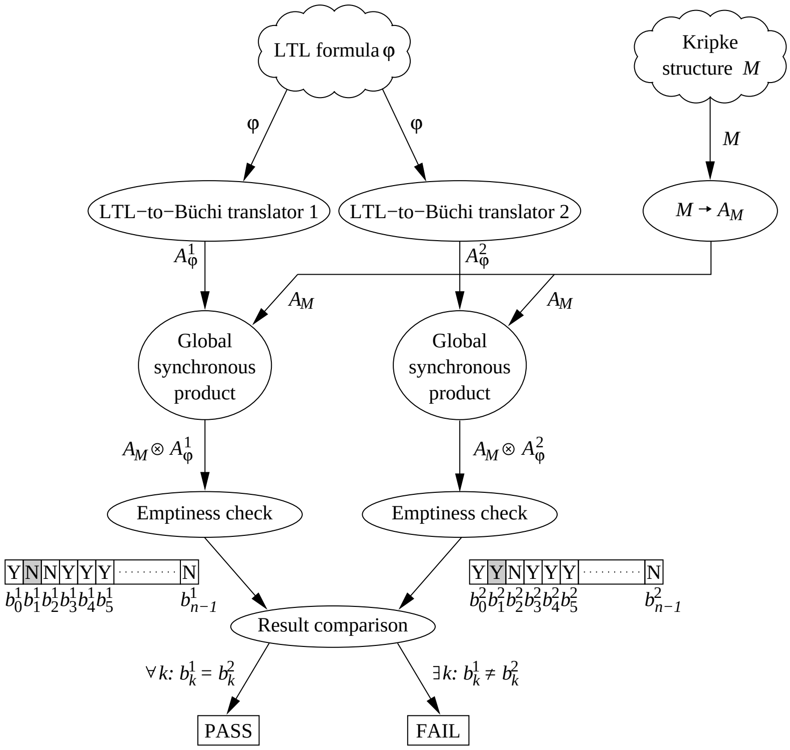


Fig. 5.3: Model checking result cross-comparison check for two LTL-to-Büchi translation algorithm implementations

The above steps are collected together in Test 3. See also Fig. 5.3.

Test 3 (Model checking result cross-comparison check)
Input: Kripke structure M, LTL formula φ.

1. *Convert the formula φ into Büchi automata A_φ^i using each of the available LTL-to-Büchi translation algorithm implementations $i \in I$.*

2. *Compute the global synchronous products $A_M \otimes A_\varphi^i$.*

3. *Check each product automaton $A_M \otimes A_\varphi^i$ for emptiness, i.e., determine for each product state (q_k, q_i^0) (where q_k is a state of A_M and q_i^0 is the initial state of A_φ^i) whether the product automaton has any accepting executions beginning at (q_k, q_i^0). Denote the answers to this question by $b_k^i \in \{$"Y","N"$\}$ such that $b_k^i =$ "Y" if the product automaton $A_M \otimes A_\varphi^i$ can reach an accepting execution from the state (q_k, q_i^0), and $b_k^i =$ "N" otherwise. (These answers also tell whether the Kripke structure M has an execution that satisfies the property φ and begins at the state corresponding to q_k.)*

4. *Test whether for all states q_k of A_M, $\forall i, j \in I : b_k^i = b_k^j$. If this does not hold, one of the LTL-to-Büchi translation algorithms must have failed on the formula φ (under the assumption that all other model checking phases are performed correctly).*

■

As with Tests 1 and 2, any inconsistencies detected in Test 3 do not directly reveal the implementation (or implementations) which had failed. However, if the model checking results obtained using two different LTL-to-Büchi translators are inconsistent in some state of the Kripke structure, at least one of the automata is again certainly incorrect. This follows from the fact that the truth values of the LTL formula φ are uniquely defined in any Kripke structure, so two correct model checking procedures for LTL cannot give a different answer to the existence of an execution satisfying φ in any state of the Kripke structure. However, as in the previous tests, the only thing that can be said about the correctness of the other automaton is that it gives the correct result in one particular state of the Kripke structure M (but not necessarily in other Kripke structures, or even in other states of M). This is all that can be said about the absolute correctness of the tested implementations also in the case when there are *no* inconsistencies in the model checking results, so there is a possibility for false positives. Intuitively, this possibility could again be made smaller by running the cross-comparison tests using several independent LTL-to-Büchi translation algorithm implementations.

Test 3 may detect the inequivalence of languages accepted by Büchi automata constructed by different LTL-to-Büchi translators from the same LTL formula. However, the test is inherently dependent on the Kripke structures used for model checking, and it cannot be practically applied to *proving* the equivalence of the languages accepted by the automata even on a single LTL formula. This makes the test less powerful as Tests 1 and 2 taken together. However, since Test 2 may be difficult to implement and may therefore not be available, Test 3 may improve the odds of detecting errors in LTL-to-Büchi translation algorithm implementations. In addition, Test 3 can be automated quite easily, since the actual test steps are not dependent on the formulae or the Kripke structures used as input. Therefore, Test 3 can be simply run on e.g. randomly generated formulae and Kripke structures.

The rest of this subsection focuses on one additional test to be used as a simple consistency check for a single LTL-to-Büchi translation algorithm implementation. This test is based on the relationship between the satisfiability of φ and $\neg\varphi$ in the same Kripke structure.

In the discussion at the end of Chap. 3, it was noted that it is not possible for both φ and $\neg\varphi$ to hold in the same Kripke structure, although neither of these formulae might be satisfied in the structure. Let φ and M denote a given LTL formula and a given Kripke structure, respectively. By converting the formula φ into a Büchi automaton using an LTL-to-Büchi translator and then checking the product automaton $A_M \otimes A_\varphi$ for emptiness, we can try to see whether any of the executions of M has the property φ. The emptiness of the automaton $A_M \otimes A_\varphi$ suggests that *no* execution of M has this property, and therefore $M \models \neg\varphi$ should hold by the semantics of LTL. Similarly, we can also convert the formula $\neg\varphi$ into another Büchi automaton using the same translator and check the emptiness of the automaton $A_M \otimes A_{\neg\varphi}$. If also this automaton is found to be empty, there is a contradiction, since the emptiness of $A_M \otimes A_{\neg\varphi}$ suggests that also $M \models \varphi$ should hold in the structure. However, this is impossible if $M \models \neg\varphi$ is already known to be true in the structure.[3] Therefore, we must conclude that either the formula φ or

[3]By Definition 1, Kripke structures have a total transition relation. Therefore, there must

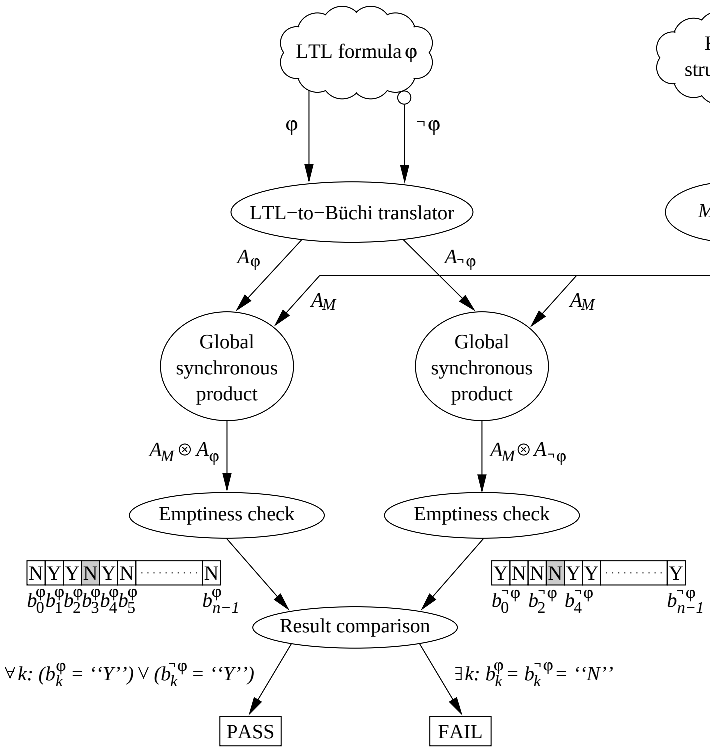


Fig. 5.4: Model checking result consistency check for a single LTL-to-Büchi translation algorithm implementation

$\neg\varphi$ was incorrectly translated into a Büchi automaton, and the LTL-to-Büchi implementation has an error.

This check does not give much useful information about the correctness of the Büchi automata if either of the product automata $A_M \otimes A_\varphi$ or $A_M \otimes A_{\neg\varphi}$ is nonempty. To improve the effectiveness of this test, we can again take advantage of the global synchronous product to obtain more test data from a single Kripke structure by performing the consistency check in each state of the structure.

Performing the consistency check individually on all tested implementations requires constructing two sets of automata A_φ and $A_{\neg\varphi}$. Each of the automata A_φ and $A_{\neg\varphi}$ is synchronized with the system automaton A_M, and the product automata are checked for emptiness. The emptiness check results can now be directly analyzed using also the model checking result cross-comparison check (Test 3). This fact makes it possible to combine Tests 3 and 4 together such that the required product automata need to be computed only once.

A summary of the steps in the model checking result consistency check follows. See also Fig. 5.4.

Test 4 (Model checking result consistency check)
Input: LTL formula φ, Kripke structure M.

---

exist at least one infinite path starting in the initial state of the structure, and this path must satisfy either of the formulae φ or $\neg\varphi$. Therefore, either of the emptiness checks must return with a negative answer.

1. *Construct the automata A_φ and $A_{\neg\varphi}$ from the formulae φ and $\neg\varphi$ using some LTL-to-Büchi translator.*

2. *Compute the synchronous products $A_M \otimes A_\varphi$ and $A_M \otimes A_{\neg\varphi}$.*

3. *Check the product automaton $A_M \otimes A_\varphi$ for emptiness, i.e. check for all states (q_k, q^0) (where q_k is a state of A_M and q^0 is the initial state of A_φ) whether the automaton $A_M \otimes A_\varphi$ has an accepting execution starting in the state (q_k, q^0). Denote the answers to this question by $b_k^\varphi \in \{\text{“Y”},\text{“N”}\}$, where $b_k^\varphi = \text{“Y”}$ if an accepting execution can be reached, and “N” otherwise.*

4. *Repeat Step 3 for the product automaton $A_M \otimes A_{\neg\varphi}$. Denote the obtained answers in this case by $b_k^{\neg\varphi}$.*

5. *Test whether $b_k^\varphi = b_k^{\neg\varphi} = \text{“N”}$ for any state q_k of A_M. 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.*

■

An inconsistency detected in Test 4 reveals the existence of an input not recognized by either of the automata A_φ and $A_{\neg\varphi}$ constructed using some LTL-to-Büchi translator from an LTL formula φ and its negated version $\neg\varphi$, respectively. This means that the union of the languages accepted by the two automata is not the universal language 2^{AP}. Although Test 4 depends on the Kripke structures used for running the test, it may help in detecting some of the errors that would otherwise be left undetected in case Test 2 is not available.

When Tests 3 and 4 are combined together, it is sufficient to perform Test 4 on each pair of automata A_φ and $A_{\neg\varphi}$ generated by a single implementation. This is because performing the test on automata generated by different implementations cannot find any inconsistencies that could not be detected by the other tests.[4]

5.2 TEST FAILURE ANALYSIS

Running different LTL-to-Büchi translators against each other does not still give any information as to *which one* of the tested implementations may have an error, in case some of the tests fail. This makes it difficult to determine which implementation should be fixed. (Tests 1, 2 and 4 can detect the

[4]To see this, assume that two different LTL-to-Büchi translators i and j pass Test 3 against each other on both formulae φ and $\neg\varphi$, and both of the translators also pass Test 4 individually, but Test 4 fails on two automata A_φ^i and $A_{\neg\varphi}^j$ generated by the implementations. (Assume that the indices i and j are chosen such that this holds.) Then, $b_k^{i,\varphi} = b_k^{j,\neg\varphi} = \text{“N”}$ for some state q_k of A_M. Because i and j pass Test 4 individually, it follows that $b_k^{i,\neg\varphi} = b_k^{j,\varphi} =$ “Y”.

Because the implementations also pass Test 3, it must be that $b_k^{i,\varphi} = b_k^{j,\varphi}$ and $b_k^{i,\neg\varphi} = b_k^{j,\neg\varphi}$. Since $b_k^{i,\varphi} = b_k^{j,\neg\varphi} = \text{“N”}$, it now follows that $b_k^{j,\varphi} = b_k^{i,\neg\varphi} = \text{“N”}$. But then it cannot be that the implementations pass Test 4 individually, which is a contradiction.

Therefore, if the implementations pass Test 3 against each other on φ and $\neg\varphi$, and each of them also passes Test 4 individually, they cannot fail Test 4 against each other.

incorrectness of even a single implementation; however, they do not provide any useful information about which one of the automata used in the failed test is clearly incorrect, which might be useful for debugging.)

A simple method for distinguishing the incorrect implementations from the correct ones is to increase the number of independent LTL-to-Büchi translators taking part in the tests and then to try to look for patterns in the detected inconsistencies. For example, if some translator sometimes fails a test against all other tested translators, which in turn pass all tests against each other, there is likely to be an error in that one translator. However, this method might not be applicable if there are not many implementations available, or if the implementations are not independent (e.g., if the implementations to be tested are only different versions of a particular translator).

A unifying fact between all tests is that it is possible to construct a *witness*—an infinite sequence over 2^{AP}—that gives a concrete proof of the test failure. More importantly, however, the same witness can be used to distinguish the incorrect automaton in any pair of two automata for which one of the tests failed. This then reveals an error in the implementation that generated the incorrect automaton. Intuitively, the role of the witness in each failed test is as follows:

(i) In Test 1, the witness is a sequence that is accepted by two automata supposed to recognize two complementary LTL properties φ and $\neg\varphi$, respectively.

(ii) In Test 2, the witness sequence is accepted by neither of two automata supposed to recognize the properties φ and $\neg\varphi$.

(iii) In Test 3, the witness is a sequence that is accepted by one and rejected by the other of two automata, both of which should represent the same property φ.

(iv) Analogously to Test 2, the failure of Test 4 can be proved with a witness that is rejected by both automata A_φ and $A_{\neg\varphi}$ (supposedly) representing two complementary LTL properties.

In the first three cases, the witness can be obtained as a side result of the emptiness check performed on some Büchi automaton in each of the three cases. In Tests 1 and 2, the witness can be extracted from the nonempty synchronous product (or the union complement) automaton constructed from the automata generated by the LTL-to-Büchi translators from the input formulae. In Test 3, the witness can be taken from a nonempty product automaton that claims the existence of a path satisfying the property in some system state in which Test 3 failed.

In case (iv), the witness can be extracted from the system automaton A_M instead of the product automaton used in Test 4. (Of course, the product automaton is still needed for determining the result of Test 4; a witness exists only if the test failed.)

Each case is therefore associated with some *nonempty* Büchi automaton (in case (iv), A_M is nonempty by definition; see Lemma 1). The witness is then constructed from an *accepting execution* of this automaton. This accepting execution can always be constructed so that it consists of a finite prefix of states followed by an infinitely repeating cycle of states. In cases (i)–(iii), such an accepting execution can be found during the emptiness check of a Büchi automaton, using the techniques discussed in Sect. 4.2.5. In case (iv), any execution of A_M having the desired structural properties can be taken as the witness. (Such an execution can be found e.g. by a simple depth-first or breadth-first search in the automaton, stopping as soon as some state of the automaton is visited twice. This is bound to happen, since A_M is finite and has a total transition relation.)

The labels on the automaton transitions in the execution can now be projected onto an infinite sequence ξ over 2^{AP} by selecting any of the symbols in each successive transition label σ into this sequence. It is easy to see from the definition of Büchi automata that the automaton from which the execution was extracted then accepts this sequence. Since the witness execution already has a finite representation, the elements of ξ can be chosen so that ξ forms the concatenation of two finite-length sequences over subsets of AP such that the latter sequence is thought to repeat itself infinitely often.

Let φ and $\neg\varphi$ be the LTL formulae that resulted in the failure of Tests 1, 2 or 4. (In case Test 3 failed, the only formula involved in the test is simply φ.) The key idea is now to model check the LTL formula φ again in ξ, using an independent implementation of an LTL model checking procedure for this purpose. The result of this check can now be used as a "yardstick" to determine which one of the two Büchi automata is incorrect in each of the cases (i), (ii), (iii) and (iv) above:

(i) If it is confirmed that $\xi \models \varphi$ holds, then the automaton $A_{\neg\varphi}$ constructed for the formula $\neg\varphi$ is incorrect. This is because the automaton $A_{\neg\varphi}$ should accept only those inputs $\eta \in (2^{AP})^\omega$ for which $\eta \not\models \varphi$ is true. Conversely, confirming that $\xi \not\models \varphi$ shows that the translation of φ into a Büchi automaton has failed.

(ii) Confirming that $\xi \models \varphi$ implies that the automaton A_φ must be incorrect, since it erroneously rejects φ (in this case, ξ is a witness *rejected* by both A_φ and $A_{\neg\varphi}$). Likewise, if $\xi \not\models \varphi$, the automaton $A_{\neg\varphi}$ is incorrect.

(iii) If $\xi \models \varphi$, then the automaton rejecting φ is incorrect; if $\xi \not\models \varphi$, the automaton that accepts ξ does not correctly recognize the language $\mathcal{L}_\varphi$.

(iv) Analogous to case (ii).

Instead of using a general LTL model checking procedure for testing whether $\xi \models \varphi$, it is possible to model check the formula in the sequence *directly* using more simple techniques [14, 27], such as a restricted LTL model checking algorithm that operates on infinite sequences over 2^{AP} having a similar finite representation as ξ above. Intuitively, a restricted model checking algorithm is easier to implement correctly than a general algorithm. Therefore, it should be possible to perform the analysis to find the incorrect automaton in a reliable way. This lifts the need for using e.g. another LTL-to-Büchi translator for testing whether $\xi \models \varphi$ is true.

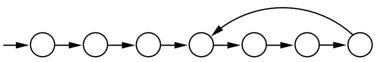


Fig. 5.5: A sequential Kripke structure without state labels

The rest of this section describes an algorithm for model checking the LTL formula φ in ξ. In the algorithm, the formula φ is assumed to consist solely of atomic propositions and the operators $\neg$, $\vee$, X and U. (Since all other LTL operators can be expressed using these basic operators, φ can first be converted into this form if necessary; another straightforward option is to extend the algorithm to support these operators directly.)

Actually, ξ can be considered to be the *temporal interpretation* of an execution of some underlying Kripke structure $M = \langle S, s^0, \rho, \pi \rangle$. The simplest such structure is one in which the states are connected into a sequence. That is, each state of the structure has a *unique* successor (since S is always finite, the successor of the "last" state in the sequence is one of its predecessors in the sequence), and the states in the sequence are labelled with the elements of ξ in the same order as they appear in ξ. (Thus, ξ is the *only* execution of the structure.) In the following, these structures are called *sequential* Kripke structures; see Fig. 5.5 for an example.

Definition 6 A sequential *Kripke structure $\langle S, s^0, \rho, \pi \rangle$ is a Kripke structure whose each state $s \in S$ has exactly one successor, and each state of the structure is reachable from s^0 by zero or more arcs.* ■

Since each state of the structure has exactly one successor, the transition relation ρ is actually a *function*. In this case, the successor of a state $s \in S$ is denoted by $\rho(s)$. For convenience, we also define $\rho^0(s) = s$ and $\rho^{k+1}(s) = \rho(\rho^k(s))$ for any integer $k > 0$.

The algorithm for model checking an LTL formula φ in a sequential Kripke structure M is shown in Fig. 5.6. Intuitively, the algorithm works in a "bottom-up" manner according to the syntactic structure of the formula. Starting from the atomic propositions occurring in φ, the algorithm processes each subformula φ' of φ in turn such that each subformula φ' is processed only after all of its subformulae have been processed. (In practice, this can be done e.g. by processing the subformulae in the postorder imposed by a depth-first search in the parse graph of φ.) For each state s of the structure, the algorithm then determines whether the subformula φ' holds on the (unique) infinite path that begins in s. This is repeated for the other subformulae of φ, until φ itself has been processed.

The algorithm uses a set *Result* for storing the model checking results. At the end of the algorithm, the set will contain a pair (φ, s) if and only if the formula φ holds on the infinite path beginning in the state s. *ToEval* denotes the set of φ's subformulae that have not yet been processed. During each iteration of the main loop (lines 4–35), the algorithm picks a formula from this set and then evaluates it on the paths starting from each state of the structure. The method of evaluating a subformula is determined by the syntactic structure of the formula, so there are five mutually exclusive cases to consider (lines 7–34). In each of these cases, the model checking results for

```
function eval(φ : LtlFormula, M : SequentialKripkeStructure) : Boolean
    Result := ∅;
    ToEval := {φ' | φ' is a subformula of φ};
    while ToEval ≠ ∅ do begin
        φ' := a formula in ToEval such that for all proper subformulae ψ of φ', ψ ∉ ToEval;
        ToEval := ToEval \ {φ'};
        case φ'
            φ' ∈ AP:
                for all s ∈ S do
                    if φ' ∈ π(s) then Result := Result ∪ (φ', s);
            φ' = ¬ψ:
                for all s ∈ S do
                    if (ψ, s) ∉ Result then Result := Result ∪ (φ', s);
            φ' = (ψ1 ∨ ψ2):
                for all s ∈ S do
                    if (ψ1, s) ∈ Result or (ψ2, s) ∈ Result then Result := Result ∪ (φ', s);
            φ' = X ψ:
                for all s ∈ S do
                    if (ψ, ρ(s)) ∈ Result then Result := Result ∪ (φ', s);
            φ' = (ψ1 U ψ2):
                s := s⁰; Marked := ∅;
                for i := 1 to |S| do begin
                    if (ψ2, s) ∈ Result then begin
                        Result := Result ∪ (φ', s);
                        for all s' ∈ Marked do Result := Result ∪ (φ', s');
                        Marked := ∅;
                    end
                    else if (ψ1, s) ∈ Result then Marked := Marked ∪ {s}
                    else Marked := ∅;
                    s := ρ(s);
                end;
                if (φ', s) ∈ Result then
                    for all s' ∈ Marked do Result := Result ∪ (φ', s');
        end;
    end;
    if (φ, s⁰) ∈ Result then return "YES" else return "NO";
end;
```

Fig. 5.6: LTL model checking algorithm for sequential Kripke structures

the formula φ' are computed using previously computed information about its constituent formulae. (In the last case, the set *Marked* is used to keep information about states in which the formula *may* be true.)

The following proposition establishes the correctness of the algorithm.

Proposition 1 (Correctness of the algorithm) *The algorithm of Fig. 5.6 returns the value "YES" if and only if the LTL formula φ holds in the sequential Kripke structure M.*

Proof: See Appendix B. □

As a matter of fact, LTL formulae can be easily translated into CTL formulae such that the LTL formula holds in a *sequential* Kripke structure if and only if the corresponding CTL formula holds in the same structure [14, 27]. The above LTL model checking algorithm for sequential Kripke structures is very similar to a global CTL model checking algorithm (see e.g. [11]) that has been restricted to work only in a certain subclass of Kripke structures. The complexity of the above algorithm can be shown to be $\mathcal{O}(|\varphi| \cdot |S|)$, where $|\varphi|$ denotes the number of symbols in φ.[5]

[5]The main loop of the algorithm is executed at most once for each subformula of φ;

In practice, the set *Result* computed in the algorithm can be used to generate a proof showing whether the formula φ holds in the given sequential Kripke structure. Basically, this can be done by applying LTL semantics to the formula and using the *Result* set to find the truth values of φ's subformulae in the states of the structure. In the analysis of inconsistent Büchi automata, the witness and the proof together show that one of the Büchi automata is incorrect.

The model checking algorithm for sequential Kripke structures provides also an algorithm against which the LTL-to-Büchi translation algorithm implementations can be tested in Test 3. Obviously, this restricts the Kripke structures used in the tests to sequential Kripke structures. However, these are very easy to generate automatically and can be used in the automatic testing of LTL-to-Büchi translators.

---

since $|\{\varphi' \mid \varphi' \text{ is a subformula of } \varphi\}| \leq |\varphi|$, the loop is therefore executed $\mathcal{O}(|\varphi|)$ times. The selection of a subformula from $ToEval$ can be implemented in constant time, e.g., if the subformulae are first inserted in the correct order into a list on line 3 before entering the main loop. The ordering can be done in linear time in the number of subformulae.

It is clear that the loops between lines 9–10, 12–13, 15–16 and 18–19 are of complexity $\mathcal{O}(|S|)$; also the lines 21–33 can be shown to take $\mathcal{O}(|S|)$ time. (This requires noting that the loop between lines 22–31 inserts at most one element into *Marked* in each iteration, and no element is inserted into the set more than once.)

6 EXPERIMENTAL RESULTS

This chapter begins with a description of an automated testbench for LTL-to-Büchi translation algorithm implementations, based on the methods described in Chap. 5. This description is followed by an overview of the arrangements for the tests made with the testbench on several LTL-to-Büchi translation algorithm implementations. The chapter ends with a section presenting the obtained test results with some discussion.

6.1 AUTOMATED TESTBENCH FOR LTL-TO-BÜCHI TRANSLATORS

The test methods presented in Chap. 5 were partially implemented into a testbench for automatically testing LTL-to-Büchi translation algorithm implementations. The testbench includes Tests 1, 3 and 4, using the two latter tests to try to compensate for the missing Test 2, which was left unimplemented. To gather as much data as possible for comparison, the testbench repeats Tests 1 and 3 for all valid combinations of the Büchi automata taking part in the tests, using global synchronous products.[1]

The testbench uses simple randomized algorithms for generating the LTL formulae (used as input for the LTL-to-Büchi translators to be tested) and the Kripke structures (needed in Tests 3 and 4).

The testbench also includes an implementation of the LTL model checking algorithm for sequential Kripke structures. It can optionally be used as another algorithm with which the tested algorithms can be compared in Test 3 by restricting to the use of random sequential Kripke structures. The algorithm can also be used for analyzing an inconsistency detected in Tests 1 or 3 between two implementations, in order to determine which one of the implementations is incorrect.

The testbench was implemented in the C++ programming language. The source code for the program is available through the author's homepage at <URL: `http://www.tcs.hut.fi/%7Ehtauriai/`>.

6.1.1 Testbench Operation

The automatic testing procedure begins with generating a formula φ and a Kripke structure. The tested LTL-to-Büchi translators are then invoked to obtain Büchi automata A^i_φ from the formula. This is repeated also for the negated formula $\neg\varphi$ to obtain the automata $A^i_{\neg\varphi}$ needed in Tests 1 and 4.

Since the input syntax for LTL formulae and the output representation for Büchi automata are usually translator-specific, the testbench uses a separate input/output conversion module for each translator with a unique input/output representation. This allows adding new translators into the tests by attaching an appropriate translation module into the testbench.

Having obtained the Büchi automata, the testbench performs Tests 1, 3

[1] The testbench used here is an extended version of the implementation whose previous versions have been described in [26] and [27]. The most significant extension not included in previous work is the incorporation of Test 1 into the automated testing procedure.

and 4 on the generated automata, using an internal implementation for computing synchronous products and checking them for emptiness. (Test 3 is actually performed twice, using each sets of automata A_φ and $A_{\neg\varphi}$.) This implementation is based on the straightforward techniques described in Sect. 4.2.6. After all tests have been performed, the test procedure is repeated using another LTL formula or Kripke structure.

After each test round, the user can examine the LTL formulae and the Kripke structure used in the test round, together with the Büchi automata generated by the different implementations in that test round. If Tests 1 or 3 detected an inconsistency, the testbench can optionally give a suggestion about which of the tested implementations had failed. This is done by constructing a witness that proves a test failure on some formula φ between some two implementations as described in Sect. 5.2, and then using the LTL model checking algorithm for sequential Kripke structures on the witness to determine which one of the implementations is incorrect. To justify the result, the testbench also gives a proof whether the property φ holds in the witness.

After a predetermined number of test rounds, the testbench finally reports the number of different types of failures detected in the tests between each pair of implementations. Due to implementation errors in an LTL-to-Büchi translator, it may well occur that the translator fails to produce *any* acceptable output on some input formulae; also these kinds of failures are reported for each implementation taking part in the tests.

6.1.2 Generating Input for the Tests

The testbench uses simple randomized algorithms for generating LTL formulae and Kripke structures to be used as input for the automated test procedure. The main goal in designing the algorithms was to obtain simple procedures that generate output that satisfies simple structural requirements. No formal analysis was specifically performed in the design of the algorithms in order to make them satisfy any explicit requirements regarding the output distribution. Therefore, it is very likely that the produced output is biased according to any formal criteria that might be considered (e.g., that the output of the algorithms should be "uniformly distributed" according to some notion of uniformity).

The behaviour of the algorithms can be adjusted with several parameters. This enables having some "intuitive" control over the expected properties of the generated LTL formulae and the Kripke structures, even though the exact distributions remain unknown. Actually, some of the parameters can be adjusted such that the algorithms indeed generate "uniform" output according to some explicit criteria. For example, see Appendix C for the analysis that was done for adjusting the random formula generation parameters in the experiments described later in this chapter.

LTL formulae. The testbench generates random LTL formulae using a straightforward recursive algorithm [26, 27, 5]. The pseudocode for this algorithm is shown in Fig. 6.1. The algorithm generates formulae with a parse tree having a given number of nodes. The algorithm first chooses a logical

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

Fig. 6.1: Pseudocode for the formula generation algorithm [27]

or a temporal operator, recursively constructs one or two smaller formulae according to the arity of the operator, and finally concatenates the formulae with the chosen operator into a single formula. At the leaves of the parse tree (i.e., when generating a subformula with a parse tree of size 1), the algorithm selects either an atomic proposition from a given set of propositions AP or a Boolean constant $\top$ or $\bot$ as the formula. In the algorithm of Fig. 6.1, n denotes the size of the parse tree of the formula.

The full set of operators supported by the testbench implementation consists of the unary operators $\{\neg, \mathsf{X}, \Box, \Diamond\}$ and the binary operators $\{\vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$. In the testbench implementation, the probability of selecting each individual operator into the generated formula can be controlled by specifying a “priority” for each individual operator as a nonnegative integer. These priorities can be adjusted to disable the use of some operator or operators altogether, for example, if one of the tested LTL-to-Büchi translators does not support all the available operators directly. Let OP be the set of operators from which the algorithm chooses a random operator at some point in the execution of the algorithm, and let $op \in OP$. Denote by $pri(op) \geq 0$ the priority given for op. Then, the probability of selecting the operator into the formula at that point in the execution is simply given by $pri(op)/\sum_{op' \in OP} pri(op')$. (Of course, $pri(op')$ must be positive for at least one operator $op' \in OP$ for the probability to be defined.) While this method for choosing the operators is easy to implement, it can easily be seen to favour unary operators, since they are available for selection in two separate places of the algorithm (lines 7 and 12). However, it is still possible to adjust the operator priorities so that each generated formula will have the same expected number of each operator in it; this was the criterion used in the experiments presented later in this chapter. (See Appendix C.)

Kripke structures. In this work, the transition relation for Kripke structures is always assumed to be total (Definition 1, page 4). This must be taken into account in the algorithms for generating random Kripke structures, so they must ensure that every state of each generated structure has at least one successor. In the following, this is referred to as the "successor constraint".

The simple graph construction algorithms used in the testbench all generate Kripke structures with a given number of states n. The valuations for the atomic propositions are defined in each state by choosing the truth value of each proposition $p \in AP$ randomly from the two possibilities. Each proposition is given the value "true" with a given probability t. The algorithms also make use of a parameter d (approximating the "density" of the graph, i.e., the probability of having an arc between any two nodes; the parameter does not, however, affect the arcs that must be added between states to enforce the successor constraint). Three different types of graphs can be used:

1. Random graphs. These are generated by simply taking each state of the structure in turn and adding a random transition between that state and any other state with the given probability d. If the resulting graph does not satisfy the successor constraint, each state violating the constraint is then connected to some randomly selected state of the structure.

2. Random connected graphs. These are random graphs satisfying the successor constraint with the additional requirement that each state of the structure should be reachable from some designated "initial state" of the structure by zero or more arcs. (The intuition behind such a requirement is that the structure can then be thought of as "simulating" the reachable part of the state space of some system.)

   The pseudocode for this algorithm is shown in Fig. 6.2. The algorithm uses s_0 as the initial state of the Kripke structure. The set *UnreachableNodes* keeps track of the states which cannot yet be reached from s_0 in the graph. The set *NodesToProcess* contains the states that are known to be reachable from s_0 but have not yet been processed itself. Initially, the only such state is the initial state s_0.

   In each iteration of the outermost loop of the algorithm, the algorithm chooses some previously unvisited state s known to be reachable from s_0 (lines 7–8) and then defines the valuation for the atomic propositions in that state (lines 9–12). After this, s is connected to some yet unreachable state s' (if such a state exists), making s' now ready to be eventually visited itself (lines 13–18). Then, the algorithm adds random edges from s to other states of the structure with the given probability d (lines 19–26). (This may cause some yet unreachable states to become reachable from s_0, so the sets *UnreachableNodes* and *NodesToProcess* must be updated accordingly.) Finally, if s still has no successors, it is simply connected to itself to maintain the successor constraint (lines 27–28). (The path from s_0 to s in the structure can in this case be seen as a terminating execution of the "system" corresponding to the structure.)

3. Random sequential structures. These structures simply consist of the states of the structure arranged into a sequence with a back edge added

```
function RandomGraph(n : Integer, d : Real ∈ [0.0, 1.0], t : Real ∈ [0.0, 1.0])
            : KripkeStructure
    S := {s0, s1, ... , s(n−1)};
    NodesToProcess := {s0};
    UnreachableNodes := {s1, s2, ... , s(n−1)};
    ρ := ∅;
    while NodesToProcess ≠ ∅ do begin
        s := a random node in NodesToProcess;
        NodesToProcess := NodesToProcess \ {s};
        π(s) := ∅;
        for all P ∈ AP do
            if RandomNumber(0.0, 1.0) < t then
                π(s) := π(s) ∪ {P};
        if UnreachableNodes ≠ ∅ then begin
            s′ := a random node in UnreachableNodes;
            UnreachableNodes := UnreachableNodes \ {s′};
            NodesToProcess := NodesToProcess ∪ {s′};
            ρ := ρ ∪ {(s, s′)};
        end;
        for all s′ ∈ S do
            if RandomNumber(0.0, 1.0) < d then begin
                ρ := ρ ∪ {(s, s′)};
                if s′ ∈ UnreachableNodes then begin
                    UnreachableNodes := UnreachableNodes \ {s′};
                    NodesToProcess := NodesToProcess ∪ {s′};
                end;
            end;
        if there is no edge (s, s′) in ρ for any s′ ∈ S then
            ρ := ρ ∪ (s, s);
    end;
    return ⟨S, ρ, s0, π⟩;
end;
```

Fig. 6.2: Pseudocode for the Kripke structure generation algorithm [27]

from the "last" state in the sequence to some randomly selected previous state in the sequence (see Fig. 5.5). The parameter d is not used in this case, since each state always has exactly one successor.

As mentioned previously, using sequential Kripke structures as input for the test procedure allows comparing the model checking results obtained using the LTL-to-Büchi translators in Test 3 with the results given by the restricted LTL model checking algorithm of Sect. 5.2. This testing will be enabled automatically in the testbench whenever using sequential structures.

6.2 TEST ARRANGEMENTS

The experiments in this work were made by running the automated testing procedure on several available LTL-to-Büchi translation algorithm implementations. The implementations taking part in the tests were:

ÅSA+ The ÅSA+ implementation is an LTL-to-Büchi translator derived from Mauno Rönkkö's C++ class library [22] implementing the translation algorithm presented in [8]. The class library is also a part of the ÅSA model checking package [17]. The class library was rewritten to make use of the containers of the C++ Standard Template Library (STL), including some of the other code optimizations proposed in [22]. The library also had to be extended with code for computing the acceptance conditions of the generated automata. In addition, direct rules were implemented for the operators $\rightarrow$, $\leftrightarrow$, $\Box$ and $\Diamond$ that were handled by rewriting rules in the original implementation.

SPIN 3.x.x The model checker SPIN [10] by Gerard J. Holzmann includes a module for automatically converting LTL formulae into "never claims", which are basically Büchi automata encoded in SPIN's modelling language PROMELA. Also this implementation is originally based on the algorithm in [8], but it includes several optimizations (some of which are described in [6]).

The automated testing procedure has been used on this implementation since its version 3.3.3. The testing has uncovered some implementation errors in various versions of the tool [26, 27]. In this work, the behaviour of version 3.3.3 (July 1999; the first version to be ever tested with some of the methods presented in this work) was compared with versions 3.3.9 (January 2000; a version with some corrections to the LTL-to-Büchi translation module, incorporating feedback given on errors found using the testing procedure) and 3.4.1 (August 2000; the latest version at the time of writing) to see how the behaviour of the implementation has changed between the different versions.

LTL2AUT LTL2AUT is the LTL-to-Büchi translator written by the authors of [5]. It is based on a translation algorithm presented in the same paper. The implementation actually contains three different algorithms: the "GPVW" algorithm [8] (the same algorithm on which the previous two implementations were based), the "GPVW+" algorithm based on some improvements proposed already in [8], and the "LTL2AUT" algorithm of [5] itself.

In this work, all algorithms included in the LTL2AUT implementation were tested. In the experiments, these are referred to as LTL2AUT(GPVW), LTL2AUT(GPVW+) and LTL2AUT(LTL2AUT), respectively.

PROD The Pr/T net reachability analyzer PROD [33, 34] includes an LTL-to-Büchi translator module based on the algorithm presented in [31]. This implementation was also included in the tests made in this work. The version used was from 27 July 2000.

The tests were divided into several batches according to the number of nodes in the parse tree of the generated LTL formulae, in order to (roughly) see how the size of the Büchi automata generated by each translator depends on the input formula size in practice. Each batch consisted of 1,000 LTL formulae with a fixed parse tree size. There were a total of eight batches, each of which consisted of 1,000 randomly generated LTL formulae with a parse tree of 5, 6, 7, 8, 9, 10, 11 and 12 nodes, respectively.

The operators in the formulae were chosen from a set of operators *directly* supported by all the tested translation algorithm implementations. Even though formulae including unsupported operators could in some cases have been rewritten using more primitive operators before giving the formula to a translator, this was not done, since applying rewriting rules to a formula can change the size of the parse tree of the formula. This would have resulted in test batches with formulae of varying parse tree size, which would in turn have made it more complicated to investigate the relationship between the sizes of the formulae and the generated Büchi automata.[2] Unfortunately, restricting to operators directly supported by all the tested implementations left some operators unused even though some implementations would have been able to accept them. As a matter of fact, all the above implementations except PROD supported exactly the same operators as the testbench (the unary operators $\{\neg, \mathsf{X}, \Box, \Diamond\}$ and the binary operators $\{\vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$) directly. The PROD tool, however, lacked direct support for the X, $\leftrightarrow$ and R operators, leaving only the operators $\neg, \Box, \Diamond, \vee, \wedge, \rightarrow$ and U to be used for all the implementations. For this reason, the test procedure was repeated on the other implementations, this time allowing the full set of available operators to be used when generating the formulae.

A further requirement adopted for generating the input formulae was that the formulae in each batch should contain the *same expected number* of each available operator.[3] The details on how this was achieved in practice can be found in Appendix C, which contains an analysis of the random formula generation algorithm shown in Fig. 6.1.

The set AP was in all test batches fixed to five propositions, with each individual proposition having the probability of 0.18 of being chosen by the formula generation algorithm of Fig. 6.1 each time line 3 is executed. Each of the Boolean constants had the probability of 0.05 of being selected.

The Kripke structures for Tests 3 and 4 were generated using the random connected graph algorithm (Fig. 6.2). Each graph consisted of 50 states, and the value 0.1 was used for the approximate graph density d. Each proposition was equally likely to get assigned either of the values "true" and "false" in each state (i.e., the value 0.5 was used for the parameter t). A new Kripke structure was generated whenever a new LTL formula was generated, so 1,000 structures were used in each test batch.

Each of the tested implementations listed above was connected into the testbench with a separate input/output conversion module. All of the tested translation algorithm implementations and the testbench itself were com-

[2]It may well be argued that an implementation might still in practice "change" the size of the input formula using e.g. some simplification rules before translating the formula into a Büchi automaton. However, since the implementations are treated as "black boxes" in this work, such implementation-dependent issues are irrelevant to the simple testing strategy used here. It must, nevertheless, be recognized that such internal details of a translator can have a significant effect on the size of the automata generated by the implementation, which is likely to result in much variation in the sizes of the automata, even though seemingly "fixed-size" formulae are used as input.

[3]The only reason for selecting this strategy in this work was to have some knowledge about the distribution of the generated formulae, instead of e.g. simply assigning arbitrary priorities to the different operators. Clearly, the chosen strategy is biased in comparison to some other intuitively reasonable criteria that might be considered, e.g. that every formula of a given parse tree size should be "equally likely" to be generated.

(However, assuming that the implementations perform some operator-based case analysis—in the manner of the model checking algorithm of Fig. 5.6, for example—the chosen formula generation strategy might help in trying to exercise each case equally often on the average.)

piled from C or C++ sources with version 2.95.2 of the GNU Compiler Collection (gcc). The tests were run in Debian Linux 2.1 environment on Pentium II/III PCs with 256 MB of memory. In the experiments, each translation algorithm implementation was given 128 MB of memory space: the formula translation was interrupted in case this memory limit was exceeded. (This creates another source of automaton generation failures that must be distinguished from those failures in which an implementation fails due to some other reason.) The testbench itself was given all available memory space for performing the various tests; unfortunately, this was not always enough to perform all tests on some large automata; see the discussion in the next section.

Finally, a different set of tests were run on the implementations supporting the "full" set of operators. The goal of these tests was to try to see whether the observed test failure rates in Test 3 (the model checking result cross-comparison check) seem to have any dependency on the approximate density d of the Kripke structures used in the tests, while keeping all other parameters (Kripke structure size, number of atomic propositions) fixed. Such correlation might give information on how to "best" choose the value for the parameter d in order to maximize the odds of finding errors in LTL-to-Büchi translators using Test 3 (at least in the used value combination for the other parameters).

The above tests correspond to repeating the original experiment (with all formula parse tree sizes 5–12) on all algorithms except PROD using different values for the parameter d. The value 0.1 was already tested above; the values used for d in the new tests were 0.0, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 and 1.0. (If $d = 0.0$, the algorithm of Fig. 6.2 will generate tree-like structures whose each "leaf node" is connected to itself; if $d = 1.0$, there will be an edge between each pair of states.) In addition, the experiment was still repeated using *sequential Kripke structures* instead of random connected graphs to see whether this causes any significant change in the observed failure rate.

6.3 TEST RESULTS

Tables 6.3 and 6.4 show the numbers of times each of the tested implementations failed to generate a Büchi automaton from an LTL formula, including the total numbers of failures and generated automata. Here, a failure means any reason that prevented a translator from producing valid output from some input formula. Based on the test results, the failures could be categorized into the following types:

(1) The translator process was terminated due to a fatal internal error (e.g., a segmentation fault).

(2) An internal assertion violation occurred in the translator.

(3) The translator produced syntactically incorrect output according to its output format specification.

(4) The translator process exceeded the memory limit of 128 MB.

Table 6.3: Büchi automaton generation failures with operators $\{\neg, \Box, \Diamond, \vee, \wedge, \rightarrow, \mathsf{U}\}$

Parse tree size of random formulae	Number of automaton generation failures <number of failures>(<number of failures due to memory exhaustion>) [2000 attempts]							
	ÅSA+	SPIN v3.3.3	SPIN v3.3.9	SPIN v3.4.1	LTL2AUT (GPVW)	LTL2AUT (GPVW+)	LTL2AUT (LTL2AUT)	PROD
5...10	0	0	0	0	0	0	0	0
11	0	**2** (2)	0	0	0	0	0	**2** (2)
12	0	**3** (3)	0	**2** (2)	0	0	0	**4** (4)
TOTAL	0	**5** (5)	0	**2** (2)	0	0	0	**6** (6)
Number of automata	16000	15995	16000	15998	16000	16000	16000	15994

Table 6.4: Büchi automaton generation failures with operators $\{\neg, \mathsf{X}, \Box, \Diamond, \vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$

Parse tree size of random formulae	Number of automaton generation failures <number of failures>(<number of failures due to memory exhaustion>) [2000 attempts]						
	ÅSA+	SPIN v3.3.3	SPIN v3.3.9	SPIN v3.4.1	LTL2AUT (GPVW)	LTL2AUT (GPVW+)	LTL2AUT (LTL2AUT)
5	0	**100** (0)	0	0	0	0	0
6	0	**137** (0)	0	0	0	0	0
7	0	**172** (0)	0	0	0	0	0
8	0	**185** (0)	**1** (1)	**1** (1)	0	0	0
9	0	**219** (0)	**2** (2)	**2** (2)	0	0	0
10	0	**235** (1)	**4** (4)	**5** (5)	0	0	0
11	0	**215** (1)	**8** (7)	**10** (10)	0	0	0
12	0	**302** (0)	**22** (21)	**18** (18)	0	0	0
TOTAL	0	**1565** (2)	**37** (35)	**36** (36)	0	0	0
Number of automata	16000	14435	15963	15964	16000	16000	16000

Since the fourth type of failure is not (necessarily) due to errors in an implementation itself, each cell with a nonzero failure rate in Tables 6.3 and 6.4 includes also the number of failures that were actually due to memory exhaustion.

The failure rates shown in the tables are over *all* formulae given to the implementations, including both the randomly generated formulae with the shown parse tree size and the *negated* formulae (whose parse trees have one additional node) that were needed to run Tests 1 and 4 on the implementations. Thus, each cell in the tables corresponds to the number of failures among a set of 2,000 formulae.

In these experiments, PROD and the various versions of SPIN seemed to require more memory than the other implementations, which is seen as an increasing number of failures due to memory exhaustion on these implementations as the parse tree size of the formulae increases. The other implementations were able to operate in 128 MB of available memory space.

Table 6.3 (with test results obtained using a restricted set of formula operators) does not reveal errors in any of the implementations, since all failures in this table are due to memory exhaustion. However, allowing the use of a larger set of operators in the tests caused some versions of SPIN (3.3.3 and 3.3.9) to behave more unreliably, resulting in errors of types (1) to (3) described above. SPIN v3.3.3 suffered from all these types of errors; version 3.3.9 failed only due to internal assertion violations.

Test 1 and Test 3 Results

The results of Test 1 and Test 3 are reported in Tables 6.5 and 6.6 for the two formula symbol sets $\{\neg, \Box, \Diamond, \vee, \wedge, \rightarrow, \mathsf{U}\}$ and $\{\neg, \mathsf{X}, \Box, \Diamond, \vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$, respectively. An important observation was that the ÅSA+, LTL2AUT (GPVW), LTL2AUT (GPVW+), LTL2AUT (LTL2AUT) and PROD implementations *never* failed any of these tests against each other, regardless of the formula symbol set. For this reason, the tables combine the results for these algorithms together, and these algorithms are collectively referred to as "Å/L/P" in the tables.[4] The independence of these three implementations, together with the observation that also SPIN seems to "converge" towards the same results as the tool version number increases, gives a strong suggestion that these implementations are quite reliable and correct.

Each cell in the top part of each table contains a triple of integers $a/b/c$ representing the following information:

- a is the number of failed Büchi automata intersection emptiness checks between two supposedly complementary Büchi automata (Test 1). Each nondiagonal element in the matrix associated with some formula parse tree size corresponds to testing two different implementations against each other, so the maximum number of tests performed is 2,000

[4]There were slight differences in the failure rates when testing these implementations against various versions of SPIN. This is a consequence of Büchi automaton generation failures that sometimes occurred on some of these translators, preventing some of the tests from being performed. The number reported in the tables is always the *minimum* failure rate obtained using these implementations (i.e., some of the "Å/L/P" implementations may actually have had a slightly higher failure rate against SPIN by themselves than the rate shown in the table).

Table 6.5: Failure rates for Tests 1 and 3 with operators $\{\neg, \Box, \Diamond, \vee, \wedge, \rightarrow, \mathsf{U}\}$

Parse tree size of random formulae	Implementation	Number of test failures <Test 1 failures> / <Test 3 failures (local failures)> / <Total number of inconsistent automata detected> [Diagonal cells: max. 1000 tests; other cells: max. 2000 tests]			
		Å/L/P	Spin v3.3.3	Spin v3.3.9	Spin v3.4.1
5	Å/L/P	0 / 0 / 0			
	S3.3.3	21 / 4(3) / 22	20 / – / 20		
	S3.3.9	0 / 0 / 0	21 / 4(3) / 22	0 / – / 0	
	S3.4.1	0 / 0 / 0	21 / 4(3) / 22	0 / 0 / 0	0 / – / 0
6	Å/L/P	0 / 0 / 0			
	S3.3.3	32 / 8(6) / 33	31 / – / 31		
	S3.3.9	0 / 0 / 0	32 / 8(6) / 33	0 / – / 0	
	S3.4.1	0 / 0 / 0	32 / 8(6) / 33	0 / 0 / 0	0 / – / 0
7	Å/L/P	0 / 0 / 0			
	S3.3.3	49 / 9(7) / 50	48 / – / 48		
	S3.3.9	0 / 0 / 0	49 / 9(7) / 50	0 / – / 0	
	S3.4.1	0 / 0 / 0	49 / 9(7) / 50	0 / 0 / 0	0 / – / 0
8	Å/L/P	0 / 0 / 0			
	S3.3.3	66 / 14(12) / 69	61 / – / 61		
	S3.3.9	0 / 1(1) / 1	65 / 13(11) / 68	0 / – / 0	
	S3.4.1	0 / 0 / 0	66 / 14(12) / 69	0 / 1(1) / 1	0 / – / 0
9	Å/L/P	0 / 0 / 0			
	S3.3.3	69 / 16(8) / 70	69 / – / 69		
	S3.3.9	0 / 0 / 0	69 / 16(8) / 70	0 / – / 0	
	S3.4.1	0 / 0 / 0	69 / 16(8) / 70	0 / 0 / 0	0 / – / 0
10	Å/L/P	0 / 0 / 0			
	S3.3.3	73 / 19(13) / 75	66 / – / 66		
	S3.3.9	0 / 1(1) / 1	71 / 18(12) / 74	0 / – / 0	
	S3.4.1	0 / 0 / 0	73 / 19(13) / 75	0 / 1(1) / 1	0 / – / 0
11	Å/L/P	0 / 0 / 0			
	S3.3.3	86 / 19(8) / 88	83 / – / 83		
	S3.3.9	0 / 2(1) / 2	87 / 17(7) / 87	0 / – / 0	
	S3.4.1	0 / 0 / 0	87 / 19(8) / 89	0 / 2(1) / 2	0 / – / 0
12	Å/L/P	0 / 0 / 0			
	S3.3.3	101 / 31(17) / 110	91 / – / 91		
	S3.3.9	0 / 3(2) / 3	102 / 32(19) / 112	0 / – / 0	
	S3.4.1	0 / 1(0) / 1	102 / 30(17) / 110	0 / 2(2) / 2	0 / – / 0

	Total number of tests performed <Test 1> / <Test 3> [Diagonal cells: max. 8000 tests; other cells: max. 16000 tests]			
Å/L/P	Å↔Å, L↔L: 8000 / – ; P↔P: 7767 / – ; Å↔L: 16000 / 16000 ; {Å,L}↔P: 15907 / 15966			
S3.3.3	15933 / 15961	7995 / –		
S3.3.9	15950 / 15966	15995 / 15995	8000 / –	
S3.4.1	15949 / 15964	15993 / 15993	15998 / 15998	7998 / –

Table 6.6: Failure rates for Tests 1 and 3 with operators $\{\neg, \mathsf{X}, \Box, \Diamond, \vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$

Parse tree size of random formulae	Implementation	Number of test failures <Test 1 failures> / <Test 3 failures (local failures)> / <Total number of inconsistent automata detected> [Diagonal cells: max. 1000 tests; other cells: max. 2000 tests]			
		Å/L	Spin v3.3.3	Spin v3.3.9	Spin v3.4.1
5	Å/L	0 / 0 / 0			
	S3.3.3	215/270(108)/299	35 / – / 35		
	S3.3.9	17 / 15(8) / 17	227/263(102)/311	15 / – / 15	
	S3.4.1	0 / 0 / 0	215/270(108)/299	17 / 15(8) / 15	0 / – / 0
6	Å/L	0 / 0 / 0			
	S3.3.3	244/294(103)/331	26 / – / 26		
	S3.3.9	11 / 9(6) / 11	253/290(101)/340	10 / – / 10	
	S3.4.1	0 / 0 / 0	244/294(103)/331	11 / 9(6) / 11	0 / – / 0
7	Å/L	0 / 0 / 0			
	S3.3.3	297/338(130)/400	45 / – / 45		
	S3.3.9	17 / 14(9) / 17	311/334(131)/414	15 / – / 15	
	S3.4.1	0 / 0 / 0	297/338(130)/400	17 / 14(9) / 17	0 / – / 0
8	Å/L	0 / 0 / 0			
	S3.3.3	321/351(134)/434	46 / – / 46		
	S3.3.9	24 / 21(10) / 24	329/339(127)/441	18 / – / 18	
	S3.4.1	0 / 0 / 0	320/350(133)/434	24 / 21(10) / 24	0 / – / 0
9	Å/L	0 / 0 / 0			
	S3.3.3	358/374(124)/478	51 / – / 51		
	S3.3.9	30 / 28(19) / 30	376/366(117)/493	23 / – / 23	
	S3.4.1	0 / 0 / 0	357/373(123)/476	30 / 28(19) / 30	0 / – / 0
10	Å/L	0 / 0 / 0			
	S3.3.3	409/408(129)/527	46 / – / 46		
	S3.3.9	16 / 13(6) / 16	419/406(127)/536	15 / – / 15	
	S3.4.1	0 / 0 / 0	407/407(129)/524	16 / 13(6) / 16	0 / – / 0
11	Å/L	0 / 0 / 0			
	S3.3.3	470/452(137)/598	64 / – / 64		
	S3.3.9	31 / 25(9) / 31	477/435(128)/607	22 / – / 22	
	S3.4.1	0 / 0 / 0	462/445(133)/592	30 / 24(8) / 29	0 / – / 0
12	Å/L	0 / 0 / 0			
	S3.3.3	506/466(150)/622	64 / – / 64		
	S3.3.9	35 / 27(12) / 35	512/444(139)/629	26 / – / 26	
	S3.4.1	0 / 0 / 0	496/458(148)/615	35 / 27(12) / 35	0 / – / 0

	Total number of tests performed <Test 1> / <Test 3> [Diagonal cells: max. 8000 tests; other cells: max. 16000 tests]			
Å/L	Å↔Å, L↔L: 8000 / – ; Å↔L: 16000 / 16000			
S3.3.3	14435 / 14435	7139 / –		
S3.3.9	15963 / 15963	14410 / 14411	7963 / –	
S3.4.1	15964 / 15964	14411 / 14412	15928 / 15957	7965 / –

in this case. The diagonal cells correspond to testing an implementation against itself; the maximum number of tests for these cells is 1,000, since intersecting two Büchi automata generated by the same implementation twice does not give any new information. (However, although not shown in the table, the "combined" Å/L/P implementations were tested 2,000 times against each other, and no inconsistencies were detected.)

- b is the number of failed model checking result cross-comparison checks between two implementations (Test 3). Each cell again corresponds to at most 2,000 performed tests; in addition, since no implementation can fail this test against itself, the diagonal cells are not relevant in this case. The zeros in the diagonal cells corresponding to the "combined" Å/L/P (Table 6.5) or Å/L (Table 6.6) implementation are only intended to emphasize that there were no inconsistencies between these implementations.

  The number in boldface gives the failure rate when the results were compared in each state of the Kripke structure (corresponding to the "global" emptiness check described in Sect. 5.1.2). The number in parentheses gives the failure rate when the results were compared only locally with respect to a single "initial" state of a Kripke structure.

- c gives the total number of automata (out of at most 2,000 automata involved in the tests) that were determined to be incorrect by either of the two above tests. This number can be at most the sum of a and b; usually, it is considerably smaller, since an incorrect automaton may well fail both of the above tests.

The lower parts of Tables 6.5 and 6.6 report the total numbers of each type of test performed between any two implementations. The differences in the numbers of tests performed is both due to the implementations' occasional failures to generate Büchi automata and *unsuccessful tests caused by the testbench itself running out of memory*. This sometimes occurred with large Büchi automata that could not be synchronized with the random Kripke structures (or other Büchi automata when performing Test 1) in the memory space available to the testbench (ca. 300 MB). (These cases were not counted in the reported failure rates, so the failure rates correspond correctly to the actual numbers of detected inconsistencies.)

The growth in the number of observed test failures can be seen to (roughly) follow the increase in formula parse tree size. Comparing the two tables with each other, it can be seen that a larger variety of operators in the randomly generated LTL formulae increased the number of observed test failures. The results reveal errors in SPIN versions 3.3.3 and 3.3.9, since these implementations sometimes failed Test 1 by themselves (however, version 3.3.9 failed only in the tests performed using all available operators). SPIN v3.4.1 always passed this test also against the ÅSA+, LTL2AUT (all variants) and PROD implementations. However, Table 6.5 reveals one inconsistency in the result cross-comparison check between SPIN v3.4.1 and the other implementations (12 nodes in the formula parse tree). Analyzing this case separately with the testbench (i.e., automatically constructing a witness

Table 6.7: Failure rates for Test 4 on SPIN versions 3.3.3 and 3.3.9 (max. 1000 tests)

Parse tree size of random formulae	Operator set used $\{\neg, \Box, \Diamond, \vee, \wedge, \rightarrow, \mathsf{U}\}$		Operator set used $\{\neg, \mathsf{X}, \Box, \Diamond, \vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$	
	SPIN v3.3.3	SPIN v3.3.9	SPIN v3.3.3	SPIN v3.3.9
5	0	0	**1**	0
6	0	0	**6**	0
7	0	0	**2**	0
8	0	**1**	**5**	0
9	0	0	**5**	0
10	0	**1**	**4**	0
11	0	**1**	**6**	0
12	0	**2**	**3**	0
TOTAL	0	**5**	**32**	0
Total number of tests performed	7995	8000	7139	7963

and then analyzing it with the LTL model checking algorithm for sequential Kripke structures, as described in Sect. 5.2) confirmed the incorrectness of the Büchi automaton generated by SPIN v3.4.1, revealing an error in the implementation. This analysis can be found in Appendix D.

Tests 1 and 3 often revealed the "same" errors in the automata. This can be seen in that the total number of automata that were determined to be incorrect by these two tests is usually far less than the sum of the failure rates of the individual tests. These tests can still be considered useful together, since Test 3 is may detect inconsistencies that are impossible to find using only Test 1.

Test 4 Results

Test 4 (the model checking result consistency check) failed occasionally on SPIN versions 3.3.3 and 3.3.9. All other implementations passed this test whenever it could be performed. The failure rates for these two implementations on each set of formula symbols are shown in Table 6.7. (In this case, each cell of the table corresponds to the number of failures in a maximum of 1,000 tests.)

As can be seen in the table, the failure rates are relatively small in comparison to the failure rates in Tests 1 and 3. In addition, these tests do not reveal any clear dependence between the formula size and the number of consistency check failures. However, the formula symbols used in the different tests seem to have a peculiar effect on the failure rates: when using only the smaller operator set, only SPIN v3.3.9 ever failed; the situation was exactly the opposite when using the larger set of operators. This might suggest that the errors in SPIN v3.3.3 may be related to the use of operators missing in the smaller set of operators, while the errors in SPIN v3.3.9 are related only to the common operators included in both sets. Intuitively, these errors should be less likely to surface when using a larger set of operators, which might offer some explanation to why no errors were detected in SPIN v3.3.9 in the tests with a larger set of operators.

Comparing only the magnitudes of the failure rates observed in Test 4 and Test 1 (performed on a single implementation), Test 4 seems to be less efficient. However, since these two tests in fact apply to *different* kinds of errors (see Fig. 5.2 and the discussion in Sect. 5.1.2), they really complement each other. For example, even though Test 1 did not reveal any inconsistencies in SPIN v3.3.9 in the tests with the smaller set of operators, Test 4 failed on this implementation several times; the same phenomenon occurred reversed with the same implementation using the larger operator set. (Of course, since Test 4 is only an "approximation" of the unimplemented Test 2, the number of detected errors is likely to remain quite small in comparison to e.g. the failure rate observed in Test 1.)

Test 3 and the Approximate Density of Kripke Structures

As mentioned in the end of the previous section, the experiment was repeated with Kripke structures having a different approximate density than 0.1, in order to see whether the failure rate in Test 3 seems to depend on the value of this test parameter in any systematic way. The experiment was also repeated using sequential Kripke structures. (The density can have no effect on the failure rate observed in Test 1, because this test is based on the direct analysis of Büchi automata instead of the LTL model checking procedure.)

Figure 6.8 shows the observed failure rates in Test 3 between the ÅSA+ and SPIN v3.3.3 implementations for different values for the approximate density d and for the formula parse tree size n. Figure 6.9 repeats the results for the SPIN v3.3.9 implementation. (The figures use ÅSA+ as a reference because the previous tests gave a strong suggestion about its correctness. The failure rates between other combinations of implementations behaved similarly.) In the diagrams, each point corresponds to the observed failure rate on 2,000 randomly generated LTL formulae for a fixed value of d; "`seq.`" corresponds to the failure rate obtained using sequential Kripke structures.

As could be expected, the failure rates seem to slightly increase along with the formula size. The diagrams do not, however, help in concluding much about the dependency between the approximate density d and the observed failure rates, since there is so much fluctuation between the failure rates (especially with SPIN v3.3.9, whose failure rates are extremely small, just around 1 %).

However, in many diagrams the failure rate can be seen to drop as the graph density increases, at least around the smallest values of d. In addition, in all diagrams the failure rate obtained with sequential Kripke structures (the rightmost data point of each diagram) is quite large. Although not generally shown by the diagrams, there might be one intuitive argument supporting a hypothesis that the failure rates should decrease as d increases (i.e., as there is more *branching* in the generated Kripke structures). Namely, the more transitions there are between different states of the structure, the more paths (or executions) there are in the structure. Furthermore, the more paths there are, the more likely it is that one of them is accepted by a (nonempty) Büchi automaton, regardless of whether the automaton correctly corresponds to some LTL formula or not (since the truth values for the atomic propositions were chosen at random in each state). This would then imply that Test 3 would

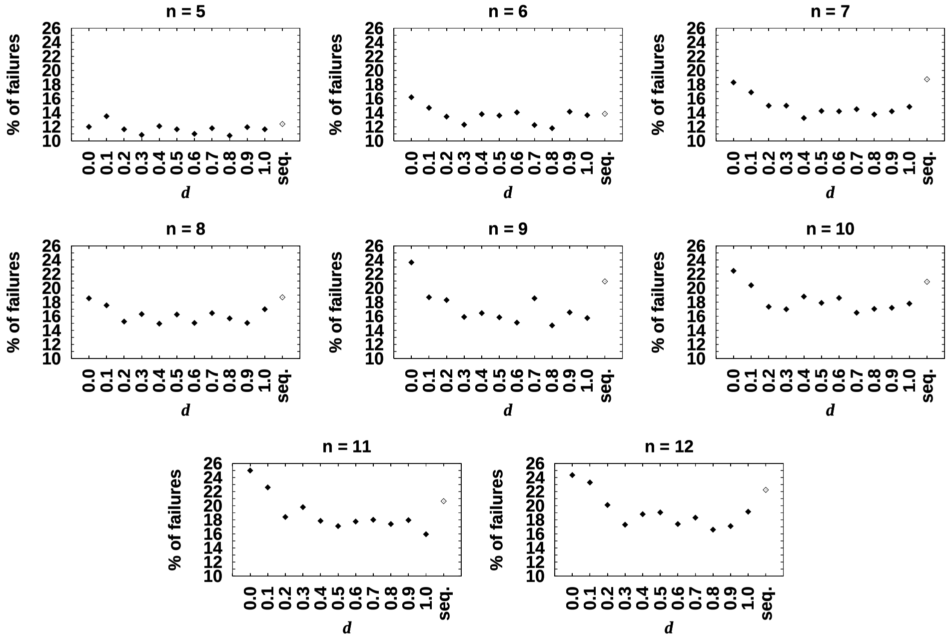


Fig. 6.8: Test 3 failure rates (ÅSA+ ↔ SPIN v3.3.3)

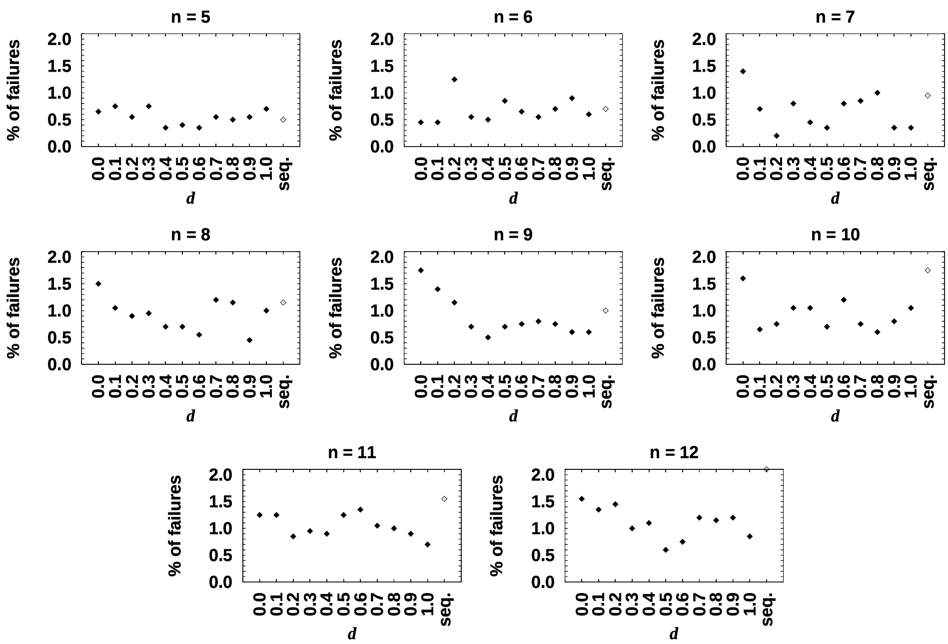


Fig. 6.9: Test 3 failure rates (ÅSA+ ↔ SPIN v3.3.9)

Table 6.10: Average sizes of successfully generated Büchi automata (number of states / number of transitions); operators $\{\neg, \Box, \Diamond, \vee, \wedge, \rightarrow, \mathsf{U}\}$)

Parse tree size of random formulae	ÅSA+	SPIN v3.3.3	SPIN v3.3.9	SPIN v3.4.1	LTL2AUT (GPVW)	LTL2AUT (GPVW+)	LTL2AUT (LTL2AUT)	PROD
5	5/8	2/4	2/3	2/3	5/8	5/7	4/6	10/62
6	6/11	3/5	3/4	3/4	6/11	5/9	5/8	16/144
7	8/15	4/6	3/5	3/5	7/14	6/12	5/10	25/302
8	9/19	4/7	3/6	3/6	8/17	7/14	6/11	38/712
9	10/26	4/10	4/7	3/6	10/25	8/18	6/13	55/1718
10	12/33	5/12	4/9	4/8	11/29	9/22	7/15	81/3101
11	15/54	6/17	5/10	4/9	14/48	11/31	8/20	125/6292
12	17/67	6/18	5/13	4/11	15/59	12/37	8/22	183/9414

succeed more often. The high observed failure rates for sequential Kripke structures would be consistent with this hypothesis, since there is only trivial branching in a sequential Kripke structure.

It is clear that there are very many parameters, all of which might affect the number of observed test failures. These include the size of the Kripke structures and the methods with they were generated, together with even the internal behaviour of the tested implementations themselves on a particular set of LTL formulae. In these experiments, however, the difference between the minimum and maximum failure rates was relatively small (at most, approximately 10 units of percentage change), so finding an "optimal" value for d may not be extremely essential to the effectiveness of testing, and even sequential Kripke structures could be used. (Sequential Kripke structures in Test 3 also have the advantage that they allow the restricted LTL model checking algorithm of Sect. 5.2 to be used in the tests. In addition, the synchronous product of a sequential Kripke structure with a Büchi automaton required in the tests may be smaller than the product obtained using a more general graph of the same size, thus saving memory.)

Sizes of the Generated Automata

Since the memory requirements of the automata-theoretic LTL model checking procedure are in practice highly dependent on the size of the Büchi automata used for model checking, the sizes of the automata generated by the tested implementations is also of interest.

Tables 6.10 and 6.11 collect the average sizes of the Büchi automata (successfully) generated by each implementation from a sample of 1,000 randomly generated LTL formulae with a given parse tree size. These averages should be seen only as a very rough comparison on the relative performance of the tested implementations; estimating the average behaviour of any implementation accurately should be done by taking also of the internal structure of the implementation into account.

As could be expected, the size of the generated automata grows with the

Table 6.11: Average sizes of successfully generated Büchi automata (number of states / number of transitions; operators $\{\neg, \mathsf{X}, \Box, \Diamond, \vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$)

Parse tree size of random formulae	ÅSA+	SPIN v3.3.3	SPIN v3.3.9	SPIN v3.4.1	LTL2AUT (GPVW)	LTL2AUT (GPVW+)	LTL2AUT (LTL2AUT)
5	6/9	3/5	3/4	3/4	6/9	5/8	5/7
6	7/13	4/6	4/6	3/5	7/12	6/11	6/10
7	8/16	4/7	4/7	3/6	8/16	7/14	6/12
8	11/24	5/10	5/10	4/9	10/23	9/19	8/16
9	13/32	5/12	6/14	5/12	12/30	10/24	9/19
10	16/47	6/16	6/16	5/14	16/45	13/32	10/25
11	19/64	7/18	7/19	6/17	19/60	15/41	12/31
12	23/84	7/22	8/22	7/20	23/80	17/55	13/38

formula size, and increasing the number of available formula operators has the same effect. The results show the difference in the power of different LTL-to-Büchi translation algorithms: PROD, which is based on one of the first translation algorithms presented in the literature [31], had worse performance than the other algorithms based on the GPVW algorithm [8] and its variants. The automata generated by ÅSA+ and LTL2AUT (GPVW) were very close to each other in size, and the other variants of LTL2AUT performed even better. The smallest automata were generated by SPIN v3.4.1; the two older versions of the tool were almost as efficient. The SPIN and PROD tools have the additional advantage of always generating Büchi automata with only one acceptance condition. Such automata can be used efficiently with e.g. the nested-depth-first search on-the-fly model checking algorithm of [3]. This is not the case with ÅSA+ and the LTL2AUT variants that produce generalized Büchi automata, usually with more than one acceptance condition.

Summary

In conclusion, the main results of the tests were:

- ÅSA, LTL2AUT (all variants) and PROD behaved very consistently in all tests. No errors were detected in any tests between these implementations. PROD, however, generated very large automata in comparison to those of the other implementations. This is due to the translation algorithm that PROD uses (the algorithm is quite different from those used in the other implementations).

- SPIN v3.3.3 and v3.3.9 suffered from some internal failures and sometimes also generated incorrect automata. One model checking cross-comparison failure was still detected also with SPIN v3.4.1 when testing it against the three above implementations.

  Although the automata generated by the various SPIN versions were very small in comparison to those produced by the other implementations, this seems to have been achieved using various optimizations requiring much memory, increasing also the complexity of the implementation. This complexity may be one reason behind the errors found in the implementation.

- Some notes on Test 3 (the model checking result cross-comparison check) were:
  - Performing the cross-comparison check with respect to every state of the Kripke structure increased testing efficiency.
  - Although based on a less systematic approach than Test 1, Test 3 had not significantly worse performance than Test 1, at least when allowing the "full" set of operators to be used in the randomly generated LTL formulae (Table 6.6). Therefore, it can be useful to perform also Test 3 in order to try to optimize testing efficiency.
  - Altering the approximate density of the Kripke structures did not have significant effect on the observed failure rates; using sequential Kripke structures as input did not notably improve or degrade testing efficiency either.
- Since Test 4 complements both Test 1 and Test 3, using all of the tests together can increase testing efficiency.

In all, the LTL-to-Büchi translator testbench based on very straightforward implementation techniques proved to be quite effective in practice, although some of the tests could not be performed due to the large size of some automata (so the testbench itself ran out of memory when performing the tests). The situation could be somewhat improved by using more sophisticated implementation techniques in the testing procedure.

7 CONCLUSIONS

This work has presented techniques for testing the correctness of implementations of LTL-to-Büchi translation algorithms used in LTL model checking tools based on the automata-theoretic approach. The methods are based on direct analysis of Büchi automata and the automata-theoretic LTL model checking procedure. Ultimately, however, the basis for all presented test methods lies in the semantics of linear temporal logic—more precisely, in the mutually exclusive relationship between the satisfiability of an LTL formula and its negation on an infinite path of a Kripke structure. This common basis can be seen in the similarity of the tests itself: all tests can basically be reduced to an emptiness check of Büchi automata (with possibly some additional result comparison).

The similar nature of most of the tests allows their easy integration into an automatic testing tool for LTL-to-Büchi translators. The experiments made in this work did not include Test 2 (the universality check for the union of two Büchi automata), which made it impossible to prove the absolute correctness of any implementation on a single LTL formula. However, the cross-comparison of several implementations against each other, together with checking the emptiness of the product of two Büchi automata that should be complementary to each other, proved to work well together as error detection techniques. Actually, even the plain result cross-comparison approach has been successful in uncovering implementation errors in actual LTL model checking tools [26, 27]: for example, this approach has helped to improve the robustness of the LTL-to-Büchi translation algorithm implementation of the SPIN model checker. The usefulness of the testing strategy was again confirmed in this work: several previously untested implementations were found to behave quite consistently with each other, and a previously undiscovered error was revealed in the SPIN model checker. This was achieved using randomly generated LTL formulae and Kripke structures of moderate size as input for the tests.

However, simple random "black box" testing is not adequate for *proving* the correctness of any LTL-to-Büchi translator. For example, the multitude of available test parameters makes it very hard to assess the actual coverage of the tests. The random Kripke structures and their possible influence on the effectiveness of testing could be removed by including also Test 2 into the testing procedure, in which case the tests would depend only on the used LTL formulae. However, integrating Test 2 into the automatic testing procedure would require the implementation of a Büchi automata complementation algorithm with exponential worst-case space requirements in the size of the input.

Of course, also the implementation details could be taken into account when adjusting test parameters. However, even though this may increase testing efficiency, the test results would still remain at best inconclusive, no matter how much testing was performed. As can be seen in the experiments with SPIN v3.4.1, random "black box" testing will very rarely find any errors in an "almost correct" implementation. Increasing the number of 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 easy-to-fix bugs and omissions in the implementation. The test methods might also be of some use in making optimizations or other improvements to a translation algorithm implementation, in order to test whether the implementation seems to preserve its correctness between different releases.

Section 5.2 presented a restricted model checking algorithm for sequential Kripke structures. The algorithm was used in the analysis of test failures between two LTL-to-Büchi translation algorithm implementations in order to detect which one of the implementations had failed. Since counter-examples produced by real LTL model checking tools can usually be interpreted as sequential Kripke structures, this algorithm could validate the counter-examples found by the tool as an additional final step of the model checking procedure [27]. This way, the tool could by itself ensure the validity of the counter-example, which provides the tool a means for automatically detecting an internal failure that would otherwise have resulted in a false negative answer. This specialized model checking algorithm may also have applications elsewhere. For example, it may be possible to further extend the validation of counter-examples into additional properties not directly specified in the original property to be verified, such as assumptions concerning the environment of the system to be verified. The algorithm may also have some application as a subroutine in more general LTL model checking algorithms.

Bibliography

[1] E. M. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[2] E. M. Clarke and A. P. Sistla. The complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery*, 32(3):733–749, 1985.

[3] 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.

[4] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99), volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer-Verlag, 1999.

[5] 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)*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1999. See also "Software packages" at <URL: `http://www.cs.rice.edu/CS/Verification/`>.

[6] K. Etessami and G. Holzmann. Optimizing Büchi automata. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer-Verlag, 2000.

[7] H. J. Genrich. Predicate/transition nets. In *Petri Nets: Central Models and Their Properties – Advances in Petri Nets, Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, 1987.

[8] 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.

[9] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[10] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. See also the WWW homepage of the tool at <URL: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`>.

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

[12] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 1993.

[13] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1998.

[14] O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183. Springer-Verlag, 1999. See also an extended version at <URL: `http://www.cs.rice.edu/vardi/papers/`>.

[15] T. Latvala and K. Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1–4):175–193, 2000.

[16] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL'85)*, pages 97–107. Addison-Wesley, 1985.

[17] J. Lilius. ÅSA: The Åbo System Analyser, 1999. Available only on the WWW. See the WWW page at <URL: `http://www.abo.fi/%7Ejolilius/mc/aasa.html`>.

[18] K. L. McMillan. *Symbolic model checking – an approach to the state-explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[19] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[20] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[21] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.

[22] Mauno Rönkkö. 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 <URL: `http://www.abo.fi/%7Emauno.ronkko/`>.

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

[24] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 247–263. Springer-Verlag, 2000.

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

[26] 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.

[27] H. Tauriainen and K. Heljanko. Testing SPIN's LTL formula translation into Büchi automata using randomly generated input. In *Proceedings of the 7th International* SPIN *Workshop on Model Checking of Software (*SPIN*'2000)*, volume 1885 of *Lecture Notes in Computer Science*, pages 54–72. Springer-Verlag, 2000.

[28] W. Thomas. Languages, automata and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume III, pages 385–455. Springer-Verlag, New York, 1997.

[29] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[30] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–265. Springer-Verlag, 1996.

[31] 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.

[32] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[33] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD reference manual. Technical Report B13, Helsinki University of Technology, Digital Systems Laboratory, 1995.

[34] 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)*, volume 1254 of *Lecture Notes in Computer Science*, pages 472–475. Springer-Verlag, June 1997.

[35] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.

A EMPTINESS CHECKING IN GLOBAL SYNCHRONOUS PRODUCT

As discussed in Sect. 5.1.2, it is possible to try to improve the effectiveness of Tests 3 and 4 by computing a *global synchronous product* of a Büchi automaton A_M corresponding to some Kripke structure M with a Büchi automaton A_φ corresponding to some LTL property φ. In Sect. 5.1.2, it was stated that the construction presented in Lemma 2 (page 16) results in a structure that satisfies the global synchronous product requirements.

However, a straightforward implementation of this construction always generates a structure whose size equals the product of the sizes of A_M and A_φ, respectively. For emptiness checking purposes, only the states that are reachable from the states (q, q^0) (including these states themselves) are actually needed (here, q is a state of A_M, and q^0 is the initial state of A_φ). However, the straightforward construction always generates the worst-case product that may contain states not reachable from any of the states (q, q^0).

This same problem was addressed already in Sect. 4.2.6, where only one state (q, q^0) (the "initial state" of the product, q fixed) was considered. There, the straightforward product construction was replaced by a graph search algorithm that generates only the part of the product that contains the states reachable from (q, q^0). It is very easy to generalize this approach to multiple "initial states" of the form (q, q^0) by simply restarting the search from each such state (if the state has not already been visited during the construction). Although the worst-case result size still remains the same, it may be avoided in some cases, which will save memory.

As stated in Sect. 5.1.2, checking the emptiness of the global synchronous product also requires minor changes in the implementation. It was proposed that the algorithm for computing the MSCCs of the product automaton should be restarted in every state (q, q^0) of the product (where q is some state of A_M and q^0 the initial state of A_φ). However, simply restarting the MSCC algorithm in each of these states has the disadvantage that some states of the product automaton may be visited several times in the different runs of the MSCC algorithm. This problem can be avoided by applying Tarjan's algorithm to the product automaton only once with the following modifications:

- If the search cannot at some point find any new reachable states, it must be checked whether the product automaton still has any unvisited states. If this is the case, the search must be continued (not restarted) from any previously unvisited state, until all states of the product automaton have been visited.

- If the search finds a nontrivial MSCC with an accepting execution, it is not immediately clear from which states of the form (q, q^0) the MSCC is actually reachable. This can be determined by performing a *backward* search in the product graph to find all the states (q, q^0) that can reach the MSCC, starting the search in any state of the MSCC. The states q then correspond to M's states with an execution satisfying φ. (In practice, this search does not require extra storage space for the backward product transition relation. As a matter of fact, we can do without the *forward* transition relation for performing *all* the searches in the product automaton. The only place where the forward relation might at first seem to be needed is the search for the MSCCs, but actually the MSCCs of a graph do not depend on the direction of the arcs and can therefore be found using the reversed transition relation.)

The above improvements were used in the implementation of the emptiness checking algorithm of the testbench described in Sect. 6.1.

B CORRECTNESS OF LTL MODEL CHECKING ALGORITHM FOR SEQUENTIAL KRIPKE STRUCTURES

This appendix contains the correctness proof of the LTL model checking algorithm for sequential Kripke structures, shown in Fig. 5.6 in Sect. 5.2 (page 42).

To prove the correctness of the algorithm, we first show that the algorithm always terminates. (In the following discussion, φ denotes the given LTL formula to be model checked in the sequential Kripke structure $M = \langle S, s^0, \rho, \pi \rangle$.)

Lemma 3 *The algorithm of Fig. 5.6 terminates.*

Proof: It is easy to see that the loops between lines 9–10, 12–13, 15–16 and 18–19 will always terminate (if ever entered), because the sets AP and S are always assumed to be finite.

Also the outer loop between lines 22 and 31 will always terminate whenever it is entered. The termination of this loop would be prevented if the loop on line 25 never terminated; however, this is not possible, since the set $Marked$ always contains only finitely many elements. (The set is initially empty when entering the outer loop at line 21, and at most one element is added to it in each iteration of the outer loop. The fact that the number of iterations of the outer loop is bounded by $|S|$ now establishes the termination of the outer loop.) By the same reason, also the loop on line 33 terminates.

The termination of the main loop (lines 4–35) depends on the condition whether the set $ToEval$ is empty (line 4). Since the number of subformulae of φ is bounded by $|\varphi|$ (the number of symbols in the formula), the set $ToEval$ (initialized on line 3) initially has a finite number of elements. We argue that the algorithm removes some subformula from this set during each iteration of the main loop, decreasing the number of elements in the set. This then establishes (together with the finiteness of $ToEval$ and the fact that all of the loops inside the main loop terminate) that the set $ToEval$ will be empty after exactly $|ToEval|$ iterations, and the main loop terminates.

Assume that the algorithm *cannot* select and remove an element from $ToEval$ (lines 5–6) during some iteration of the main loop. This can happen in two cases:

- The set $ToEval$ is empty. However, this would have been detected on line 4, so the loop would not have been entered in this case at all.
- For all $\varphi' \in ToEval$, $ToEval$ also contains some proper subformula ψ of φ'. Since $\psi \in ToEval$, the same should hold for ψ, and again for some proper subformula ψ' of ψ. Continuing this way, we would obtain an infinite sequence of different formulae (all in $ToEval$), each of which (excluding the first one) is a proper subformula of the preceding formula in the sequence. But this is clearly impossible, since $ToEval$ initially contains only a finite number of formulae.

Therefore, $|ToEval|$ must decrease in each iteration of the main loop, and the algorithm terminates. □

Lemma 4 *Let φ' be the formula chosen by the algorithm from the set $ToEval$ in some iteration of the main loop. Then, the set $Result$ contains no pairs of the form (φ', s) for any $s \in S$ in the beginning of the iteration. Furthermore, no subsequent iteration will manipulate pairs of the form (ψ, s), adding them to or removing them from the set $Result$, where ψ is a subformula of φ'.*

Proof: By the proof of Lemma 3, the algorithm must choose some formula from $ToEval$ in each iteration of the main loop. It is clear that every formula ψ chosen by the algorithm in any previous iteration must be different from φ', since otherwise φ' would already have been removed from $ToEval$ and could not be selected again.

The set $Result$ is initially empty. It is easy to see from the algorithm definition (lines 10, 13, 16, 19, 24, 25 and 33) that all pairs added to the set $Result$ during a single iteration of the main loop are never associated with any formula other than the one picked from $ToEval$ in that iteration. The fact that all formulae processed before φ' are different from φ' now establishes the first part of the lemma.

It is immediate from the algorithm definition that the set $ToEval$ can contain no subformulae of φ' when φ' is selected. The second part of the lemma now follows from the fact that φ' is removed from $ToEval$ in the iteration in which it is selected, together with the note that nothing is ever removed from the set $Result$. □

The informal meaning of the previous lemma is that the algorithm "builds" the contents of the set $Result$ incrementally, one subformula at a time.

The following lemma proves a result about the way that the set $Result$ is updated during each iteration of the main loop of the algorithm.

Lemma 5 *Let φ' be the subformula chosen by the algorithm from the set $ToEval$ in some iteration of its main loop. At the end of the iteration, for all $s \in S$, $(\varphi', s) \in Result$ if and only if*

(a) $[\varphi' \in AP]$ $\quad \varphi' \in \pi(s)$;

(b) $[\varphi' = \neg\psi]$ $\quad (\psi, s) \notin Result$;

(c) $[\varphi' = (\psi_1 \lor \psi_2)]$ $\quad (\psi_1, s) \in Result$ or $(\psi_2, s) \in Result$;

(d) $[\varphi' = \mathsf{X}\,\psi]$ $\quad \big(\psi, \rho(s)\big) \in Result$;

(e) $[\varphi' = (\psi_1 \,\mathsf{U}\, \psi_2)]$ $\quad \exists j \geq 0 : \big(\psi_2, \rho^j(s)\big) \in Result$ and $\forall 0 \leq k < j : \big(\psi_1, \rho^k(s)\big) \in Result$.

Proof: By Lemma 4, we know that $\forall s \in S : (\varphi', s) \notin Result$ at the beginning of the iteration, and the algorithm will not manipulate pairs of the form (ψ, s) for any subformula ψ of φ' after the iteration. Cases (a), (b), (c) and (d) are now immediate from the definition of the algorithm. We show that case (e) also holds.

The variable s is initialized to s^0 on line 21 of the algorithm. The loop between lines 22 and 31 is repeated $|S|$ times, and the value of s is updated

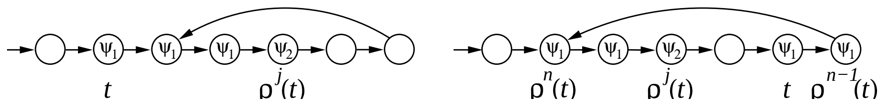


Fig. B.1: Illustration of the proof of Lemma 5

to its successor in each iteration of the loop (line 30). Thus, s cycles through all states of the Kripke structure M in sequential order. Since the Kripke structure is sequential, the "last" state of the sequence is known to be connected to some previous state in the sequence. Now, because the variable s is still updated to point to its successor also in the last iteration of the loop, it follows that s will point to some *previously visited state* of the structure at the end of the loop.

Let t be a state in S such that $\exists j \geq 0 : \big(\psi_2, \rho^j(t)\big) \in \mathit{Result}$ and $\forall 0 \leq k < j : \big(\psi_1, \rho^k(t)\big) \in \mathit{Result}$. Without loss of generality, we may assume that j is the *smallest* nonnegative integer for which $\big(\psi_2, \rho^j(t)\big) \in \mathit{Result}$ is true, so for all $0 \leq k < j$, $\big(\psi_2, \rho^k(t)\big) \notin \mathit{Result}$. By the previous note, there is an iteration of the loop between lines 22 and 31 such that the variable s points to the state t. Consider now this iteration of the loop.

If $j = 0$, it holds that $\big(\psi_2, \rho^0(t)\big) \in \mathit{Result}$, that is, $(\psi_2, t) \in \mathit{Result}$. The condition on line 23 is now true, and the lines 24–26 get executed. On line 24, the pair (φ', t) is inserted into Result. Since the algorithm never removes anything from this set and the algorithm visits each state of the Kripke structure, the result will hold for all $t \in S$ for which $(\psi_2, t) \in \mathit{Result}$ at the end of the **case** statement.

If $j > 0$, there are two cases (see Fig. B.1):

(a) The algorithm visits t before $\rho^j(t)$. Since $\forall 0 \leq k < j : \big(\psi_1, \rho^k(t)\big) \in \mathit{Result}$, it follows that the condition on line 28 will hold for $\rho^0(t) = t$ and all subsequent states $\rho^k(t)$ with $0 \leq k < j$. Therefore, after j iterations of the loop, the set Marked will contain all states $\rho^0(t) = t, \rho^1(t), \ldots, \rho^{j-1}(t)$. The algorithm then proceeds to the state $\rho^j(t)$. Since $\big(\psi_2, \rho^j(t)\big) \in \mathit{Result}$, the condition on line 23 is true. The set Result is now extended with all states (φ', s'), where $s' \in \mathit{Marked}$. Since we know that $t \in \mathit{Marked}$ still holds at this point, $(\varphi', t) \in \mathit{Result}$ will hold at the end of this iteration. Because nothing is ever removed from the set Result, $(\varphi', t) \in \mathit{Result}$ will still hold at the end of the **case** statement.

(b) The algorithm visits $\rho^j(t)$ before t. This can happen if both t and $\rho^j(t)$ are inside the cycle contained in the Kripke structure M, and $\rho^j(t)$ can be reached from t through the transition connecting the "last" state of the sequential structure to one of its predecessors. It now follows that there exists an integer $1 \leq n \leq j$ such that the algorithm has not yet visited the state $\rho^{n-1}(t)$ (or is currently in that state), but it has already visited the state $\rho^n(t)$.

Since $\big(\psi_1, \rho^k(t)\big) \in \mathit{Result}$ for all $0 \leq k < n$, all states $\rho^k(t)$ with $0 \leq k < n$ will be inserted into the set Marked during subsequent iterations of the loop, after which the loop terminates. At this point, the variable s points to the state $\rho^n(t)$.

We show that $\big(\varphi', \rho^n(t)\big) \in \mathit{Result}$ now holds at this point of the algorithm, and therefore the condition on line 32 is true. This will cause the insertion of elements (φ', s') into the set *Result* for all $s' \in \mathit{Marked}$. Since $t \in \mathit{Marked}$, it then follows that *Result* will contain the pair (φ', t) at the end of the **case** statement.

To show that $\big(\varphi', \rho^n(t)\big) \in \mathit{Result}$ holds when the loop between lines 22 and 31 terminates, we first note that the claim holds if $\rho^n(t) = \rho^j(t)$. This is because the pair $(\varphi', \rho^j(t))$ has already been inserted into *Result* when processing the state $\rho^j(t)$.

If $\rho^n(t) \neq \rho^j(t)$, the algorithm must have processed $\rho^n(t)$ before $\rho^j(t)$, since otherwise $\rho^j(t)$ would not be reachable from the state t. Therefore, $\rho^n(t)$ precedes $\rho^j(t)$ in the sequential Kripke structure. Since $\forall 0 \leq k < j : \big(\psi_1, \rho^k(t)\big) \in \mathit{Result}$, it also holds that $\forall n \leq k < j : \big(\psi_1, \rho^k(t)\big) \in \mathit{Result}$. We also know that $\big(\psi_2, \rho^j(t)\big) \in \mathit{Result}$. Since $\rho^n(t)$ and $\rho^j(t)$ have already been visited, we can apply case (a) above to conclude that $\big(\varphi', \rho^n(t)\big) \in \mathit{Result}$ holds when the loop terminates.

For the other direction, assume that $(\varphi', t) \in \mathit{Result}$ at the end of the **case** statement for some $t \in S$. We show that there now exists a $j \geq 0$ such that $\big(\psi_2, \rho^j(t)\big) \in \mathit{Result}$ and for all $0 \leq k < j$, $\big(\psi_1, \rho^k(t)\big) \in \mathit{Result}$.

By Lemma 4, *Result* contains no pairs of the form (φ', s') for any $s' \in S$ in the beginning of the loop between lines 22–31. Therefore, the pair (φ', t) must have been added to this set somewhere after line 21. The only places where this can have occurred are lines 24, 25 and 33.

In the following, we shall rely on the fact that each state of the structure is visited exactly once in the loop. This implies that each state is inserted at most once into the set *Marked*, which is easy to see from the algorithm definition.

- If (φ', t) was inserted into *Result* on line 24, the condition $(\psi_2, t) \in \mathit{Result}$ must also have been true at this point, since otherwise line 24 would not have been executed. The result now follows immediately with $j = 0$.

- If (φ', t) was inserted into *Result* on line 25, there must exist an $s' \in S$ for which the condition $(\psi_2, s') \in \mathit{Result}$ was true in some iteration of the loop between lines 22 and 31. In addition, $t \in \mathit{Marked}$ was true at this point. Since *Marked* was initially empty (and t was not inserted into it in that iteration), t was inserted into *Marked* in some *previous* iteration of the loop. The only place this may have happened is at line 28, which can have been executed for t only if $(\psi_1, t) \in \mathit{Result}$ and $(\psi_2, t) \notin \mathit{Result}$. Since t was visited before s', it must be a predecessor of s'. Therefore, there exists a $j \geq 0$ such that $s' = \rho^j(t)$.

  If $s' = \rho(t)$, the result now follows with $j = 1$.

  If $\rho(t) \neq s'$, s' is not an immediate successor of t. Assume then that there exists a $0 \leq k < j$ such that $\big(\psi_1, \rho^k(t)\big) \notin \mathit{Result}$, or

$\big(\psi_2, \rho^k(t)\big) \in \mathit{Result}$. When processing the state $\rho^k(t)$, the algorithm must have executed either lines 24–26 or the line 29, in both cases setting *Marked* to be empty (in effect, removing t from this set). It is now impossible that $t \in \mathit{Marked}$ would any longer hold when processing the state s', which is a contradiction. Therefore, it must be that for all $0 \leq k < j$, $\big(\psi_1, \rho^k(t)\big) \in \mathit{Result}$, and $\big(\psi_2, \rho^k(t)\big) \notin \mathit{Result}$. This establishes the result in this case.

- Assume that the insertion of (φ', t) into *Result* occurred at line 33. As in the previous case, $t \in \mathit{Marked}$ must have held at this point, which can be true only if for t and all of its first n successors (of which $\rho^n(t)$ is the last state processed by the loop), $\big(\psi_1, \rho^k(t)\big) \in \mathit{Result}$ and $\big(\psi_2, \rho^k(t)\big) \notin \mathit{Result}$ for all $0 \leq k \leq n$.

  At line 33, the variable s points to some previously visited state in the sequential Kripke structure. It is necessary that $(\varphi', s) \in \mathit{Result}$ was true already at line 32, since otherwise the loop on line 33 would not have been executed. Since *Result* contained no pairs related to the formula φ' before the loop on lines 22–31, the pair (φ', s) must have been inserted into *Result* in that loop on line 24 or 25. We have already shown that the result holds for such states, and therefore we may conclude that $\exists m \geq 0 : \big(\psi_2, \rho^m(s)\big) \in \mathit{Result}$ and $\forall 0 \leq k < m : \big(\psi_1, \rho^k(s)\big) \in \mathit{Result}$. The result then holds for t with $j = n + m + 1$.

$\square$

The following lemma connects the previous results with model checking LTL in the paths of the given sequential Kripke structure M.

Lemma 6 *Let φ' be the subformula of φ that the algorithm of Fig. 5.6 chooses from the set ToEval in the beginning of some iteration of the main loop. Then, at the end of the algorithm,*

$$\forall s \in S : \quad (\varphi', s) \in \mathit{Result} \quad \text{iff} \quad \xi_s \models \varphi,$$

where ξ_s is the (unique) infinite path of M $\langle s, \ldots \rangle$ starting in s.

Proof: If $\varphi' \in AP$, the algorithm enters the loop between the lines 9–10. At the end of the current iteration of the main loop, it now follows by Lemma 5 (a) that $(\varphi', s) \in \mathit{Result}$ if and only if $\varphi' \in \pi(s)$, if and only if $\xi_s \models \varphi'$ (by the semantics of LTL). By Lemma 4, the algorithm does not manipulate pairs (φ', s) after this iteration, so $(\varphi', s) \in \mathit{Result}$ still holds at the end of the algorithm. The result therefore holds for all atomic propositions occurring in φ.

Assume then that the result holds for all subformulae φ' for which $|\varphi'| \leq n$. Let φ' be a subformula of φ such that all proper subformulae of φ' are at most of length n. Therefore, φ' is either $\neg\psi_1$, $\mathsf{X}\,\psi_1$, $(\psi_1 \vee \psi_2)$ or $(\psi_1 \,\mathsf{U}\, \psi_2)$, where $|\psi_1| \leq n$ and $|\psi_2| \leq n$. We have the following cases:

- If $\varphi' = \neg\psi_1$, the loop between lines 12 and 13 is entered. We see that at the end of the loop,

$$\begin{array}{ll} (\varphi', s) \in \mathit{Result} & \text{iff [Lemma 5 (b)]} \\ (\psi_1, s) \notin \mathit{Result} & \text{iff [induction hypothesis]} \\ \xi_s \not\models \psi_1 & \text{iff [semantics of LTL]} \\ \xi_s \models \neg\psi_1 & \text{iff} \\ \xi_s \models \varphi' & \end{array}$$

for all $s \in S$.

- If $\varphi' = (\psi_1 \vee \psi_2)$, the loop between lines 15 and 16 is executed. In this case,

$$\begin{array}{ll} (\varphi', s) \in \mathit{Result} & \text{iff [Lemma 5 (c)]} \\ (\psi_1, s) \in \mathit{Result} \text{ or } (\psi_2, s) \in \mathit{Result} & \text{iff [induction hypothesis]} \\ \xi_s \models \psi_1 \text{ or } \xi_s \models \psi_2 & \text{iff [semantics of LTL]} \\ \xi_s \models (\psi_1 \vee \psi_2) & \text{iff} \\ \xi_s \models \varphi' & \end{array}$$

for all $s \in S$.

- If $\varphi' = \mathsf{X}\,\psi_1$, the algorithm enters the loop between lines 18–19. As above, we see that

$$\begin{array}{ll} (\varphi', s) \in \mathit{Result} & \text{iff [Lemma 5 (d)]} \\ \big(\psi_1, \rho(s)\big) \in \mathit{Result} & \text{iff [induction hypothesis]} \\ \xi_{\rho(s)} \models \psi_1 & \text{iff } [\xi_{\rho(s)} = \xi_s^1] \\ \xi_s^1 \models \psi_1 & \text{iff [semantics of LTL]} \\ \xi_s \models \mathsf{X}\,\psi_1 & \text{iff} \\ \xi_s \models \varphi' & \end{array}$$

for all $s \in S$.

- If $\varphi' = (\psi_1 \,\mathsf{U}\, \psi_2)$, the algorithm executes the case between lines 21–33.

$$\begin{array}{ll} (\varphi', s) \in \mathit{Result} & \text{iff [Lemma 5 (e)]} \\ \exists j \geq 0 : \big(\psi_2, \rho^j(s)\big) \in \mathit{Result} & \\ \quad \text{and for all } 0 \leq k < j, \big(\psi_1, \rho^k(s)\big) \in \mathit{Result} & \text{iff [ind. hypothesis]} \\ \exists j \geq 0 : \xi_{\rho^j(s)} \models \psi_2 & \\ \quad \text{and for all } 0 \leq k < j, \xi_{\rho^k(s)} \models \psi_1 & \text{iff [LTL semantics]} \\ \xi_s \models (\psi_1 \,\mathsf{U}\, \psi_2) & \text{iff} \\ \xi_s \models \varphi' & \end{array}$$

for all $s \in S$.

In all previous cases, Lemma 4 guarantees that the result will still hold at the end of the algorithm. □

We can now prove the correctness of the algorithm.

Proposition 1 (Correctness of the algorithm) *The algorithm of Fig. 5.6 returns the value "YES" if and only if the LTL formula φ holds in the sequential Kripke structure M.*

Proof: It is clear from the algorithm definition that $\varphi \in \mathit{ToEval}$ holds after line 3 has been executed.

By the proof of Lemma 3, the size of the set ToEval decreases in each iteration of the main loop. From this follows that there must exist an iteration in which the algorithm chooses the the formula φ from ToEval and then removes it from this set. At this point, there can be no proper subformulae of φ left in the set ToEval (otherwise φ could not be chosen), so the algorithm terminates after this iteration.

By Lemma 6, the set Result will after this iteration contain the pair (φ, s) for some $s \in S$ if and only if $\xi_s \models \varphi$. Since the algorithm then terminates, there are no subsequent iterations that could change the contents of Result.

From the algorithm definition we see that the algorithm returns the value "YES" if and only if $(\varphi, s^0) \in \mathit{Result}$ at the end of the algorithm, i.e. if and only if $\xi_{s^0} \models \varphi$ (again by Lemma 6).

Because each state of M has exactly one successor, M has only one execution beginning in its initial state s^0, and this execution corresponds to the sequence ξ_{s^0}. Therefore, $M \models \varphi$ if and only if $\xi_{s^0} \models \varphi$, if and only if the algorithm returns "YES". $\square$

C ANALYSIS OF THE LTL FORMULA GENERATION ALGORITHM

This appendix contains an analysis of the random LTL formula generation algorithm presented in Fig. 6.1 (page 46) used in the LTL-to-Büchi translator testbench and describes how the parameters in the algorithm can be adjusted so that each generated formula will have the same expected number of every individual logical or temporal operator. The analysis relies on the standard axioms of probability; for a reference, see any basic textbook on probability or statistics.

C.1 FINDING THE EXPECTED NUMBER OF OPERATORS IN A FORMULA

We will begin with finding the probability with which a given formula of parse tree size n generated by the algorithm of Fig. 6.1 contains exactly k instances of a given operator op. For this purpose, let $A_{op,k,n}$ denote the random event

$A_{op,k,n}$: "A formula with a parse tree of size n contains k instances of operator op"

Let U denote the set of available unary operators, and let B be the set of all available binary operators (in the testbench implementation, $U = \{\neg, \mathsf{X}, \Box, \Diamond\}$ and $B = \{\vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$). Let OP denote the set of all operators $U \cup B$.

As described in Chap. 6, the testbench implementation assigns to each operator $op \in OP$ an integer priority $pri(op)$ that determines the probability with which the algorithm will choose op whenever picking a random operator at lines 7 or 12. Let $\mathbf{P}_n(op)$ denote this probability for some fixed formula parse tree size $n \geq 1$. From the algorithm we can see that

$$\mathbf{P}_n(op) = \begin{cases} 0 & \text{if } n = 1 \\ \dfrac{pri(op)}{\sum\limits_{op' \in U} pri(op')} & \text{if } n = 2 \text{ and } op \in U \\ 0 & \text{if } n = 2 \text{ and } op \in B \\ \dfrac{pri(op)}{\sum\limits_{op' \in OP} pri(op')} & \text{if } n \geq 3 \end{cases} \tag{C.1}$$

We now proceed by looking at how the algorithm can generate a formula with a parse tree of size n so that the formula contains exactly k instances of operator op. For now, it is assumed that $k \geq 1$; the case $k = 0$ will be handled later. When the algorithm is called with the parameter n (before any recursive calls are executed), it can be seen that

$$\begin{aligned}
&\mathbf{P}(A_{op,k,n}) \\
&\quad = \mathbf{P}_n(\text{“the algorithm chooses } op\text{”} \\
&\qquad\qquad \wedge \text{“the algorithm will later choose } op\ k-1 \text{ times”}) \\
&\qquad + \mathbf{P}_n(\text{“the algorithm chooses an operator } op' \neq op\text{”} \\
&\qquad\qquad \wedge \text{“the algorithm will later choose } op\ k \text{ times”})
\end{aligned} \tag{C.2}$$

(It is clear that the two events in the probabilities are mutually exclusive, so the probability of the occurrence of either event is simply the sum of the probabilities of the individual events. “Later” refers to the recursive calls made by the algorithm.)

The behaviour of the algorithm in the recursive calls depends on the arity of the chosen operator and the formula parse tree size n. (Let $arity(op)$ denote the arity of op; it is always either 1 or 2.) Also the number of recursive calls depends on the arity of the chosen operator. The event “the algorithm chooses an operator $op' \neq op$” can be split into two mutually exclusive cases according to the arity of the chosen operator:

$$\begin{aligned}
&\mathbf{P}(A_{op,k,n}) \\
&\quad = \mathbf{P}_n(\text{“the algorithm chooses } op\text{”}) \\
&\qquad\qquad \cdot \mathbf{P}_{arity(op),n}(\text{“the algorithm will later choose } op\ k-1 \text{ times”}) \\
&\qquad + \mathbf{P}_n(\text{“the algorithm chooses an operator } op' \neq op\text{”}) \\
&\qquad\qquad \cdot \mathbf{P}_{arity(op'),n}(\text{“the algorithm will later choose } op\ k \text{ times”}) \\
&\quad = \mathbf{P}_n(op) \cdot \mathbf{P}_{arity(op),n}(\text{“the algorithm will later choose } op\ k-1 \text{ times”}) \\
&\qquad + \mathbf{P}_n(\text{“the algorithm chooses a unary operator } op' \neq op\text{”}) \\
&\qquad\qquad \cdot \mathbf{P}_{1,n}(\text{“the algorithm will later choose } op\ k \text{ times”}) \\
&\qquad + \mathbf{P}_n(\text{“the algorithm chooses a binary operator } op' \neq op\text{”}) \\
&\qquad\qquad \cdot \mathbf{P}_{2,n}(\text{“the algorithm will later choose } op\ k \text{ times”})
\end{aligned} \tag{C.3}$$

We have used here the fact that $\mathbf{P}_n(\text{“the algorithm chooses } op\text{”})$ is the probability $\mathbf{P}_n(op)$ defined in (C.1). In (C.3) we also have

$$\begin{aligned}
&\mathbf{P}_n(\text{“the algorithm chooses a unary operator } op' \neq op\text{”}) \\
&\quad = \begin{cases} 0 & \text{if } n = 1 \\[2ex] \dfrac{\sum_{op' \in U \setminus \{op\}} pri(op')}{\sum_{op' \in U} pri(op')} & \text{if } n = 2 \\[2ex] \dfrac{\sum_{op' \in U \setminus \{op\}} pri(op')}{\sum_{op' \in OP} pri(op')} & \text{if } n \geq 3 \end{cases}
\end{aligned} \tag{C.4}$$

and

$$\begin{aligned}
&\mathbf{P}_n(\text{“the algorithm chooses a binary operator } op' \neq op\text{”}) \\
&\quad = \begin{cases} 0 & \text{if } 1 \leq n \leq 2 \\ \dfrac{\sum_{op' \in B \setminus \{op\}} pri(op')}{\sum_{op' \in OP} pri(op')} & \text{if } n \geq 3 \end{cases}
\end{aligned} \tag{C.5}$$

If the operator chosen by the algorithm is a unary operator, the algorithm proceeds to recursively generate a subformula with a parse tree of size $n-1$. This can be considered an *independent* invocation of the algorithm with a different value for the parameter n. Clearly, n must be greater or equal to 2 for any recursive call to be generated. Thus, for all $x \geq 2$,

$$\begin{aligned}
&\mathbf{P}_{1,n}(\text{“the algorithm will later choose } op\ x \text{ times”}) \\
&\quad = \mathbf{P}(\text{“a formula with a parse tree of size } n-1 \text{ contains } x \text{ instances of} \\
&\qquad\qquad \text{operator } op\text{”}) \\
&\quad = \mathbf{P}(A_{op,x,n-1}) \qquad \text{(C.6)}
\end{aligned}$$

Choosing a binary operator results in two recursive calls to generate two subformulae with parse trees of size m and $n-m-1$ for some $1 \leq m \leq n-2$. It is safe to assume that $n \geq 3$ in this case, since otherwise the algorithm cannot choose a binary operator. If each of the possible values for m is equally probable, the possible ways to split the formula gives rise to $n-2$ equally probable cases. (In addition, these cases are again mutually exclusive: one might think of partitioning the formula into a “left-hand” and a “right-hand” subformula.) Therefore,

$$\begin{aligned}
&\mathbf{P}_{2,n}(\text{“the algorithm will later choose } op\ x \text{ times”}) \\
&\quad = \frac{1}{n-2}\sum_{m=1}^{n-2} \mathbf{P}(\text{“there are a total of } x \text{ instances of } op \text{ in two formulae} \\
&\qquad\qquad \text{with parse trees of size } m \text{ and } n-m-1, \qquad \text{(C.7)} \\
&\qquad\qquad \text{respectively”}),
\end{aligned}$$

for all $x \geq 3$.

This case can be split further into subcases according to *how many* instances of op appears in each subformula. There are $x+1$ ways to partition an integer $x \geq 0$ into two nonnegative integers such that their sum equals x. (These cases are again mutually exclusive if we think that there is a “left-hand” and a “right-hand” subformula.)

$$\begin{aligned}
&\mathbf{P}(\text{“there are a total of } x \text{ instances of } op \text{ in two formulae with parse} \\
&\qquad \text{trees of size } m \text{ and } n-m-1 \text{, respectively”}) \\
&\quad = \sum_{i=0}^{x}\Big[\mathbf{P}(\text{“a formula with a parse tree of size } m \text{ contains } i \text{ instances of} \\
&\qquad\qquad op\text{”}) \\
&\qquad \cdot \mathbf{P}(\text{“a formula with a parse tree of size } n-m-1 \text{ contains} \\
&\qquad\qquad x-i \text{ instances of } op\text{”})\Big] \\
&\quad = \sum_{i=0}^{x}\Big[\mathbf{P}(A_{op,i,m})\mathbf{P}(A_{op,x-i,n-m-1})\Big] \qquad \text{(C.8)}
\end{aligned}$$

Applying (C.8) to (C.7), we get

$$\begin{aligned}
&\mathbf{P}_{2,n}(\text{“the algorithm will later choose } op\ x \text{ times”}) \\
&\quad = \frac{1}{n-2}\sum_{m=1}^{n-2}\sum_{i=0}^{x}\Big[\mathbf{P}(A_{op,i,m})\mathbf{P}(A_{op,x-i,n-m-1})\Big] \qquad \text{(C.9)}
\end{aligned}$$

Equations (C.1), (C.4), (C.5), (C.6) and (C.9) can now be applied to (C.3) to obtain the equation for $\mathbf{P}(A_{op,k,n})$. We also make note of the following:

- Since a formula cannot contain more operators than there are nodes in the formula parse tree, and because the formula also contains at least one atomic proposition (or a Boolean constant), it follows that $\mathbf{P}(A_{op,k,n}) = 0$ for all $k \geq n$.
- The event that the formula does *not* contain an instance of some operator op is complementary to the event that the formula contains one or more instances of that operator. Taking also the previous note into account, we see that $\mathbf{P}(A_{op,0,n}) = 1 - \sum_{k=1}^{n-1} \mathbf{P}(A_{op,k,n})$.

The probability of the event $A_{op,k,n}$ is then given by the equation

$$
\mathbf{P}(A_{op,k,n}) \\
= \begin{cases}
0 & (a) \\[2ex]
\dfrac{pri(op)}{\sum\limits_{op' \in U} pri(op')} & (b) \\[2ex]
0 & (c) \\[2ex]
\dfrac{pri(op)}{\sum\limits_{op' \in OP} pri(op')} \mathbf{P}(A_{op,k-1,n-1}) + P(op,k,n) & (d) \\[2ex]
\dfrac{pri(op)}{(n-2) \sum\limits_{op' \in OP} pri(op')} \sum\limits_{m=1}^{n-2} \sum\limits_{i=0}^{k-1} \Big[\mathbf{P}(A_{op,i,m}) \mathbf{P}(A_{op,k-1-i,n-m-1}) \Big] & (e) \\
\quad + P(op,k,n) & \\[2ex]
1 - \sum\limits_{k=1}^{n-1} \mathbf{P}(A_{op,k,n}) & (f)
\end{cases} \tag{C.10}
$$

$$
\begin{array}{ll}
(a) & \text{if } k \geq n \text{ or } n = 1 \\
(b) & \text{if } k = 1,\, n = 2 \text{ and } op \in U \\
(c) & \text{if } k = 1,\, n = 2 \text{ and } op \in B \\
(d) & \text{if } k \geq 1,\, n \geq 3 \text{ and } op \in U \\
(e) & \text{if } k \geq 1,\, n \geq 3 \text{ and } op \in B \\
(f) & \text{if } k = 0 \text{ and } n \geq 1
\end{array}
$$

where

$$
\begin{aligned}
&P(op,k,n) \\
&\quad = \frac{\sum\limits_{op' \in U \setminus \{op\}} pri(op')}{\sum\limits_{op' \in OP} pri(op')} \mathbf{P}(A_{op,k,n-1}) \\
&\qquad + \frac{\sum\limits_{op' \in B \setminus \{op\}} pri(op')}{(n-2) \sum\limits_{op' \in OP} pri(op')} \sum_{m=1}^{n-2} \sum_{i=0}^{k} \Big[\mathbf{P}(A_{op,i,m}) \mathbf{P}(A_{op,k-i,n-m-1}) \Big]
\end{aligned}
$$

Equation (C.10) expresses the probability $\mathbf{P}(A_{op,k,n})$ using probabilities $\mathbf{P}(A_{op,k',n'})$, where either $k' < k$ or $n' < n$ (or both). In addition, the probabilities $\mathbf{P}(A_{op,k,n})$ with $k = 1$ and $n \leq 2$ are given. These probabilities can be used as a basis for calculating probabilities $\mathbf{P}(A_{op,k,n})$ for higher values of k and n. This leads to a "bottom-up" algorithm that can be used for finding the probability for any values of k and n. (This algorithm can run in polynomial time e.g. if the computed values $\mathbf{P}(A_{op,k,n})$ are stored into an array, which is then used to retrieve values for probabilities that have already been computed.)

Using the probability $\mathbf{P}(A_{op,k,n})$, the expected number of instances of a given operator op in a formula with a parse tree of size n is now given by

$$\mathbf{E}_{op,n} = \sum_{k=0}^{n-1} \big[k \cdot \mathbf{P}(A_{op,k,n})\big] \tag{C.11}$$

C.2 ADJUSTING OPERATOR PRIORITIES IN THE ALGORITHM

To adjust the priorities of the different operators so that each generated formula (with a fixed parse tree size) will contain the same expected number of each individual operator, we first note that it is sufficient to distinguish the operators only by their arity. This is because all choices made by the algorithm are never based on exact operator symbols. Therefore, we can identify all unary operators and all binary operators with each other, respectively, and proceed to find only two priorities pri_{u} and pri_{b} shared by the operators of different arity. Therefore, $\forall op \in U : pri(op) = pri_{\mathrm{u}}$, and $\forall op \in B : pri(op) = pri_{\mathrm{b}}$. Substituting these into (C.10) results in the slightly simplified equation

$$\mathbf{P}(A_{op,k,n}) = \begin{cases} 0 & (a) \\ \frac{1}{|U|} & (b) \\ 0 & (c) \\ \frac{pri_{\mathrm{u}}}{|U|pri_{\mathrm{u}}+|B|pri_{\mathrm{b}}}\mathbf{P}(A_{op,k-1,n-1}) + P(op,k,n) & (d) \\ \frac{pri_{\mathrm{b}}}{(n-2)(|U|pri_{\mathrm{u}}+|B|pri_{\mathrm{b}})} \sum_{m=1}^{n-2} \sum_{i=0}^{k-1} \Big[\mathbf{P}(A_{op,i,m})\mathbf{P}(A_{op,k-1-i,n-m-1})\Big] & (e) \\ \quad + P(op,k,n) & \\ 1 - \sum_{k=1}^{n-1} \mathbf{P}(A_{op,k,n}) & (f) \end{cases} \tag{C.12}$$

where the conditions (a) to (f) are as before, $|U|$ and $|B|$ are the numbers of available operators of different arities, respectively, and

$$\begin{aligned} &P(op, k, n) \\ &= \frac{\Big(|U| - \big(2 - arity(op)\big)\Big) pri_{\mathrm{u}}}{|U| pri_{\mathrm{u}} + |B| pri_{\mathrm{b}}} \mathbf{P}(A_{op,k,n-1}) \\ &\quad + \frac{\Big(|B| - \big(arity(op) - 1\big)\Big) pri_{\mathrm{b}}}{(n-2)(|U| pri_{\mathrm{u}} + |B| pri_{\mathrm{b}})} \sum_{m=1}^{n-2} \sum_{i=0}^{k} \Big[\mathbf{P}(A_{op,i,m}) \mathbf{P}(A_{op,k-i,n-m-1})\Big] \end{aligned}$$

The problem now reduces to solving the equation $\mathbf{E}_{op_1,n} - \mathbf{E}_{op_2,n} = 0$ for any two operators $op_1 \in U$, $op_2 \in B$, where the expected values are computed using (C.11). By treating another of the priorities pri_{u} and pri_{b} as a constant in this equation, the equation could now in principle be solved for the other priority to find the dependency between the two priorities. However, solving this equation exactly may be very tedious in practice. In addition, since (C.11) depends on the formula parse tree size n and the number of available unary and binary operators $|U|$ and $|B|$, it is clear that the relationship between pri_{u} and $\mathrm{pri}_{\mathrm{b}}$ will be different for each value combination for the three previous parameters. This means that a new equation would have to be solved for each such combination. (Furthermore, it may also occur that the equation has no solutions at all for some values of n, $|U|$ and $|B|$.)

Instead, it is possible to try to find approximate values for the priorities by simply guessing a value for another of the priorities and then trying to find a suitable value for the other priority such that the difference $|\mathbf{E}_{op_1,n} - \mathbf{E}_{op_2,n}|$ is minimized. Here we can use the fact that since the algorithm can choose a unary operator in two separate places, it should be that $pri_{\mathrm{b}} > pri_{\mathrm{u}}$.

The values for the priorities can be computed automatically for small values of n (as used in the experiments made in this work) by using even the following brute-force approach:

1. Let $pri_{\mathrm{u}} = 1$, and let $pri_{\mathrm{b}} = pri_{\mathrm{u}} + 1$.
2. Compute the difference $\delta = \mathbf{E}_{op_1,n} - \mathbf{E}_{op_2,n}$, where $op_1 \in U$ and $op_2 \in B$.
3. If $-\epsilon/2 < \delta < \epsilon/2$ for a given tolerance $\epsilon > 0$, return the current values of pri_{u} and pri_{b} and stop.
4. Otherwise, if $\delta < 0$, increment pri_{u}; if $\delta > 0$, increment pri_{b}. Go then back to step 2 (or stop after some maximum iteration limit has been exceeded).

Since only small values of n were used in the experiments of Chap. 6, this simplistic approach was sufficient for finding the values for the parameters pri_{u} and pri_{b}, using $\epsilon = 2 \cdot 10^{-8}$ as the tolerance. The priorities could be found for all $n \in \{5, 6, 7, 8, 9, 10, 11, 12\}$ and for both sets of operators $OP_1 = \{\neg, \Box, \Diamond, \vee, \wedge, \rightarrow, \mathsf{U}\}$ and $OP_2 = \{\neg, \mathsf{X}, \Box, \Diamond, \vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$ used with the different LTL-to-Büchi translators. For OP_1, $|U| = 3$ and $|B| = 4$; for the set OP_2, $|U| = 4$ and $|B| = 6$.

Table C.1: Operator priorities for different operator sets and different values of n

n	Operator set used			
	$\{\neg, \Box, \Diamond, \vee, \wedge, \rightarrow, \mathsf{U}\}$		$\{\neg, \mathsf{X}, \Box, \Diamond, \vee, \wedge, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$	
	pri_{u}	pri_{b}	pri_{u}	pri_{b}
5	3667	13443	1678	7357
6	2810	9909	1455	6679
7	2417	7462	2333	8757
8	1305	3736	2914	9959
9	3773	10229	1769	5646
10	1933	5031	2507	7607
11	6771	17072	4133	12061
12	3242	7969	2609	7381

The values used for the priorities in the experiments for different sets of operators and formula parse tree sizes are shown in Table C.1.

D SPIN V3.4.1 ERROR ANALYSIS

This appendix presents a short analysis on the test case that uncovered an error in SPIN v3.4.1 in the experiments of Chap. 6.

In this test case, the randomly generated formula was

$$\varphi = \Box\Box\Big(p_4 \wedge \big(p_2 \,\mathsf{U}\, (\neg\neg p_3 \wedge \Diamond p_4)\big)\Big),$$

where p_2, p_3 and p_4 are atomic propositions. This formula has 12 nodes in its parse tree. A fragment of the Büchi automaton (including its initial state with all outgoing transitions) generated by the implementation from this formula is shown in Fig. D.1.[1] It is important that the initial state of the automaton is *not* an accepting state (the automaton has one acceptance condition).

The following sequence ξ provides a witness that proves the incorrectness of the automaton:

$$\xi = \langle \{p_3, p_4\}, \{p_3, p_4\}, \{p_3, p_4\}, \ldots \rangle.$$

This witness was found automatically using the testbench.

It is easy to see that the automaton can never execute the transition corresponding to the downward arrow when given ξ as input. Instead, the automaton can only stay forever in its nonaccepting initial state, so the automaton will reject the witness.

However, the formula φ is *satisfied* in the sequence ξ, so the automaton should *accept* ξ:

First of all, $\xi^0 \models p_4$, so $\xi^0 \models \top \,\mathsf{U}\, p_4 \equiv \Diamond p_4$. In addition, $\xi^0 \models p_3$, so $\xi^0 \not\models \neg p_3$, from which it follows that $\xi^0 \models \neg\neg p_3$. Therefore, $\xi^0 \models \neg\neg p_3 \wedge \Diamond p_4$. This in turn implies that $\xi^0 \models p_2 \,\mathsf{U}\, (\neg\neg p_3 \wedge \Diamond p_4)$, and since $\xi^0 \models p_4$, $\xi^0 \models p_4 \wedge \big(p_2 \,\mathsf{U}\, (\neg\neg p_3 \wedge \Diamond p_4)\big)$ is true.

Since $\xi^i = \xi$ for all $i \geq 0$, it now follows that $\xi^i \models p_4 \wedge \big(p_2 \,\mathsf{U}\, (\neg\neg p_3 \wedge \Diamond p_4)\big)$ is true for all $i \geq 0$. From this it follows directly that $\xi^i \models \Box\Box\Big(p_4 \wedge \big(p_2 \,\mathsf{U}\, (\neg\neg p_3 \wedge \Diamond p_4)\big)\Big)$ for all $i \geq 0$, so especially $\xi^0 = \xi \models \varphi$, and the formula is satisfied in the witness. This proves that the Büchi automaton incorrectly rejects the witness.

(In practice, the testbench did a similar analysis automatically by first converting the witness into a sequential Kripke structure consisting of one state with a self-loop and then model checking the formula in the structure using the restricted LTL model checking algorithm of Sect. 5.2.)

[1]The same automaton was obtained also from the slightly simplified formula $\Box\Big(p_4 \wedge \big(p_2 \,\mathsf{U}\, (p_3 \wedge \Diamond p_4)\big)\Big)$. Actually, the formula still contains some redundancy: it can be checked that $\Box\Big(p_4 \wedge \big(p_2 \,\mathsf{U}\, (p_3 \wedge \Diamond p_4)\big)\Big)$ is equivalent to $\Box\big(p_4 \wedge (p_2 \,\mathsf{U}\, p_3)\big)$; however, this formula does not translate into the same automaton any longer. It could be argued that "real" formulae to be model checked do not usually contain this kind of redundancy. However, any implementation errors should still be fixed in order to remove any possibility of ever obtaining incorrect automata.

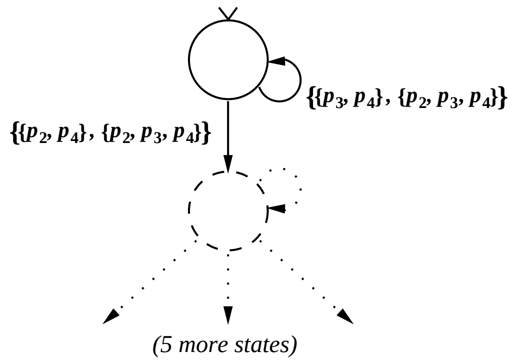


Fig. D.1: A fragment of the Büchi automaton generated by SPIN v3.4.1 from the formula $\Box\Box\Big(p_4 \wedge \big(p_2 \,\mathsf{U}\, (\neg\neg p_3 \wedge \Diamond p_4)\big)\Big)$

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A53	Stefan Rönn Semantics of Semaphores. 1998.
HUT-TCS-A54	Antti Huima Analysis of Cryptographic Protocols via Symbolic State Space Enumeration. August 1999.
HUT-TCS-A55	Tommi Syrjänen A Rule-Based Formal Model For Software Configuration. December 1999.
HUT-TCS-A56	Keijo Heljanko Deadlock and Reachability Checking with Finite Complete Prefixes. December 1999.
HUT-TCS-A57	Tommi Junttila Detecting and Exploiting Data Type Symmetries of Algebraic System Nets during Reachability Analysis. December 1999.
HUT-TCS-A58	Patrik Simons Extending and Implementing the Stable Model Semantics. April 2000.
HUT-TCS-A59	Tommi Junttila Computational Complexity of the Place/Transition-Net Symmetry Reduction Method. April 2000.
HUT-TCS-A60	Javier Esparza, Keijo Heljanko A New Unfolding Approach to LTL Model Checking. April 2000.
HUT-TCS-A61	Tuomas Aura, Carl Ellison Privacy and accountability in certificate systems. April 2000.
HUT-TCS-A62	Kari J. Nurmela, Patric R. J. Östergård Covering a Square with up to 30 Equal Circles. June 2000.
HUT-TCS-A63	Nisse Husberg, Tomi Janhunen, Ilkka Niemelä (Eds.) Leksa Notes in Computer Science. October 2000.
HUT-TCS-A64	Tuomas Aura Authorization and availability - aspects of open network security. November 2000.
HUT-TCS-A65	Harri Haanpää Computational Methods for Ramsey Numbers. November 2000.
HUT-TCS-A66	Heikki Tauriainen Automated Testing of Büchi Automata Translators for Linear Temporal Logic. December 2000.

ISBN 951-22-5275-9
ISSN 1457-7615