Improving the model checking of stutter-invariant LTL properties

Ala Eddine Ben Salem

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Ala Eddine BEN SALEM

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Improving the Model Checking of Stutter-Invariant LTL Properties

Soutenue le 25 septembre 2014 devant le jury composé de :

M. **Fabrice KORDON**, Professeur à l'UPMC (Paris 6)	Directeur de thèse
M. **Alexandre DURET-LUTZ**, Maître de conférences à l'EPITA	Encadrant
M. **Radu MATEESCU**, Directeur de recherche à l'INRIA Grenoble	Rapporteur
M. **Stefan SCHWOON**, Maître de conférences à l'ENS Cachan	Rapporteur
Mme **Béatrice BÉRARD**, Professeur à l'UPMC (Paris 6)	Examinateur
M. **Didier BUCHS**, Professeur à l'Université de Genève	Examinateur

Acknowledgments

Firstly, I would like to express my sincerest gratitude to my supervisors Fabrice KORDON and Alexandre DURET-LUTZ, for their continuous support and guidance during my PhD work, and for their patience and encouragement that greatly helped me to realize this thesis work.

I also would like to express my warm thanks to the jury members: Radu MATEESCU and Stefan SCHWOON for accepting to review my thesis, Béatrice BÉRARD and Didier BUCHS for their interest in my work.

I am especially grateful to Joël COURTOIS (director of EPITA) and Olivier RICOU (director of LRDE) for giving me the opportunity to pursue my thesis work in the EPITA Research and Development Laboratory (LRDE). I am also grateful to all my colleagues at the LRDE for the excellent and productive working environment, especially, Akim DEMAILLE, Clément DÉMOULINS, Daniela BECKER, Didier VERNA, Edwin CARLINET, Etienne RENAULT, Guillaume LAZZARA, Jonathan FABRIZIO, Myriam ROBERT-SEIDOWSKY, Nicolas BOURTRY, Reda DEHAK, Roland LEVILLAIN, Stefania CALARASANU, Thierry GÉRAUD and Yongchao XU.

Many thanks go to members of MoVe team (LIP6), especially, Denis POITRENAUD, Béatrice BÉRARD, Yann THIERRY-MIEG and Soheib BAARIR for their advices and constructive suggestions, and Etienne RENAULT, Laure MILLET, Matthieu SASSOLAS, Maximilien COLANGE, Raveca OARGA, Yann BEN MAÏSSA and Yan ZHANG for their friendship and assistance.

Last but certainly not least, I would like to thank deeply all of my family for their infinite support during the difficult time of this thesis and throughout my studies, and for all the good times we spent together.

Contents

CHAPTER 1

Introduction

Contents

1.1 Context

Software and hardware systems have become ubiquitous in our everyday life. These systems replace humans for critical tasks that involve high costs and sometimes human lives. This is the case in many areas such as medical devices, telesurgery, nuclear power plants, aircrafts industry, transportation, ...

The serious consequences caused by the failure of such systems make crucial the use of rigorous methods for system validation.

Methods based on testing and simulation have long been used for the validation of systems. However, these techniques allow to explore only a part of the possible system behaviors. The formal verification techniques are exhaustive, they guarantee that a property is satisfied by all possible system executions.

1.1.1 Model Checking and Automata-theoretic Approach

Formal verification [24] provides mathematical-based methods to ensure the correctness of a system with respect to specified behavioral properties (for example, a typical property to check in concurrent systems is the absence of deadlocks). One of the widely-used formal verification methods is *model checking* [25, 54, 8].

Taking as input a high level model describing all possible executions of the system and the property to be checked expressed as a temporal logic formula, a model checker answers if the model satisfies or not the formula. When the property is not satisfied, the model checker returns a *counterexample*, i.e., an execution of the model invalidating the property. This counterexample

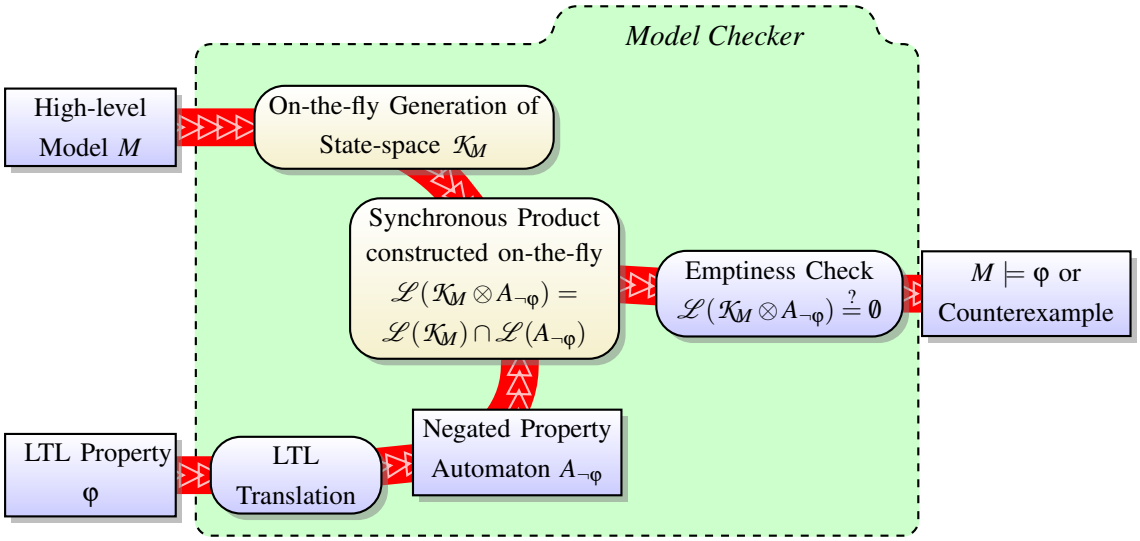


Figure 1.1: Automata-theoretic approach to *model checking*.

is useful to find errors in complex systems. This is an advantage of model checking compared to the other formal methods, such as theorem proving, which can disapprove a property but without providing such a counterexample. Another advantage is the fact that the model checking procedure is completely automatic and easy to use. This automatic procedure is based on the exploration of the system's *state-space*, i.e., a structure that describes all reachable states of the system and all transitions that the system can make between those states. The main disadvantage of model checking is discussed in Section 1.1.2.

The *automata-theoretic approach* [89, 90] to model checking represents this state-space and the property to check using variants of ω-automata [42], i.e., an extension of the classical finite automata to recognize words having infinite length (called ω-words).

The *automata-theoretic approach* splits the verification process into four operations as shown in figure 1.1:

1. Computation of the state-space for the model M. This state-space can be seen as an ω-automaton A_M whose language, $\mathcal{L}(A_M)$, represents all possible infinite executions of M.
2. Translation of the temporal property φ into an ω-automaton $A_{\neg\varphi}$ whose language, $\mathcal{L}(A_{\neg\varphi})$, is the set of all infinite executions that would invalidate φ.
3. Synchronization of these automata. This constructs a product automaton $A_M \otimes A_{\neg\varphi}$ whose language, $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\varphi})$, is the set of executions of M invalidating φ.
4. Emptiness check of this product. This operation tells whether $A_M \otimes A_{\neg\varphi}$ accepts an infinite word, and can return such a word (a counterexample) if it does. The model M verifies φ *iff* $\mathcal{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$.

1.1.2 The State-space Explosion Problem

The main difficulty of model checking is the state-space explosion problem [87] caused by the large size of the state-space of the model. For instance, modeling a system of concurrent pro-

cesses can generate a state-space which grows exponentially in the number of processes. Thus, the obtained state-space contains a very important number of states even if each process of the system has only few states. Consequently, in the automata-theoretic approach, the synchronous product of the model's state-space with the property automaton is often too large to be emptiness checked in a reasonable run time and memory. In the literature, different approaches have been proposed to improve the performance of model checking in order to push away the barrier of the State-space Explosion Problem.

The objective of this thesis is to join the fight against this problem, in the context of the verification of stutter-invariant LTL properties. In this work, we consider only finite-state systems (i.e. having a finite number of different configurations). The state-space of these systems have a finite number of states, but possibly infinite number of infinite executions.

1.1.3 Explicit versus Symbolic model checking

There are two main variants of the automata-theoretic approach: explicit and symbolic.

- In the explicit approach [e.g., 26, 48], the state-space of the model and the product are explicitly constructed by enumerating their states. "On-the-fly" emptiness check algorithms avoid the construction of the entire product and state-space by building them lazily during exploration. These on-the-fly algorithms are more efficient because they stop as soon as they find a counterexample and therefore possibly before building the entire product, thereby reducing the amount of memory and time used by the emptiness check in the case of violated properties.
- The symbolic approach [17] tries to overcome the state-space explosion obstacle by representing the state-space implicitly by means of Binary Decision Diagrams (BDDs). The intersection and union of sets of states are translated into conjunction ($\wedge$) and disjunction ($\vee$) of Boolean functions efficiently performed using BDDs (or any other variants of Decision Diagrams).

1.1.4 LTL and Stutter-invariance

In order to describe behavioral properties of complex systems, in addition to propositional logic, we need to introduce temporal operators. In this work, we focus on the model checking of stutter-invariant [39] LTL properties.

Linear Temporal Logic (LTL), introduced by Pnueli in 1977, is a propositional temporal logic widely used to express temporal properties.

The LTL syntax combines standard logical operators ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$) and temporal operators to specify that some property p happens *next time* "$\mathsf{X}\,p$", *eventually* "$\mathsf{F}\,p$", *always* "$\mathsf{G}\,p$",... (see Section 2.3.1 for a full definition of LTL syntax and semantics).

Typical examples of LTL properties have the form "$\mathsf{G}\neg p$" (p never happens), "$\mathsf{G}\,\mathsf{F}\,p$" ($p$ happens infinitely often), and "$\mathsf{F}\,\mathsf{G}\,p$" (at some point, p will hold forever).

LTL properties are constructed over the set AP of *atomic propositions*, which represent the properties of individual states. Each state of the system is labeled by a *valuation* ℓ that assigns a truth value to each atomic proposition of AP (formally, a valuation is a function $\ell : AP \mapsto \{\bot, \top\}$).

A valuation can also be viewed as a set $\ell \in 2^{AP}$ interpreted as the set of atomic propositions that are true.

An LTL property φ is interpreted over each *execution* of a system, where an execution maps each time instant to a set of atomic propositions that hold at that instant. Formally, an execution is an infinite sequence of valuations $\ell_0\ell_1 \ldots \ell_i \ldots \in (2^{AP})^\omega$.

The language of φ, denoted $\mathscr{L}(\varphi) \subseteq (2^{AP})^\omega$, is the set of all sequences of $(2^{AP})^\omega$ satisfying φ.

Among LTL properties, we want to distinguish those that are stutter-invariant.

An LTL property φ is *stutter-invariant iff* any sequence $\ell_0\ell_1\ell_3 \ldots \in \mathscr{L}(\varphi)$ remains in $\mathscr{L}(\varphi)$ after repeating any valuation ℓ_i or omitting duplicate valuations. Formally, φ is stutter-invariant iff

$$\ell_0\ell_1\ell_2 \ldots \in \mathscr{L}(\varphi) \iff \ell_0^{n_0}\ell_1^{n_1}\ell_2^{n_2} \ldots \in \mathscr{L}(\varphi) \text{ for any } n_0 > 0, n_1 > 0 \ldots$$
$$\text{where } \ell_i^{n_i} \text{ is the concatenation of } n_i \text{ copies of } \ell_i$$

Intuitively, stuttering-transitions correspond to transitions that do not change the valuation of atomic propositions between two successive states. Adding or removing stuttering in the system does not change the truth value of stutter-invariant properties. Thus, sequences that differ only in the amount of stuttering can be considered equivalent when checking stutter-invariant properties.

It is well known that any LTL$\setminus$X formula (i.e., an LTL formula that does not use the X operator) describes a stutter-invariant property. Conversely any stutter-invariant property can be expressed as an LTL$\setminus$X formula [70].

According to many research [60, 39], the restriction to stutter-invariant properties is not a serious disadvantage. In addition, there are many tools that enable specific optimizations for the verification of stutter-invariant properties, such as the partial order reduction [86, 69, 50] in Spin tool [55].

1.2 Scope of the Thesis

The general objective of this work is to tackle the state-space explosion problem by improving the performance of the model checking of stutter-invariant LTL properties.

To achieve this goal, we propose some contributions to essentially reduce the size of the product automaton and the computation time/memory used in the emptiness check of this product. Firstly, we start by looking for a form of automata that is suitable for the representation of stutter-invariant properties. As solutions, we propose new types of ω-automata that represent all the stuttering-transitions using only self-loops. Then, using these new automata, we propose some contributions to improve the performance of model checking in three contexts: explicit, symbolic and hybrid approaches (where hybrid means combining explicit and symbolic approaches).

1.2.1 Existing Work

Different kinds of ω-automata have been used in the automata-theoretic approach to explicit model checking. In the most common case, the property to be checked expressed as an LTL formula is converted into a *Büchi automaton* (BA) [16] with state-based accepting.

In Spot [64], our model checking library, we prefer to represent properties using *generalized* (i.e., multiple) Büchi acceptance conditions *on transitions* rather than on states, the obtained

automata being called *Transition-based Generalized Büchi Automata* (TGBA). We use TGBA because they allow to have a smaller [49, 36] property automaton than BA.

Unfortunately, having a smaller property automaton $A_{\neg\varphi}$ does not always imply a smaller product with the model ($A_M \otimes A_{\neg\varphi}$). Thus, instead of targeting smaller property automata, some people have attempted to build automata that are *more deterministic* [78].

Hansen et al. [52, 46] introduced an alternative type of ω-automata called *Testing Automata* (TA). TA are like BA, but instead of running them synchronously to the given transition system model, TA only observe changes on the atomic propositions. These automata are less expressive than Büchi automata since they are tailored to represent *stutter-invariant* properties. They are often larger than their equivalent BA, but according to Geldenhuys and Hansen [46], thanks to their high degree of determinism [52], the TA allow to obtain a smaller product and thus improve the performance of model checking. As a back-side, TA have two different modes of acceptance (Büchi-accepting or livelock-accepting), and consequently their emptiness check requires two passes [46], mitigating the benefits of a having a smaller product.

1.2.2 Contributions

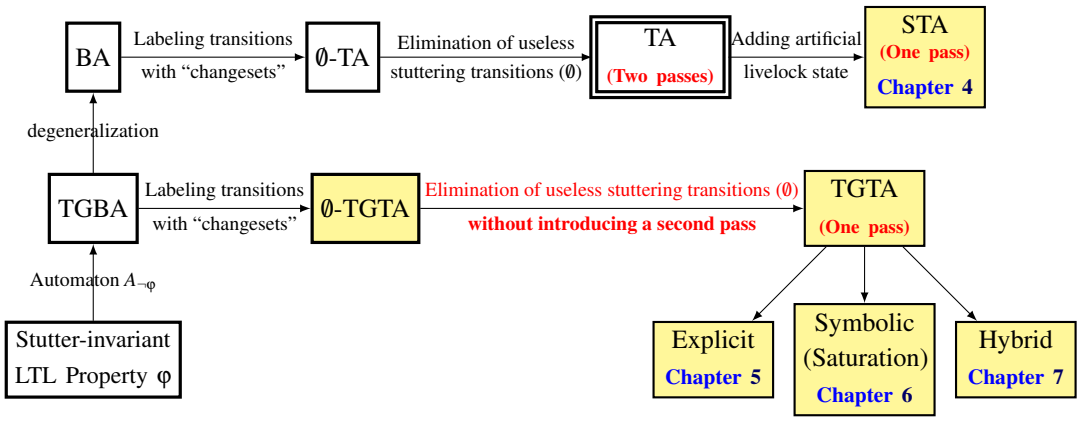


Figure 1.2: Yellow colored boxes are original contributions: the new types of ω-automata constructed in this work, with the references to the chapters that describe how to use these new automata to improve the model checking of stutter-invariant LTL properties, in three contexts: explicit, symbolic and hybrid approaches.

An overview of the existing automata and the new automata proposed in this work is shown in Figure 1.2.

On the left of this figure, the white boxes show existing ω-automata used to represent the property automaton $A_{\neg\varphi}$: the traditional *Büchi Automata* (BA) and its generalized variant *Transition-based Generalized Büchi Automata* (TGBA), and (at the center of the figure) the Testing Automata (TA), the alternative kind of automata that represent only stutter-invariant properties. TA is represented by a box with a double line edge because it requires a two-pass emptiness check algorithm (see Section 3.5.2).

The arrows between the different boxes are labeled by the successive steps used to build the different automata.

In the right part of the figure, the yellow boxes show the new kinds of ω-automata constructed in this thesis. Firstly, we propose an improvement of TA called *Single-pass Testing Automata* (STA) (Chapter 4). Secondly, we propose a more efficient new automata (Chapter 5), called *Transition-based Generalized Testing Automata* (TGTA) that combine the advantages of both STA and TGBA, and without the disadvantages of STA.

Then, using these TGTA, several improvements are proposed in three different contexts: explicit model checking (Chapter 5), symbolic model checking based on the saturation algorithm (Chapter 6) and three hybrid techniques (Chapter 7).

These contributions (yellow boxes) are briefly presented in the following:

Evaluation and improvement of the Testing Automata Approach. In Chapter 3, We experimentally evaluate the TA approach in order to extend the study of Geldenhuys and Hansen [46]. We show that while TA are statistically more efficient than BA and TGBA when the property to be verified is violated (i.e., a counterexample is found). This is not the case when the property is satisfied since the entire product has to be visited twice to check for each accepting mode of a TA (Büchi-accepting or livelock-accepting). Then, in Chapter 4 we improve the TA approach in two ways. First, we introduce some optimizations on the emptiness check algorithm in order to detect the cases where the second pass is not required. Second, we propose Single-pass Testing Automata (STA), a transformation of TA into a normal form requiring only a single pass during the emptiness check of the product. Although STA are more constrained than TA, we can automatically translate the latter into the former, by adding an artificial livelock-accepting state in STA. We have implemented these improvements in Spot library. We are thus able to compare them with the "traditional" algorithms we used on Testing Automata (TA) and Transition-based Generalized Büchi Automata (TGBA). These experiments show that STA compete well on our examples.

Transition-based Generalized Testing Automata: A Single-pass and Generalized New Automata. In Chapter 5, we propose (our main) new type of ω-automata for stutter-invariant LTL properties, called *Transition-based Generalized Testing Automata* (TGTA).

TGTA mixes features from both TA and TGBA, without the disadvantage of TA, which is the second pass of the emptiness check, and without adding an artificial state as in STA.

- From TA, TGTA reuses the labeling of transitions with changesets, and the elimination of the useless stuttering-transitions, but without introducing a second mode of acceptance (i.e, livelock-acceptance).
- From TGBA, TGTA inherit the use of transition-based generalized acceptance conditions and reuse the same single-pass emptiness check algorithm.

In addition to improving the performance, the removal of the second pass in TGTA approach also eases the implementation of the emptiness check algorithm, not only for the explicit approach, but especially for more complex implementations (such as symbolic or hybrid approaches). More generally, this simplification eases the combination of TGTA with other classical optimizations used in model checking (such as the *partial order reduction*, or the saturation technique used in the symbolic approach).

Another advantage of TGTA compared to TA is that a TA is built from a BA while a TGTA

is built from a TGBA. Therefore, TGTA can take advantage from the fact that TGBA are more concise [49, 36] than BA.

Compared to TGBA, TGTA represents all stuttering-transitions with only self-loops on all states (thanks to the elimination of the useless stuttering-transitions during the TGTA construction).

Implementation and experimentation of TGTA approach show that in most cases, it reduces the size of the synchronous product and is statistically more efficient than TA and TGBA (BA) approaches, for the explicit model checking of stutter-invariant properties,

Using TGTA to improve a Saturation-based Symbolic Model Checking. In symbolic model checking, the product is symbolically encoded by means of decision diagrams [17] and computed as a least fixpoint on its symbolic transition relation. The performance of this fixpoint computation can be improved using the saturation technique [20, 85]. In order to improve the symbolic model checking of stutter-invariant LTL properties, we investigate in Chapter 6 the use of the combination of TGTA with saturation technique. We first show how a TGTA can be symbolically encoded, Then, we show that the saturation algorithm greatly benefits from the presence of stuttering self-loops on all states of TGTA, and we propose a symbolic encoding of stuttering transitions in the product that improve the saturation-based symbolic approach using TGTA.

Implementation and experimentation of this approach confirm that it outperforms the saturation-based symbolic approach using Büchi Automata TGBA, the performance of the saturation algorithm are significantly more enhanced by TGTA than by TGBA.

This improvement was possible only because TGTA represents the stuttering-transitions specifically in a way that helps the saturation technique.

Three hybrid approaches using TGTA The *hybrid approaches* [79, 38] combine ideas from both explicit and symbolic approaches in order to benefit from the advantages of the both worlds, i.e., encoding the set of states in a concise way using decision diagrams as in the symbolic approach, and the emptiness check performed on-the-fly as in the explicit approach.

In this work, we focus on three hybrid techniques proposed in [37]: the *Symbolic Observation Graph* (SOG), the *Symbolic Observation Product* (SOP) and the *Self-Loop Aggregation Product* (SLAP).

As for symbolic model checking, Testing Automata have never been used before for hybrid model checking.

The three hybrid approaches SOG, SOP and SLAP are based on TGBA. In Chapter 7, we define and implement variations of these three approaches using TGTA instead of TGBA. Then, each original approach (SOG, SOP and SLAP) is experimentally compared against its TGTA variant (respectively SOG-TGTA, SOP-TGTA and SLAP-TGTA). The results show that the performance of these new variants depend on the type of the formula to be checked: verified or violated.

Part I

Preliminaries

CHAPTER 2

The Traditional Approaches to LTL Model Checking

Contents

2.1 Introduction

The Model checking of a behavioral property on a finite-state system is an automatic procedure that requires many phases. The first step is to formally represent the system and the property to be checked. The formalization of the system produces a model M that formally describes all

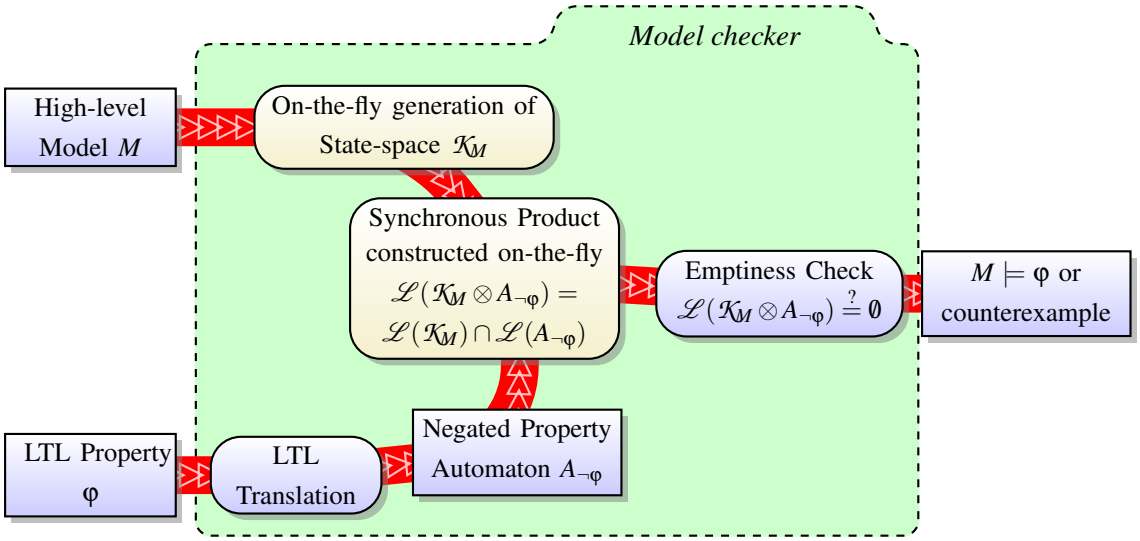


Figure 2.1: Automata-theoretic Approach to *model checking*.

the possible behaviors of the system. The property to be checked is formally described using a specification language such as branching-time (CTL) or Linear-time Temporal Logic (LTL). In this work we concentrate on LTL. The next step is to run a model checking algorithm that takes as inputs the model M and the LTL formula φ. This algorithm exhaustively checks that all the model M behaviors satisfy φ.

There are two major approaches for LTL model checking: explicit and symbolic. The *explicit* [e.g., 26, 48] approach converts M and φ into explicit graphs before running the model checking algorithm. The symbolic approach [17] encodes M and φ implicitly by means of Binary Decision Diagrams (BDDs). This symbolic approach will be presented in Chapter 6, we only consider the explicit approach in this chapter.

The automata-theoretic approach to explicit model checking relies on ω-automata, i.e., an extension of finite automata to infinite words.

Figure 2.1 summarizes the successive steps of the automata-theoretic approach. It starts by converting the negation of φ into an ω-automaton $A_{\neg\varphi}$, then composing that automaton with the state-space of a model M given as a Kripke structure $\mathcal{K}_M$ (a variant of ω-automaton), and finally checking the language emptiness of the resulting product automaton $A_{\neg\varphi} \otimes \mathcal{K}_M$ [89].

As for any model checking process, the automata-theoretic approach suffers from the well known state explosion problem [87]. In practice, it is the product automaton that can be very large, its size can reach $(|A_{\neg\varphi}| \times |\mathcal{K}_M|)$ states, which can make it impossible to be handled using the resources of modern computers.

The ω-automaton representing $A_{\neg\varphi}$ is usually a Büchi automaton (BA) or a generalization using multiple acceptance sets, such as Generalized Büchi Automata (GBA) or Transition-based Generalized Büchi Automata (TGBA).

This chapter details the successive phases of the model checking procedure, including the formalization and the different algorithms used in these phases. We also present the different variants of Büchi automata and their use in the automata-theoretic approach to model checking.

2.2 Modeling the System

A model checker tool performs verification on a model of the system rather than the system itself. A model is a high-level representation that reproduces the relevant part of the behaviors of the original system, while eliminating irrelevant details that are difficult to reproduce. The advantages are that the model includes only the relevant elements of the entire larger system, and the model is easier to build and to redesign once possible errors are reported by the model checker.

2.2.1 High Level Model

The model of a system can be described using several high-level formalisms such as Petri nets [31], Promela programs [55], communicating finite-state machines [15], or finite transition systems [1]. All these formalisms can be used to generate the state-space of the model, but the difficulty for some formalisms is to prove that the state-space is finite. For example, a state-space generated from a Petri net can be infinite, because it corresponds to the graph of reachable markings [31]. To ensure that this state-space is finite, it is necessary to check that the Petri net places are bounded.

Whatever the used formalism, a finite state-space can be viewed as a form of finite transition system, called *Kripke Structure* (its formal definition is given later).

2.2.2 Example: Robot Modeling

Let us consider a robot specification [9] as an illustrative example of System modeling.

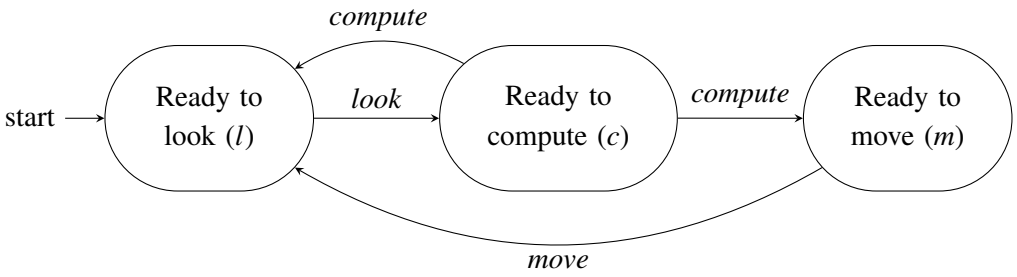


Figure 2.2: A model for robot behavior

In the finite transition system of Figure 2.2, the robot can execute only three actions: *look*, *compute*, and *move*. It begins by taking a snapshot of its environment, this action is represented by the transition *look* executed from the initial state labeled by *"Ready to look"*. Then, from the state *"Ready to compute"* the robot executes the transition *compute* to determine its future location and finally it moves to this computed location by executing the transition *move* from the state *"Ready to move"*. However, if the robot does not find a new location in the current snapshot of its environment, it returns to the initial state to take a new snapshot, hoping that its environment has changed.

2.2.3 Atomic Propositions and System Executions

The *atomic propositions* allow to describe the different states of the system. The set of atomic propositions is the set of smallest properties defined for individual states of the system (where smallest means that the truth value of each atomic proposition does not depend on the truth value of the other atomic propositions).

For instance, in order to describe the different states of the robot of Figure 2.2, let us define the three following atomic propositions:

- l = The robot is ready to look
- c = The robot is ready to compute
- m = The robot is ready to move

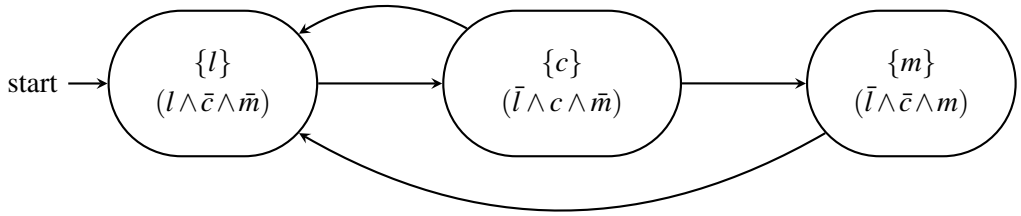


Figure 2.3: A Kripke structure for the robot model presented in Figure 2.2.

Each system state is labeled by a *valuation*, i.e., an assignment of truth value to each atomic proposition of AP.

Definition 1 (Valuation)**.** *Let AP a finite set of atomic propositions, a valuation ℓ over AP is represented by a function $\ell : AP \mapsto \{\bot, \top\}$.*

We denote by $\Sigma = 2^{AP}$ the set of all valuations over AP, where a valuation $\ell \in \Sigma$ is interpreted either as the set of atomic propositions that are true, or as a Boolean conjunction where:

$$\ell \text{ is identified to } \Big(\bigwedge_{p\in\ell} p\Big) \wedge \Big(\bigwedge_{p\in(AP\setminus\ell)} \bar{p}\Big)$$

.

For instance if $AP = \{a,b\}$, then $\Sigma = 2^{AP} = \{\{a,b\},\{a\},\{b\},\emptyset\}$ or equivalently $\Sigma = \{ab, a\bar{b}, \bar{a}b, \bar{a}\bar{b}\}$ such that:

$$\begin{array}{rcll} \{a,b\} & \leftrightarrow & a \wedge b & (ab) \\ \{a\} & \leftrightarrow & a \wedge \bar{b} & (a\bar{b}) \\ \{b\} & \leftrightarrow & \bar{a} \wedge b & (\bar{a}b) \\ \emptyset & \leftrightarrow & \bar{a} \wedge \bar{b} & (\bar{a}\bar{b}) \end{array}$$

For the robot model, the set of valuations is $\Sigma = 2^{\{l,c,m\}}$. The initial state is labeled by the valuation $\{l\}$ also noted $(l \wedge \bar{c} \wedge \bar{m})$ or $(l\bar{c}\bar{m})$, the valuations of the other states are shown in Figure 2.3.

Definition 2 (Sequence of Valuations)**.** *Let $n \in \mathbb{N} \cup \{\omega\}$ where ω is the* lowest transfinite ordinal number *defined by Cantor. A* sequence *σ of n valuations of Σ is a function $\sigma : [\![0,n[\![\mapsto \Sigma$ mapping each index from $[\![0,n[\![$ to a valuation of Σ.*

In the following, a sequence σ is often represented by the concatenation of its valuations: $\sigma = \sigma(0) \cdot \sigma(1) \cdot \sigma(2) \cdots$.
Σ^n = the set of sequences of length n and $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$ the set of finite sequences of valuations from Σ.
Σ^ω denotes the set of infinite sequences of valuations from Σ.

Definition 3 (Execution)**.** *An* execution *of a system maps to each time instant, a set of atomic propositions that hold at that instant. This execution is represented by an infinite sequence of valuations* $\sigma = \sigma(0) \cdot \sigma(1) \cdot \sigma(2) \cdots \in \Sigma^\omega$.

The set of all executions of a system is a subset of Σ^ω and can be viewed as a language of infinite words over the alphabet Σ.

An example of an execution of the robot system is the sequence of valuations $\{l\} \cdot \{c\} \cdot \{l\} \cdot \{c\} \cdot \{l\} \cdot \{c\} \cdots$ (i.e., the robot execution alternating the valuations $\{l\}$ and $\{c\}$), we can notice that there is a risk that the robot never reaches the state labeled by the valuation $\{m\}$ and therefore never moves.

In the next section, we present a variant of transition system usually used in model checking process to describe the system executions.

2.2.4 Kripke Structure

The state-space of a system can be represented by a directed graph, called Kripke structure, where vertices represent the states of the system and edges are the transitions between these states. In addition, each vertex is labeled by a valuation that represents the set of atomic propositions that are true in the corresponding state.

Definition 4 (Kripke Structure)**.** *A* Kripke structure *over the set of atomic propositions AP is a tuple* $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$*, where:*

- $\mathcal{S}$ *is a finite set of states,*
- $\mathcal{S}_0 \subseteq \mathcal{S}$ *is the set of initial states,*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ *is the transition relation,*
- $l : \mathcal{S} \to \Sigma$ *is a labeling function that maps each state s to a valuation that represents the set of atomic propositions that are true in s.*

An infinite path *or a* run *of a Kripke structure* $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ *is an infinite sequence of states* $r = s_0 \cdot s_1 \cdot s_2 \cdots$ *such that* $s_0 \in \mathcal{S}_0$ *and* $\forall i \in \mathbb{N}, (s_i, s_{i+1}) \in \mathcal{R}$.

We denote $\mathrm{Run}(\mathcal{K})$ *the set of infinite paths of* $\mathcal{K}$.

Given an infinite path $r = s_0 \cdot s_1 \cdot s_2 \cdots$ in $\mathrm{Run}(\mathcal{K})$, the infinite sequence of valuations $\sigma = l(s_0) \cdot l(s_1) \cdot l(s_2) \cdots \in \Sigma^\omega$ corresponds to an execution of the system represented by $\mathcal{K}$. Thus, σ is also called *execution* of $\mathcal{K}$.

Definition 5. *The* language *of a Kripke structure* $\mathcal{K}$ *is the set* $\mathscr{L}(\mathcal{K}) \subseteq \Sigma^\omega$ *of all executions of* $\mathcal{K}$.

$$\mathscr{L}(\mathcal{K}) = \{\sigma = l(s_0) \cdot l(s_1) \cdot l(s_2) \cdots \in \Sigma^\omega \mid s_0 \cdot s_1 \cdot s_2 \cdots \in \mathrm{Run}(\mathcal{K})\}$$

Note that $\mathscr{L}(\mathcal{K}) \subseteq \Sigma^\omega$ *can be viewed as a language of infinite words over the alphabet* $\Sigma = 2^{AP}$.

Figure 2.3.2 shows the Kripke structure representing the state-space of the robot system (described in section 2.2.2).

2.3 Specification of the Property to check

In Model checking, it is necessary to have a precise formal expression of properties to check. Logic can express these properties in a mathematical, unambiguous , and concise way. This enables the automation of the verification.

2.3.1 Linear-Time Temporal Logic (LTL)

In order to express temporal properties, *Linear Temporal Logic* (LTL) [62] adds the notion of causality to the traditional propositional logic. LTL introduces new operators, called temporal operators, interpreted over linear executions. In a linear execution, every time instant has only one immediate successor, unlike the other widely-used temporal logic, called CTL (*Computational Tree Logic*) [23], for which a time instant can have many immediate successors. This two logics are not really comparable [55]. Each logic allows to express properties that the other can not express. For instance, CTL allows to express the reset [55] properties and LTL cannot. However, CTL does not allow to express the invariance LTL properties (of the form $\mathsf{F}\,\mathsf{G}\,p$). In this work, we only focus on LTL model checking.

2.3.1.1 LTL Formula

An *LTL formula* is composed of:

- A finite set of atomic propositions $AP = \{p_1, p_2, \ldots\}$,
- the Boolean operators $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$,
- the unary temporal operators X (*next*), G (*Globally*) and F (*Future*),
- the binary temporal operator U (*Until*).

Definition 6. *Given a set of atomic propositions AP. The set of LTL formulas over AP is inductively defined as follows:*

- *For every atomic proposition $p \in AP$, p is an LTL formula,*
- *if φ is an LTL formula, then $\neg\varphi$, $\mathsf{X}\varphi$, $\mathsf{G}\varphi$ and $\mathsf{F}\varphi$ are also LTL formulas,*
- *if φ_1 and φ_2 are two LTL formulas, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$, $(\varphi_1 \leftrightarrow \varphi_2)$ and $(\varphi_1 \,\mathsf{U}\, \varphi_2)$ are also LTL formulas.*

2.3.1.2 LTL Formula Semantics

An LTL formula over *AP* is interpreted over an execution $\sigma \in \Sigma^\omega$ ($\Sigma = 2^{AP}$).
In the following, $\sigma(n)$ denotes the $(n+1)^{th}$ valuation of the execution $\sigma = \sigma(0)\sigma(1)\sigma(2)\ldots$, and the sequence $\sigma^i = \sigma(i)\sigma(i+1)\ldots$ denotes the suffix of σ starting at position i (i.e., $\forall n \in \mathbb{N}$, $\sigma^i(n) = \sigma(i+n)$).

Definition 7. *The satisfaction of an LTL formula* φ *by an execution* $\sigma \in \Sigma^\omega$*, denoted* $\sigma \models \varphi$*, is defined inductively as follows:*

$$
\begin{array}{ll}
\sigma \models p & \text{iff } p \in \sigma(0) \\
\sigma \models \neg\varphi & \text{iff } \neg(\sigma \models \varphi) \\
\sigma \models \varphi_1 \wedge \varphi_2 & \text{iff } \sigma \models \varphi_1 \textit{ and } \sigma \models \varphi_2 \\
\sigma \models \varphi_1 \vee \varphi_2 & \text{iff } \sigma \models \varphi_1 \textit{ or } \sigma \models \varphi_2 \\
\sigma \models \mathsf{X}\varphi & \text{iff } \sigma^1 \models \varphi \quad (\varphi \textit{ is true in the next time step}) \\
\sigma \models \mathsf{G}\varphi & \text{iff } \forall i \geq 0, \sigma^i \models \varphi \quad (\varphi \textit{ is true in every time step}) \\
\sigma \models \mathsf{F}\varphi & \text{iff } \exists i \geq 0 \textit{ such that } \sigma^i \models \varphi \quad (\varphi \textit{ is true now or at some future time step}) \\
\sigma \models \varphi_1 \mathsf{U} \varphi_2 & \text{iff } \exists i \geq 0 \textit{ such that } \sigma^i \models \varphi_2 \textit{ and } \forall j \in [\![0, i-1]\!], \sigma^j \models \varphi_1 \quad (\varphi_2 \textit{ is true now or} \\
 & \quad \varphi_1 \textit{ is true now and } \varphi_1 \textit{ remains true until } \varphi_2 \textit{ holds})
\end{array}
$$

Definition 8. *Given an LTL formula* φ *over AP. The language of* φ*, denoted* $\mathscr{L}(\varphi)$*, is the set of all executions* $\sigma \in \Sigma^\omega$ *satisfying* φ*.*

$$\mathscr{L}(\varphi) = \{\sigma \in \Sigma^\omega \mid \sigma \models \varphi\}$$

2.3.1.3 LTL Operators Equivalences

In Definition 6, some operators are redundant (i.e., can be expressed using other operators). The following equivalences allow to ignore these operators in the LTL translation algorithm presented in Section 2.4.6.

$$
\begin{aligned}
\top = \mathit{True} &= p \vee \bar{p} \\
\bot = \mathit{False} &= p \wedge \bar{p} \\
\mathsf{F}\varphi &= \top \mathsf{U} \varphi \\
\mathsf{G}\varphi &= \neg\mathsf{F}\neg\varphi = \neg(\top \mathsf{U} \neg\varphi) = \bot \mathsf{R} \varphi
\end{aligned}
$$

2.3.1.4 Size of an LTL formula

Definition 9. *Given an LTL formula* φ *over AP, the size (or the length) of* φ*, denoted* $|\varphi|$*, is the total number of symbols of* φ*, i.e., the number of atomic propositions, constants, and operators (logical and temporal) occurring in* φ*.*

Formally, the size of an LTL formula is defined inductively as follows:

$$\begin{aligned} |\bot| &= 1 \\ |\top| &= 1 \\ |p| &= 1, \forall p \in AP \\ |\neg\varphi| &= 1 + |\varphi| \\ |\varphi_1 \wedge \varphi_2| = |\varphi_1 \vee \varphi_2| &= 1 + |\varphi_1| + |\varphi_2| \\ |\mathsf{X}\varphi| = |\mathsf{G}\varphi| = |\mathsf{F}\varphi| &= 1 + |\varphi| \\ |\varphi_1 \,\mathsf{U}\, \varphi_2| &= 1 + |\varphi_1| + |\varphi_2| \end{aligned}$$

2.3.2 LTL and Kripke structure

The link between an LTL formula and a Kripke structure is obvious, since we can interpret the LTL formula over the executions of the Kripke structure .

Definition 10. *We say that a Kripke structure $\mathcal{K}$ over $\Sigma = 2^{AP}$, satisfies an LTL formula φ, denoted $\mathcal{K} \models \varphi$, if all executions in $\mathscr{L}(\mathcal{K})$ satisfy φ.*

$$\mathcal{K} \models \varphi \iff \mathscr{L}(\mathcal{K}) \subseteq \mathscr{L}(\varphi)$$

In the example of the robot model, the Kripke structure of Figure does not satisfy the LTL formula $\varphi = \mathsf{F}\, m$: a counterexample (i.e., an execution of the Kripke structure that does not satisfy φ) is $\{l\} \cdot \{c\} \cdot \{l\} \cdot \{c\} \cdot \{l\} \cdot \{c\} \cdots$ (the robot alternating $\{l\}$ and $\{c\}$ and never reaches the state labeled by the valuation $\{m\}$; therefore it never moves).

2.3.3 Stutter-Invariant LTL Formulas

Definition 11. *An LTL formula φ is* stutter-invariant *[39]* iff *any sequence $\sigma(0)\sigma(1)\ldots \in \mathscr{L}(\varphi)$ remains in $\mathscr{L}(\varphi)$ after repeating any valuation $\sigma(i)$ or omitting duplicate valuations. Formally, φ is stutter-invariant* iff

$$\sigma(0)\sigma(1)\ldots \in \mathscr{L}(\varphi) \iff \underbrace{\sigma(0)\ldots\sigma(0)}_{n_0\times\sigma(0)}\underbrace{\sigma(1)\ldots\sigma(1)}_{n_1\times\sigma(1)}\ldots \in \mathscr{L}(\varphi), \text{ for any } n_0 > 0, n_1 > 0 \ldots$$

Intuitively, a property is stutter-invariant if it is insensitive to stuttering transitions i.e., the transitions that do not change the values of atomic propositions between two states of the system.

For example, the LTL formula $\varphi = a \,\mathsf{U}\, b$ is stutter-invariant because valuations can be repeated or omitted in any sequence that satisfies φ (e.g., $a\bar{b};\bar{a}b;\bar{a}b;\bar{a}\bar{b}\ldots$), and the obtained sequences remain satisfying φ (e.g., $a\bar{b};a\bar{b};\bar{a}b;\bar{a}\bar{b}\ldots$). Conversely, the LTL formula $\varphi = a \wedge \mathsf{X}b$ is not stutter-invariant because it is satisfied by the sequence $(a\bar{b};\bar{a}b;\ldots)$ but unsatisfied by the sequence $(a\bar{b};a\bar{b};\bar{a}b;\ldots)$.

Theorem 1. *Any LTL$\setminus$X formula (i.e., an LTL formula that does not use the X (next-time) operator) describes a stutter-invariant property. Conversely, any stutter-invariant property can be expressed as an LTL$\setminus$X formula [70].*

2.4 From LTL to Büchi Automata

The automata-theoretic approach is based on the transformation of the LTL formula to check into an automaton that accepts the same executions (called infinite words in the context of automata theory).

The following sections present three variants of Büchi automata [16] that can be used to express properties in the automata-theoretic approach to model checking: the standard Büchi Automata (BA), Generalized Büchi Automata (GBA) and Transitions-based Generalized Büchi Automata (TGBA). The main difference between these automata is the way they accept an infinite word of $(2^{AP})^{\omega}$.

2.4.1 The "Traditional Büchi" Automata (BA)

The Büchi Automata were introduced by J.R Büchi [16] in 1962. They are ω-automata [42] with labels on transitions and acceptance conditions on states. While classical finite automata recognize words having finite length, ω-automata (and in particular Büchi Automata) recognize words of infinite length, called ω-words. Although they accept infinite words, Büchi Automata have a finite number of states.

In the following, we will use the abbreviation BA for the standard variant of Büchi Automata.

Definition 12 (BA)**.** *A* Büchi Automaton (BA) *over the alphabet* $\Sigma = 2^{AP}$ *is a tuple* $\mathcal{B} = \langle Q, I, \delta, \mathcal{F} \rangle$ *where:*

- Q *is a finite set of states,*
- $I \subseteq Q$ *is a a finite set of initial states,*
- $\mathcal{F} \subseteq Q$ *is a finite set of accepting states (*$\mathcal{F}$ *is called the accepting set),*
- $\delta \subseteq Q \times \Sigma \times Q$ *is the transition relation where each transition is labeled by a letter* ℓ *of* Σ*, i.e., each element* $(q, \ell, q') \in \delta$ *represents a transition from state* q *to state* q' *labeled by a valuation* $\ell \in 2^{AP}$*.*

Definition 13 (A BA run)**.** *A* run *of* $\mathcal{B}$ *over an infinite word* $\sigma = \ell_0\ell_1\ell_2 \ldots \in \Sigma^{\omega}$ *is an infinite sequence of transitions* $r = (q_0, \ell_0, q_1)(q_1, \ell_1, q_2)(q_2, \ell_2, q_3) \ldots \in \delta^{\omega}$ *such that* $q_0 \in I$ *(i.e., the infinite word is recognized by the run). Such a run is said to be accepting if* $\forall i \in \mathbb{N}, \exists j \geq i, q_j \in \mathcal{F}$ *(i.e., at least one accepting state is visited infinitely often).*

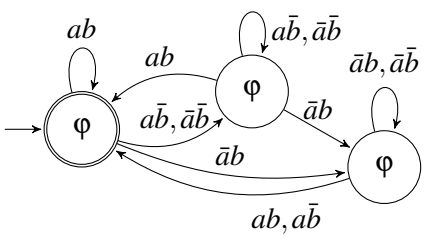
The infinite word σ *is accepted by* $\mathcal{B}$ *if there exists an accepting run of* $\mathcal{B}$ *over* σ*. The* language *accepted by* $\mathcal{B}$ *is the set* $\mathcal{L}(\mathcal{B}) \subseteq \Sigma^{\omega}$ *of the infinite words it accepts.*

Figure 2.4 shows two examples of BA:

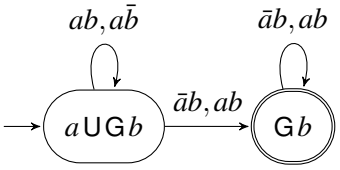
(a) Figure 2.4a is a BA recognizing the runs where a is true infinitely often and b is true infinitely often, i.e., recognizing the LTL formula $(\mathsf{GF}a \wedge \mathsf{GF}b)$,

(b) Figure 2.4b is a BA recognizing the LTL formula $(a\,\mathsf{U}\,\mathsf{G}b)$.

In these BA, the initial states are indicated by an arrow without source state " $\longrightarrow$ ". The Boolean conjunctions labeling each transition are valuations over $AP = \{a, b\}$. The accepting states are indicated by a double circle. The LTL formulas labeling each state represent the property accepted



(a) A BA for the LTL property $\varphi = \mathsf{GF}a \wedge \mathsf{GF}b$



(b) A BA for the LTL property $a\mathsf{U}\mathsf{G}b$

Figure 2.4: Two examples of BA, with accepting states shown as double circles.

starting from this state of the automaton: they are shown for the reader's convenience but not used for model checking.
As an illustration of the BA definition, the infinite word $ab;a\bar{b};\bar{a}b;ab;\bar{a}b;ab;\ldots$ is accepted by the BA of Figure 2.4b that recognizes $a\mathsf{U}\mathsf{G}b$. A run over such infinite word must start in the initial state labeled by the formula $(a\mathsf{U}\mathsf{G}b)$ and remains in this state for the first two valuations $ab;a\bar{b}$, then it changes the value of a, so it has to take the transition labeled by the valuation $\bar{a}b$ to move to the second state labeled by the formula $(\mathsf{G}b)$. Finally, to be accepted, it must stay on this accepting state by executing infinitely the transitions labeled by $\{\bar{a}b, ab\}$. The obtained accepting run is:
$(a\mathsf{U}\mathsf{G}b) \xrightarrow{ab} (a\mathsf{U}\mathsf{G}b) \xrightarrow{a\bar{b}} (a\mathsf{U}\mathsf{G}b) \xrightarrow{\bar{a}b} (\mathsf{G}b) \xrightarrow{ab} (\mathsf{G}b) \xrightarrow{\bar{a}b} (\mathsf{G}b) \xrightarrow{ab} (\mathsf{G}b)\cdots(\mathsf{G}b)\cdots$

2.4.2 Generalized Büchi Automata (GBA)

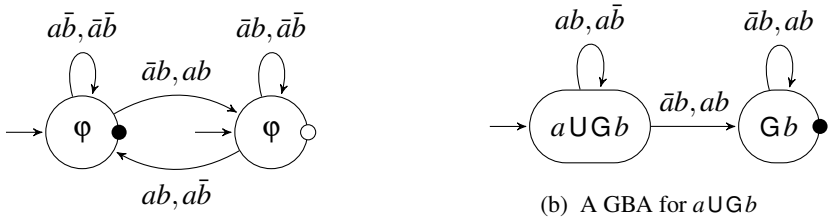
In the generalized variant of Büchi Automata (GBA) [48], there are multiple acceptance conditions, in other words, $\mathcal{F}$ is a set of accepting sets of states and a run is accepted *iff* it visits infinitely often each accepting set in $\mathcal{F}$.

Definition 14 (GBA)**.** *A Generalized Büchi Automata (*GBA*) over the alphabet $\Sigma = 2^{AP}$ is a tuple $\mathcal{G} = \langle Q, I, \delta, \mathcal{F} \rangle$ where:*

- *Q is a finite set of states,*
- *$I \subseteq Q$ is a set of initial states,*
- *$\mathcal{F} \subseteq 2^Q$ is a* set of sets *of accepting states (we call* accepting set *each set F_i of $\mathcal{F} = \{F_1, F_2, \ldots, F_k \ldots\}$),*
- *$\delta \subseteq Q \times \Sigma \times Q$ is the transition relation where each transition is labeled by a letter $\ell \in \Sigma$ (i.e., each element $(q, \ell, q') \in \delta$ represents a transition from state q to state q' labeled by a valuation $\ell \in 2^{AP}$).*

Definition 15. *A* run *of $\mathcal{G}$ over an infinite word $\sigma = \ell_0\ell_1\ell_2 \ldots \in \Sigma^\omega$ is an infinite sequence of transitions $r = (q_0, \ell_0, q_1)(q_1, \ell_1, q_2)(q_2, \ell_2, q_3)\ldots \in \delta^\omega$ such that $q_0 \in I$ (i.e., the infinite word is recognized by the run).*
Such a run is said to be accepting if $\forall F \in \mathcal{F}, \forall i \geq 0, \exists j \geq i, q_j \in F$ (i.e., at least one accepting state of each accepting set $F \in \mathcal{F}$ is visited infinitely often).

The infinite word σ is accepted by $\mathcal{G}$ iff *there exists an accepting run of $\mathcal{G}$ over σ. The* language *accepted by $\mathcal{G}$ is the set $\mathcal{L}(\mathcal{G}) \subseteq \Sigma^\omega$ of infinite words it accepts.*



(a) A GBA for $\varphi = \mathsf{GF}\,a \wedge \mathsf{GF}\,b$

(b) A GBA for $a\,\mathsf{U}\,\mathsf{G}\,b$

Figure 2.5: Two examples of GBA: (a) A GBA for $\varphi = \mathsf{GF}\,a \wedge \mathsf{GF}\,b$ with two accepting sets of states indicated by ● and ○. (b) A GBA for $a\,\mathsf{U}\,\mathsf{G}\,b$ with a single accepting set indicated by ●.

GBA are as expressive as BA: any GBA can be converted into an equivalent BA that recognizes the same language (section 2.4.2.1), and vice-versa, a BA $\langle Q, I, \delta, \mathcal{F} \rangle$ can be viewed as a GBA $\langle Q, I, \delta, \{\mathcal{F}\} \rangle$ having a single set of accepting states $\{\mathcal{F}\}$.

Figure 2.5 shows two examples of GBA that are equivalent to the two BA of Figure 2.4 such that:

(a) The GBA of Figure 2.5a recognizes $(\mathsf{GF}\,a \wedge \mathsf{GF}\,b)$. An accepting run in this GBA has to visit the two accepting states indicated by ● and ○ infinitely often. Therefore, it must explore infinitely often, the transition labeled by $\{ab, a\bar{b}\}$ (a is true) and the transition $\{\bar{a}b, ab\}$ (b is true).

(b) The GBA of Figure 2.5b is the same as the BA of Figure 2.4b, with the accepting state is indicated by the single acceptance condition $\{\bullet\}$. More generally, a BA $\langle Q, I, \delta, \mathcal{F} \rangle$ can be viewed as a GBA with a single set of accepting states $\langle Q, I, \delta, \{\mathcal{F}\} \rangle$.

2.4.2.1 From GBA to BA (degeneralization)

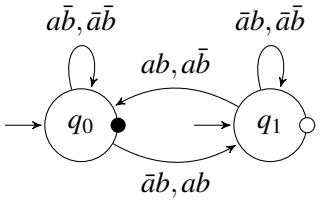
The conversion of a GBA into a BA is called *degeneralization* [4, section 4.3.4]. A GBA having s states and k accepting sets of states can be *degeneralized* into an equivalent BA with $(k \times s)$ states.

Given a GBA $\mathcal{G} = \langle Q, I, \delta, \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k\} \rangle$, the BA $\mathcal{B} = \langle Q', Q'^0, \delta', \mathcal{F}' \rangle$ constructed as follows accepts the same language as $\mathcal{G}$.

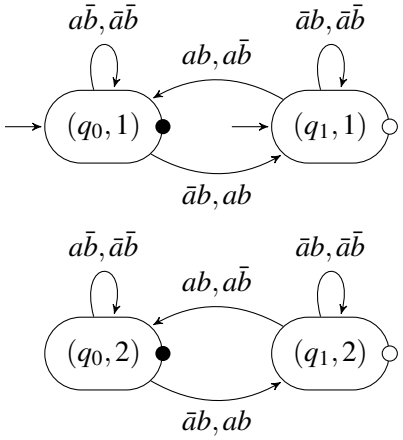
- $Q' = Q \times \{1, \ldots, k\}$ (Q' is the states of Q marked by an integer in $\{1, \ldots, k\}$)
- $Q'^0 = Q \times \{1\}$
- $\mathcal{F}' = \mathcal{F}_1 \times \{1\}$
- $\forall (q,i) \in Q'$, $\big((q,i), \ell, (q',i')\big) \in \delta'$ if $(q, \ell, q') \in \delta$ and $\begin{cases} i' = i & \text{if } q \notin \mathcal{F}_i \\ i' = (i \bmod k) + 1 & \text{if } q \in \mathcal{F}_i \end{cases}$ (In the i^{th} copy, the states of $\mathcal{F}_i$ are connected to its successors in the $(i+1)^{th}$ copy (the $(k+1)^{th}$ copy is the first copy)

Figure 2.6 illustrates the successive steps to degeneralize the GBA $\mathcal{G}$ of $\mathsf{GF}\,a \wedge \mathsf{GF}\,b$:

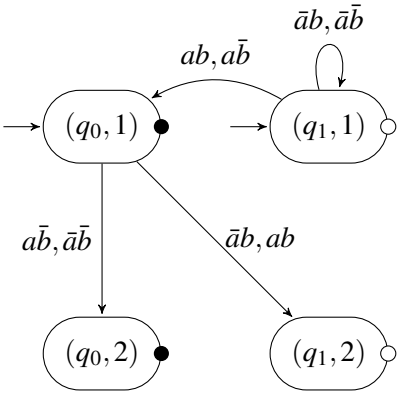
- The GBA $\mathcal{G}$ is presented in Figure 2.6a with $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2\} = \{\{q_0\}, \{q_1\}\}$ respectively indicated by ● and ○.



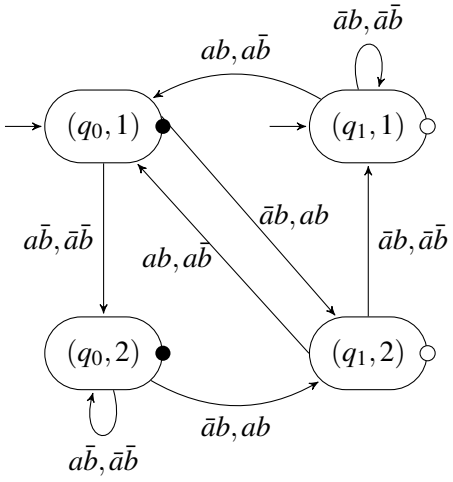
(a) GBA $\mathcal{G}$ for $\varphi = (\mathsf{GF}a \wedge \mathsf{GF}b)$, with $\mathcal{F} = \{\{q_0\}, \{q_1\}\}$ such that $\mathcal{F}_1 = \{q_0\}$ and $\mathcal{F}_2 = \{q_1\}$ respectively indicated by ● and ○.



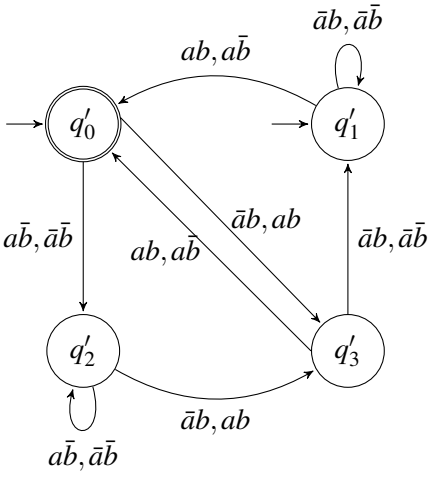
(b) Duplicate the GBA (two copies)



(c) Computing the transitions of the first copy: i.e., states $(q_0, 1)$ and $(q_1, 1)$



(d) Computing the transitions of the second copy: i.e., states $(q_0, 2)$ and $(q_1, 2)$



(e) The obtained BA $\mathcal{B}$ for $\varphi = (\mathsf{GF}a \wedge \mathsf{GF}b)$, with the accepting state is a double circle

Figure 2.6: The successive steps to degeneralize a GBA into BA.

- The degeneralization begin by duplicate k times the GBA, with $k = |\mathcal{F}| = 2$. The obtained BA of Figure 2.6b is composed of two copies of the GBA , the initial states are only in the first copy.

- In Figure 2.6c, we add the transitions of the first copy of states, the state $(q_0, 1)$ is connected to its successors in the second copy because $q_0 \in \mathcal{F}_1$.

- Similarly, in Figure 2.6d, because $q_1 \in \mathcal{F}_2$, the state $(q_1, 2)$ of the second copy is connected to its successors in the first copy. In the two copies, we keep unchanged the transitions of the two other states $(q_1, 1)$ and $(q_0, 2)$.

- Finally, the obtained BA $\mathcal{B}$ is shown in Figure 2.6e, it could be reduced by merging the two bisimilar states q'_1 and q'_3. This optimization is called bisimulation reduction and will be presented in section 2.4.5.

- Each accepting run of CB is also accepted by $\mathcal{G}$:

  An accepting run r_1 of the obtained BA $\mathcal{B}$ has to visit infinitely often at least one accepting state that is in the first copy. To achieve this, r_1 must visit all the intermediate copies that are only connected through accepting set of the GBA (each $\mathcal{F}_i$ connects the i^{th} copy to the $(i+1)^{th}$ copy). Therefore, r_1 has to visit each accepting set $\mathcal{F}_i$ infinitely often, thus r_1 is also an accepting run for the original GBA $\mathcal{G}$.

- Conversely, each accepting run of CG is also accepted by $\mathcal{B}$:

  We can consider, without loss of generality, that an accepting run r_2 of the GBA $\mathcal{G}$ visits infinitely often all the accepting set in the order $\mathcal{F}_1, \mathcal{F}_2, \ldots$. Therefore, r_2 visits infinitely often at least one state of $\mathcal{F}' = \mathcal{F}_1 \times \{1\} = \{(q_0, 1)\} = \{q'_0\}$ with q'_0 is an accepting state of $\mathcal{B}$, thus r_2 is also an accepting run for the BA $\mathcal{B}$.

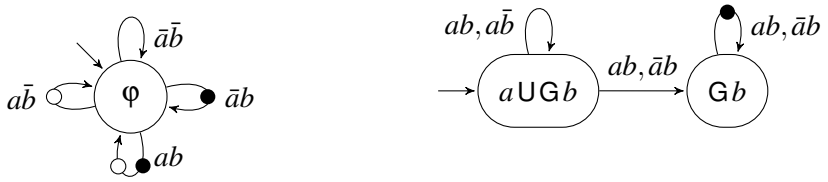
2.4.3 Transition-based Generalized Büchi Automata (TGBA)

A Transition-based Generalized Büchi Automaton (TGBA) [49] is a variant of a Büchi automaton that has multiple acceptance conditions on transitions.

Definition 16 (TGBA)**.** *A* TGBA *over the alphabet* $\Sigma = 2^{AP}$ *is a tuple* $\mathcal{G}' = \langle Q, I, \delta, \mathcal{F} \rangle$ *where:*

- Q *is a finite set of states,*
- $I \subseteq Q$ *is a set of initial states,*
- $\mathcal{F}$ *is a finite set of acceptance conditions,*
- $\delta \subseteq Q \times \Sigma \times 2^{\mathcal{F}} \times Q$ *is the transition relation, where each element* $(q, \ell, F, q') \in \delta$ *represents a transition from state* q *to state* q' *labeled by a valuation* $\ell \in 2^{AP}$*, and a set of acceptance conditions* $F \in 2^{\mathcal{F}}$*.*

Definition 17. *A* run *of* $\mathcal{G}'$ *over an infinite word* $\sigma = \ell_0 \ell_1 \ell_2 \ldots \in \Sigma^\omega$ *is an infinite sequence of transitions* $r = (q_0, \ell_0, F_0, q_1)(q_1, \ell_1, F_1, q_2)(q_2, \ell_2, F_2, q_3)\ldots \in \delta^\omega$ *such that* $q_0 \in I$ *(i.e., the infinite*



(a) TGBA of $\varphi = (\mathsf{GF}a \wedge \mathsf{GF}b)$

(b) TGBA for $(a\,\mathsf{U}\,\mathsf{G}b)$

Figure 2.7: (a) A TGBA with acceptance conditions $\mathcal{F} = \{\bullet, \circ\}$ recognizing the LTL property $\varphi = \mathsf{GF}a \wedge \mathsf{GF}b$. (b) A TGBA with $\mathcal{F} = \{\bullet\}$ recognizing the LTL property $a\,\mathsf{U}\,\mathsf{G}b$.

word is recognized by the run). Such a run is said to be accepting if $\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$ *(i.e., each acceptance condition is visited infinitely often).*

The infinite word σ *is accepted by* $\mathcal{G}'$ iff *there exists an accepting run of* $\mathcal{G}'$ *over* σ. *The* language *of* $\mathcal{G}'$ *is the set* $\mathscr{L}(\mathcal{G}') \subseteq \Sigma^\omega$ *of infinite words it accepts.*

TGBA and BA have the same expressive power: any TGBA can be converted into a language-equivalent BA and vice-versa [27, 49]. The process of converting a TGBA into a BA is also called *degeneralization* and is similar to the "GBA to BA transformation" presented in section 2.4.2.1.

Figure 2.7 shows the same properties as Figure 2.4 and Figure 2.5, but expressed as TGBA. A run in these examples is accepted if it visits infinitely often all acceptance conditions (represented by colored dots $\circ$ and $\bullet$ on transitions).

(a) The TGBA of Figure 2.7a recognizes the runs that contain infinitely often a and infinitely often b. An accepting run in this TGBA has to visit infinitely often the two acceptance conditions indicated by $\circ$ and $\bullet$. Therefore, it must explore infinitely often the transitions where a is true (i.e., transitions labeled by ab or $a\bar{b}$) and infinitely often the transitions where b is true (i.e., transitions labeled by ab or $\bar{a}b$). As an illustration of the degeneralization process, the BA from Figure 2.4a was built by degeneralizing the TGBA from Figure 2.7a. In the worst case, a TGBA with s states and n acceptance conditions will be degeneralized into a BA with $s \times (n+1)$ states. The worst case of the degeneralization occurred here, since the TGBA with 1 state and n acceptance conditions was degeneralized into a BA with $n+1$ states. It is known that no BA with less than $n+1$ states can accept the property $\bigwedge_{i=1}^{n} \mathsf{GF}\,p_i$ so this BA is optimal [22] in terms of number of states.

(b) The property $a\,\mathsf{U}\,\mathsf{G}b$ is easier to express: the TGBA of Figure 2.7b is the same as the BA of Figure. 2.4b, with the acceptance condition $\bullet$ moved on transitions. More generally, a BA can be seen as a TGBA, by simply marking as accepting the transitions leaving the accepting states, without adding states nor transitions. Algorithms that input TGBA can therefore be easily adapted to process BA or GBA.

2.4.4 Determinization of Büchi Automata

Definition 18. *A Büchi automaton is* deterministic iff *each accepted infinite word of* Σ^ω *is recognized by an unique accepting run of* δ^ω.

Formally, a Büchi automaton $\langle Q, I, \delta, \mathcal{F} \rangle$ is deterministic *iff* $|I| = 1$ and $\forall (q, \ell) \in Q \times \Sigma$, $\left|\{q' \in Q \mid (q, \ell, q') \in \delta\}\right| \leq 1$ (i.e., the outgoing transitions of each state are labeled with different valuations).

As an illustration, the Büchi automaton of Figure 2.4a is a deterministic BA. However, the Büchi automaton of Figure 2.4b is a non-deterministic BA.

The definition 18 of deterministic Büchi Automata is valid for the three variants of Büchi Automata: BA, GBA and TGBA.

The determinization operation consists in the transformation of a non-deterministic Büchi automaton into a deterministic one. However, there are non-deterministic Büchi automata that cannot be determinized. For instance, the non-deterministic Büchi automata of Figure 2.8 that recognize the LTL property $\varphi = \mathsf{FG}a$, cannot be transformed into an equivalent deterministic Büchi automaton.

The classical powerset construction [72] used to determinize finite automata does not work for any Büchi automata (it works only for a restricted class of Büchi automata [30]).

The powerset construction uses subsets of states of the non-deterministic automaton, as states of the deterministic automaton. The example of Figure 2.9 shows that this construction does not work for the Büchi automata of $\varphi = \mathsf{FG}a$. The constructed automaton of Figure 2.9b accepts the infinite word $a;\bar{a};a;\bar{a};\ldots$ (alternating a and $\bar{a}$ infinitely) and therefore does not recognize the same language as the original automaton of Figure 2.9a.

Thus, the determinization of Büchi automata requires the use of other types of ω-automata, such as Muller or Rabin automata [76].

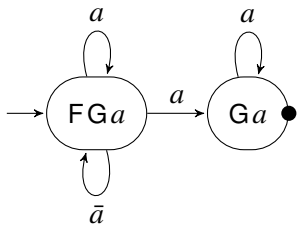
The history of determinizing Büchi automata began in 1963 with the first attempt of determinization due to Muller [65] that appears to be faulty. In 1966, McNaughton [63] proved that any non-deterministic Büchi automaton can be converted into a deterministic Muller automaton. This conversion involves a doubly exponential blow-up in the size of the original Büchi automaton ($2^{2^{O(n)}}$).

Proposed in 1988, the Safra's construction [76] transforms a non-deterministic Büchi automaton with n states into an equivalent deterministic Muller or Rabin automaton with $2^{O(n\log(n))}$ states (optimal for Rabin automata).

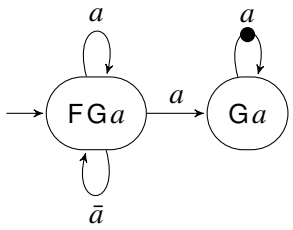
To conclude this section, it is important to note that although there are LTL formulas that are not convertible into Büchi automaton and the cost to obtain a deterministic automaton remains very high, we will see in the following that even the use of automata that are not "completely" deterministic but having a high degree of determinism can improve the performance of model checking.

2.4.5 Bisimulation Reduction of Büchi Automata

We say that two states are bisimilar if the automaton can accept the same runs starting from either of these states (this implies that two bisimilar states recognize the same language). Büchi Au-

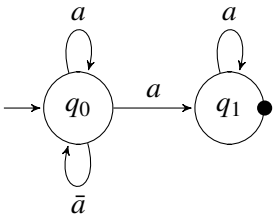


(a) BA/GBA for $\varphi = \mathsf{FG}a$

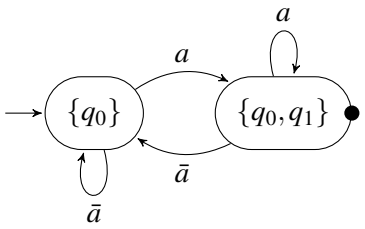


(b) TGBA for $\varphi = \mathsf{FG}a$

Figure 2.8: Examples of non-deterministic Büchi Automata: BA/GBA and TGBA for the LTL property $\varphi = \mathsf{FG}a$. Acceptance states/transitions are indicated by ●.



(a) A BA $\mathcal{B}$ for $\varphi = \mathsf{FG}a$ before determinization



(b) After faulty determinization of $\mathcal{B}$ by powerset construction

Figure 2.9: Büchi automata determinization: The classical powerset construction does not work for $\varphi = \mathsf{FG}a$ (A deterministic Büchi automaton for $\mathsf{FG}a$ does not exist).

tomata can be simplified by merging bisimilar states. This bisimulation reduction can be achieved using a partition refinement algorithm [e.g., 14, 40, 13, 91, 88].

The main idea of this algorithm is to split the set of states of the automaton into equivalence classes according to the equivalence relation of bisimilarity, then the states of each equivalence class are merged into a single state.

One way to compute the equivalence classes is to use the notion of signature [13]. A signature $sig(q)$ can be viewed as a "fingerprint" of each state q that encodes the outgoing transitions of q. Two states q and q' having different signatures are not bisimilar and therefore belong to different classes C and C'.

- For BA $\langle Q, I, \delta, \mathcal{F} \rangle$, the signature of a state q with respect to a partition Π, is the set of pairs (valuation, destination class $C \in \Pi$) of each outgoing transition from q: i.e., $sig_\Pi(q) = \{(\ell, C) \mid \exists q' \in C \in \Pi, (q, \ell, q') \in \delta\}$.

- For TGBA, the signature also includes the acceptance conditions of transitions: $sig_\Pi(q) = \{(\ell, F, C) \mid \exists q' \in C \in \Pi, (q, \ell, F, q') \in \delta\}$.

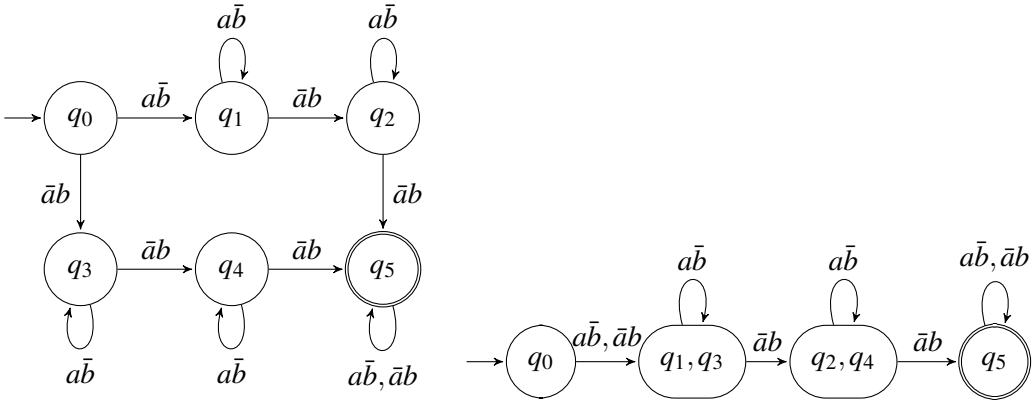
- In the case of a GBA with $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$, the signature must includes the acceptance conditions on states: $sig_\Pi(q) = \{(\ell, F, C) \mid \exists q' \in C \in \Pi, (q, \ell, F, q') \in \delta\}$, where $F = \{i \mid q \in F_i\}$.

The sets representing the signatures can be encoded symbolically [91, 14], BDDs (Binary Decision Diagrams) are used to implement the signatures of the bisimulation reduction of the different automata used in this work.

The basic idea of the bisimulation algorithm is to build a partition of the equivalence classes by refining an initial partition Π_0. This can be implemented by the following iterative procedure :

- Initialization:
  - For a standard Büchi Automaton BA, we set up with two equivalence classes $\mathcal{F}$ and $Q \setminus \mathcal{F}$: $\Pi_0 = \{\mathcal{F}, Q \setminus \mathcal{F}\}$.
  - For a TGBA or a GBA, $\Pi_0 = \{Q\}$.
- Iterate until a fixpoint is reached (i.e., $\Pi_k = \Pi_{k+1}$):
  - For each pair of states q and q' belonging to the same class C of the partition Π_k, if $sig_{\Pi_k}(q) \neq sig_{\Pi_k}(q')$ then q and q' are put into two different classes C' and C'' of the new partition Π_{k+1}

When this procedure stops, the states of each class have the same signature in the obtained partition. The termination of this algorithm is guaranteed because the number of states is finite and therefore the number of partition too. This naive implementation has a quadratic complexity (due to the comparison of all the pairs of states signatures, for each iteration). The optimization [88] reduces this complexity to $O(m\log(n))$, where n is the number of states and m is the number of transitions.



(a) A BA $\mathcal{B}$ before bisimulation reduction

(b) $\mathcal{B}$ after bisimulation reduction

Figure 2.10: An example of Bisimulation Reduction of BA

As an illustration, Figure 2.10a and Figure 2.10b show an example of BA respectively before and after applying the bisimulation procedure:

The successive iterations to compute the equivalence classes from the automaton of Figure 2.10a are:

- $\Pi_0 = \{\{q_0, q_1, q_2, q_3, q_4\}, \{q_5\}\}$, because q_5 is the unique accepting state.
- $\Pi_1 = \{\{q_0, q_1, q_3\}, \{q_2, q_4\}, \{q_5\}\}$, because in Π_0, we have $\left(\bar{a}b, \{q_5\}\right) \in sig_{\Pi_0}(q_2) = sig_{\Pi_0}(q_4)$ and $\left(\bar{a}b, \{q_5\}\right) \notin sig_{\Pi_0}(q_0) = sig_{\Pi_0}(q_1) = sig_{\Pi_0}(q_2)$.

- $\Pi_2 = \{\{q_0\},\{q_1,q_3\},\{q_2,q_4\},\{q_5\}\}$, because in Π_1, we have $\big(\bar{a}b,\{q_2,q_4\}\big) \in sig_{\Pi_1}(q_1) = sig_{\Pi_1}(q_3)$ and $\big(\bar{a}b,\{q_2,q_4\}\big) \notin sig_{\Pi_1}(q_0)$.
- $\Pi_3 = \Pi_2$ the procedure stops.

The automaton obtained after this partitions refinement is shown in Figure 2.10b.

Simulation [40, 41, 44, 81, 11] reduction is a generalization and an improvement of the bisimulation reduction. Simulation relation between two states q and q' is based on the inclusion of the sets of infinite runs starting from q and q', instead of equality between these sets of infinite runs as in bisimulation relation between q and q'. Recently, [3] proposed a simulation reduction that improves the determinism of the resulting automaton.

2.4.6 Converting LTL formula into Büchi Automaton

Any LTL formula φ can be converted into a Büchi Automaton whose language is the set of executions that satisfy φ. This conversion is central to the process of model checking, thus a substantial number of research has been conducted in this area. Several algorithms have been proposed to translate an LTL formula into a BA, GBA or TGBA.

The first translation algorithm was proposed by Wolper [93, 92], it allows to convert an LTL formula φ into an automaton whose size is always equal to $2^{O(|\varphi|)}$ states (exponential in the size of φ). In [92], the algorithm proposed is dedicated to LTL and produces a GBA. Despite its exponential complexity, this algorithm is easier to understand than others. In the following, we present a concise variant of this algorithm proposed in [66].

Using the equivalences between the LTL operators (Section 2.3.1.3), we can assume that the LTL formula φ to translate, is only composed of the operators $\neg$, $\vee$, X and U. There are several optimizations based on rewriting rules[81, 40, 11] that simplify the LTL formula before translating it into an automaton.

We note A_φ the GBA constructed by the following translation. The computation of the states of A_φ is based on the set $cl(\varphi)$, the *closure* of φ containing all the subformulas of φ and $\mathsf{X}(\phi_1 \mathsf{U} \phi_2)$ for each subformula $(\phi_1 \mathsf{U} \phi_2)$ of φ, it is in addition closed by the operator $\neg$.

Definition 19 (Closure)**.** *Let $sub(\varphi)$ be the set of all subformulas of φ. The* closure *of φ is the smallest set $cl(\varphi)$ satisfying:*

- $sub(\varphi) \subseteq cl(\varphi)$,
- $\{\mathsf{X}(\phi_1 \mathsf{U} \phi_2) \mid (\phi_1 \mathsf{U} \phi_2) \in sub(\varphi)\} \subseteq cl(\varphi)$,
- $\forall \phi \in cl(\varphi), (\neg\phi) \in cl(\varphi)$ *(With reduction of redundancies like $\phi = \neg\neg\phi$ and $\neg\mathsf{X}\phi = \mathsf{X}\neg\phi$).*

For example, $cl(\mathsf{F}\,p) = cl(\top \mathsf{U}\, p) = \{p, \bar{p}, (\top \mathsf{U}\, p), \mathsf{X}(\top \mathsf{U}\, p), \neg(\top \mathsf{U}\, p), \mathsf{X}\neg(\top \mathsf{U}\, p)\}$. It is easy to deduce that $|cl(\varphi)| \in O\big(|\varphi|\big)$.

Each state of A_φ is labeled by an *atom*, where an atom is a subset of $cl(\varphi)$ that satisfies the following three consistency rules:

Definition 20 (Atom)**.** *An atom α of φ is a subset of $cl(\varphi)$, satisfying the following rules:*

- *Logical consistency rules (α is maximal and does not contain logical contradictions):*
  - $\forall \phi \in cl(\varphi), \phi \in \alpha \Leftrightarrow \neg\phi \notin \alpha$*, this rule also implies that α is maximal, i.e., $\forall \phi \in cl(\varphi)$, either $\phi \in \alpha$ or $\neg\phi \in \alpha$,*

– $\forall(\phi_1 \vee \phi_2) \in cl(\varphi), (\phi_1 \vee \phi_2) \in \alpha \Leftrightarrow (\phi_1 \in \alpha)$ *or* $(\phi_2 \in \alpha)$,

- *Temporal consistency rule:*
  $\forall(\phi_1 \mathsf{U} \phi_2) \in cl(\varphi), (\phi_1 \mathsf{U} \phi_2) \in \alpha \Leftrightarrow (\phi_2 \in \alpha)$ *or* $\big((\phi_1 \in \alpha)$ *and* $\mathsf{X}(\phi_1 \mathsf{U} \phi_2) \in \alpha\big)$.
  This temporal consistency rule is deduced from the expansion law stating that: $\phi_1 \mathsf{U} \phi_2 = \phi_2 \vee (\phi_1 \wedge \mathsf{X}(\phi_1 \mathsf{U} \phi_2))$.

Let $Atoms(\varphi)$ denotes the set of atoms of φ, we have $Atoms(\varphi) \subseteq 2^{cl(\varphi)}$ and the number of states of A_φ is $|A_\varphi| = |Atoms(\varphi)| \leq |2^{cl(\varphi)}| \in O(2^{|\varphi|})$.

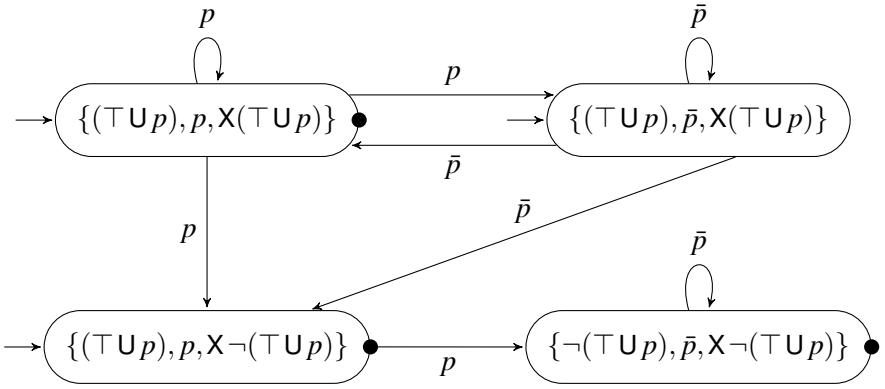


Figure 2.11: A GBA A_φ constructed form $(\varphi = \mathsf{F}\, p)$ by the algorithm of Definition 21.

As an illustration, we use the above rules to compute the atoms for the translation of the LTL formula $\varphi = \mathsf{F}\, p = (\top \mathsf{U}\, p)$, we obtain the set $Atom(\mathsf{F}\, p)$ composed of the four atoms labeling the states of the GBA $A_{(\mathsf{F}\, p)}$ shown in Figure 2.11. This set of atoms is obtained by the following reasoning:
In order to satisfy the temporal consistency rule, the atoms that contain the sub-formula $(\top \mathsf{U}\, p)$ must also contain p or $\mathsf{X}(\top \mathsf{U}\, p)$. Thus, we obtain three atoms containing $(\top \mathsf{U}\, p)$:

- $\{(\top \mathsf{U}\, p), p, \mathsf{X}(\top \mathsf{U}\, p)\} \in Atom(\mathsf{F}\, p)$
- $\{(\top \mathsf{U}\, p), \bar{p}, \mathsf{X}(\top \mathsf{U}\, p)\} \in Atom(\mathsf{F}\, p)$
- $\{(\top \mathsf{U}\, p), p, \mathsf{X}\neg(\top \mathsf{U}\, p)\} \in Atom(\mathsf{F}\, p)$

On the other hand, the atoms that do not contain $(\top \mathsf{U}\, p)$ must contain $\neg(\top \mathsf{U}\, p)$ because they have to be maximal (the first rule), and they can only contain $\bar{p}$ and $\neg\mathsf{X}(\top \mathsf{U}\, p)$ to satisfy the temporal consistency rule. We obtain an unique atom that contain $\neg(\top \mathsf{U}\, p)$: $\{\neg(\top \mathsf{U}\, p), \bar{p}, \mathsf{X}\neg(\top \mathsf{U}\, p)\} \in Atom(\mathsf{F}\, p)$.

Once the atoms labeling the states computed, we can build the GBA A_φ according to the following definition:

Definition 21 (From φ to A_φ)**.** *Given an LTL formula φ over AP, A GBA over the alphabet $\Sigma = 2^{AP}$ that accepts the same language as φ, is a tuple $A_\varphi = \langle Q, I, \delta, \mathcal{F} \rangle$ where:*

- $Q = Atoms(\varphi)$,
- $I = \{\alpha \in Atoms(\varphi) \mid \varphi \in \alpha\}$,

- $\delta = \{(\alpha, \alpha \cap AP, \alpha') \in Q \times \Sigma \times Q \mid \forall \mathsf{X}\phi \in cl(\varphi), (\mathsf{X}\phi \in \alpha) \Leftrightarrow (\phi \in \alpha')\}$,
- $\mathcal{F} = \{\mathcal{F}_{\phi_1 \mathsf{U} \phi_2} \mid (\phi_1 \mathsf{U} \phi_2) \in cl(\varphi)\}$, *with* $\mathcal{F}_{\phi_1 \mathsf{U} \phi_2} = \{\alpha \in Q \mid \phi_2 \in \alpha \vee (\phi_1 \mathsf{U} \phi_2) \notin \alpha\}$

This construction of A_φ is illustrated in Figure 2.11 presenting the GBA $A_{(\mathsf{F}\, p)}$ obtained from the LTL formula $\varphi = \mathsf{F}\, p = (\top \mathsf{U}\, p)$. Intuitively, when the constructed automaton A_φ reads a word $\sigma = \ell_0 \ell_1 \ell_2 \ldots$ by exploring a sequence of states $\alpha_0 \alpha_1 \alpha_2 \ldots$, in each step i, the "sub-"word $w_i = \ell_i \ell_{i+1} \ell_{i+1} \ldots$ satisfies all the formulas labeling the state α_i. In addition, w_i does not satisfy any other state of $Q \setminus \alpha_i$ because w_i does not satisfy any formulas of $cl(\varphi) \setminus \alpha_i$ by definition of the atom α_i. The formulas of the form $\mathsf{X}\phi$ are verified in the next step by the transition relation δ, because δ is constructed according to the equivalence $(\mathsf{X}\phi \in \alpha_i) \Leftrightarrow (\phi \in \alpha_{i+1})$ where $(\alpha_i, \alpha_i \cap AP, \alpha_{i+1}) \in \delta$ (Definition 21).

The automata produced by translation algorithms often contain redundant states, which can be eliminated using the bisimulation/simulation rOneductions presented in Section 2.4.5.

As we mentioned earlier, the algorithm presented above is not optimal, it produces a GBA composed of 4 states (Figure 2.11) for $\varphi = \mathsf{F}\, p$, while this LTL formula can be converted into a GBA having only 2 states.

Many other more efficient algorithms are proposed to translate an LTL formula into the different variants of Büchi Automata. One common way to obtain a BA from an LTL formula is to first translate the formula into some Generalized Büchi Automata with multiple acceptance conditions (GBA, TGBA,...) and then to *degeneralize* this automaton to obtain a single acceptance condition. Alternatives include the translation of the property into a *state-based* [48] generalized automaton which can then also be degeneralized, or the translation of the property into an alternating Büchi automaton that is then converted into a BA using the construction proposed by [45].

The degeneralization process can increase the size of the Büchi automaton (see Section 2.4.2.1). In addition, several model checking procedures supports generalized (i.e., multiple) acceptance conditions, making such degeneralization unnecessary and even costly [29]. Moving the acceptance conditions from the states (GBA) to the transitions (TGBA) also reduces the size of the property automaton [27, 49].

Several algorithms exist to translate an LTL formula into a TGBA [49, 27, 28, 84, 45, 2]. The one we use in the experimentations presented in this thesis is based on Couvreur's LTL translation algorithm [27]. The next chapter presents the results of an experimentation that compares different model checking approaches, including those using BA and TGBA.

Most of the works that tried to improve model checking, have focused on translating the LTL formula into the smallest possible automaton. However, [78] claims (without giving evidence) that model checking can be improved by generating more deterministic automata (determinization of Büchi automata was discussed in section 2.4.4). According to [3, 10], it is not yet clear when "more deterministic" automaton should be preferred to a small one.

2.5 Explicit LTL Model checking using Büchi Automata

The explicit model checking enumerates the states of the state-spaces of both the model and the LTL formula. These explicit state-spaces and their synchronous product are traditionally repre-

sented using variants of ω-automata. In this automata-based approach to model checking of an LTL formula φ on the model M, there are two important algorithms (represented in Figure 2.1):

1. The translation of the negation of the LTL formula $\neg\varphi$ into (one variant of) a Büchi Automaton $A_{\neg\varphi}$ (Translation of LTL properties into automata has already been presented in the previous section).

2. The emptiness check of the product $\mathcal{K}_M \otimes A_{\neg\varphi}$ where $\mathcal{K}_M$ is the Kripke structure representing the state-space of the model M. The language of this product automaton $\mathscr{L}(\mathcal{K}_M \otimes A_{\neg\varphi})$ is equal to $\mathscr{L}(\mathcal{K}_M) \cap \mathscr{L}(A_{\neg\varphi})$, i.e., the set of executions of $\mathcal{K}_M$ invalidating φ.

The goal of the emptiness check algorithm is to determine if the product automaton $\mathcal{K}_M \otimes A_{\neg\varphi}$ accepts an execution or not.

In other words, it checks if the language of the product automaton is empty or not. If it is empty, then there is no execution of the model M that invalidates the property φ and therefore $M \models \varphi$. Otherwise, there is an execution of M that invalidates φ and this execution is reported as a counterexample.

More formally, the automata-theoretic approach is based on the following equivalences:

$$\begin{aligned}
\mathscr{L}(\mathcal{K}_M \otimes A_{\neg\varphi}) = \emptyset &\iff \mathscr{L}(\mathcal{K}_M) \cap \mathscr{L}(A_{\neg\varphi}) = \emptyset \\
&\iff \mathscr{L}(\mathcal{K}_M) \cap \overline{\mathscr{L}(A_{\varphi})} = \emptyset \\
&\iff \mathscr{L}(\mathcal{K}_M) \subseteq \mathscr{L}(A_{\varphi}) \\
&\iff M \models \varphi
\end{aligned}$$

The emptiness check algorithms dealing with TGBA also work for GBA and BA, because a GBA or a BA can be seen as a TGBA by pushing the acceptance conditions on the transitions leaving accepting states (with multiple acceptance conditions in the case of GBA). For this reason, this section only focuses on TGBA.

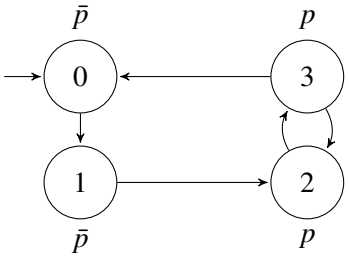
2.5.1 Synchronous Product

The product of a TGBA with a Kripke structure is a TGBA whose language is the intersection of both languages.

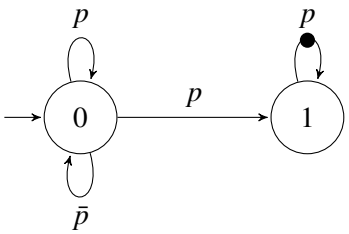
Definition 22 (Synchronous Product of a Kripke structure and a Büchi automaton)**.** *For a Kripke structure* $\mathcal{K} = \langle S, S_0, \mathcal{R}, l \rangle$ *and a TGBA*
$\mathcal{A} = \langle Q, I, \delta, \mathcal{F} \rangle$ *the product* $\mathcal{K} \otimes \mathcal{A}$ *is the TGBA* $\langle S_\otimes, I_\otimes, \delta_\otimes, \mathcal{F} \rangle$ *where*

- $S_\otimes = S \times Q$,
- $I_\otimes = S_0 \times I$,
- $\delta_\otimes = \{((s,q), \ell, F, (s',q')) \mid (s,s') \in \mathcal{R}, (q, \ell, F, q') \in \delta, l(s) = \ell\}$

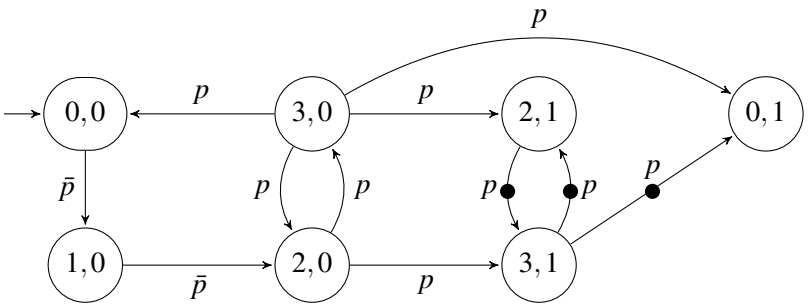
Property 1. *We have* $\mathscr{L}(\mathcal{K} \otimes \mathcal{A}) = \mathscr{L}(\mathcal{K}) \cap \mathscr{L}(\mathcal{A})$ *by construction.*



(a) A Kripke structure $\mathcal{K}$



(b) A TGBA $\mathcal{A}$ for $\mathsf{FG}\,p$



(c) Synchronous Product $\mathcal{K} \otimes \mathcal{A}$

Figure 2.12: Example of a Synchronous Product $\mathcal{K} \otimes \mathcal{A}$ between a Kripke structure $\mathcal{K}$ and a TGBA $\mathcal{A}$ recognizing the LTL formula $\mathsf{FG}\,p$, with acceptance conditions indicated by the black dot ●.

Because the set of all states of the product is defined as $S_{\otimes} = S \times Q$, the maximum size (in term of number of reachable states) of the product automaton is equal to $|S| \times |Q|$.

Figure 2.12 shows an example of a Synchronous Product between a Kripke structure $\mathcal{K}$ and a TGBA (BA) $\mathcal{A}$ recognizing the LTL formula $\mathsf{F}\,\mathsf{G}\,p$. Each state of $\mathcal{K}$ is numbered and labeled with the set of atomic propositions (of $\Sigma = \{p\}$) that hold in this state. In the TGBA representing the product $\mathcal{K} \otimes \mathcal{A}$, the states are labeled with a pairs of the form "($\mathcal{K}$ state, $\mathcal{A}$ state)".

Let us recall that since a BA or a GBA can be seen as a TGBA with accepting conditions moved from accepting states onto their outgoing transitions, the above product construction using TGBA can easily be adapted to a product between a Kripke structure and a BA or a GBA.

2.5.2 On-the-fly Emptiness check algorithms

Testing a TGBA for emptiness amounts to the search of an accepting cycle that contains at least one occurrence of each acceptance condition. This can be done in different ways: either with a variation of Tarjan or Dijkstra algorithm [27] or using several Nested Depth-First Searches (NDFS) to save some memory [84]. In NDFS algorithms, a first Depth-First Search (DFS) is performed until it reaches an accepting state s, then a second DFS is performed from s trying to return to s. [29, 46] argued that NDFS algorithms are slower than SCC-based algorithms, so in the following, we will use in our experiments Couvreur's SCC-based emptiness check algorithm [27] because it only performs a single DFS, and its complexity does not depend on the number of acceptance conditions (while NDFS may need to perform many nested DFS in the case of multiple acceptance conditions (GBA or TGBA)). This Couvreur's algorithm will be described in detail below. The comparison of the different emptiness check algorithms has raised many studies [47, 77, 29, 73].

The product automaton that has to be explored during the emptiness check is generally very large, its size can reach the value obtained by multiplying the the sizes of the model and formula automata, which are synchronized to build this product. Therefore, building the entire product must be avoided. "On-the-fly" emptiness check algorithms allow the product automaton to be constructed lazily during its exploration. These on-the-fly algorithms are more efficient because they stop as soon as they find a counterexample and therefore possibly before building the entire product, thereby reducing the amount of memory and time used by the emptiness check.

In this work, we focus on SCC-based "On-the-fly" emptiness checks. Algorithm 1 presented below is an iterative version of the Couvreur's SCC-based algorithm [27].

Algorithm. 1 computes on-the-fly the Maximal Strongly Connected Components (MSCCs) of the TGBA representing the product $\mathcal{K} \otimes \mathcal{A}$: it performs a Depth-First Search (DFS) for SCC detection and then merges the SCCs belonging to the same maximal SCC into a single SCC. After each merge, if the union of all acceptance conditions occurring in the merged SCC is equal to $\mathcal{F}$ (line 16), then an accepting run (i.e., a counterexample) is found (line 16) and the $\mathscr{L}(\mathcal{K} \otimes \mathcal{A})$ is not empty. *todo* is the DFS stack. It is used by the procedure `DFSpush` to push the states of the current DFS path and the set of their successors that have not yet been visited. H maps each visited state to its rank in the DFS order, and $H[s] = 0$ indicates that s is a dead state (i.e., s belongs to a maximal SCC that has been fully explored). Figure 2.14 illustrates a run of this algorithm on a small example.

The *SCC* stack stores a chain of partial SCCs found during the DFS. For each SCC the attribute

```
1  Input: A product TGBA 𝒦⊗𝒜 = ⟨𝒮⊗, I⊗, δ⊗, ℱ⟩
2  Result: ⊤ if and only if ℒ(𝒦⊗𝒜) = ∅
3  Data: todo: stack of ⟨state ∈ 𝒮, succ ⊆ δ⊗⟩
         SCC: stack of ⟨root ∈ ℕ, la ⊆ ℱ, acc ⊆ ℱ, rem ⊆ 𝒮⟩
         H: map of 𝒮 ↦ ℕ
         max ← 0
4  begin
5      foreach s⁰ ∈ I⊗ do
6          DFSpush(∅, s⁰)
7          while ¬todo.empty() do
8              if todo.top().succ = ∅ then
9                  DFSpop()
10             else
11                 pick one ⟨s, _, a, d⟩ off todo.top().succ
12                 if d ∉ H then
13                     DFSpush(a, d)
14                 else if H[d] > 0 then
15                     merge(a, H[d])
16                     if SCC.top().acc = ℱ then
17                         return ⊥
18     return ⊤
19 DFSpush(la ⊆ ℱ, s ∈ 𝒮⊗)
20     max ← max + 1
21     H[s] ← max
22     SCC.push(⟨max, la, ∅, ∅⟩)
23     todo.push(⟨s, {⟨q, l, a, d⟩ ∈ δ⊗ | q = s}⟩)
24 DFSpop()
25     ⟨s, _⟩ ← todo.pop()
26     SCC.top().rem.insert(s)
27     if H[s] = SCC.top().root then
28         foreach s ∈ SCC.top().rem do
29             H[s] ← 0
30         SCC.pop()
31 merge(la ⊆ ℱ, t ∈ ℕ)
32     r ← ∅
33     acc ← la
34     while t < SCC.top().root do
35         acc ← acc ∪ SCC.top().acc
                     ∪ SCC.top().la
36         r ← r ∪ SCC.top().rem
37         SCC.pop()
38     SCC.top().acc ← SCC.top().acc ∪ acc
39     SCC.top().rem ← SCC.top().rem ∪ r
```

Algorithm 1: Emptiness check algorithm for TGBA.

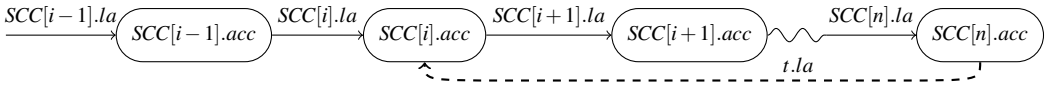


Figure 2.13: *SCC* stack: the use of the SCCs fields *la* and *acc*.

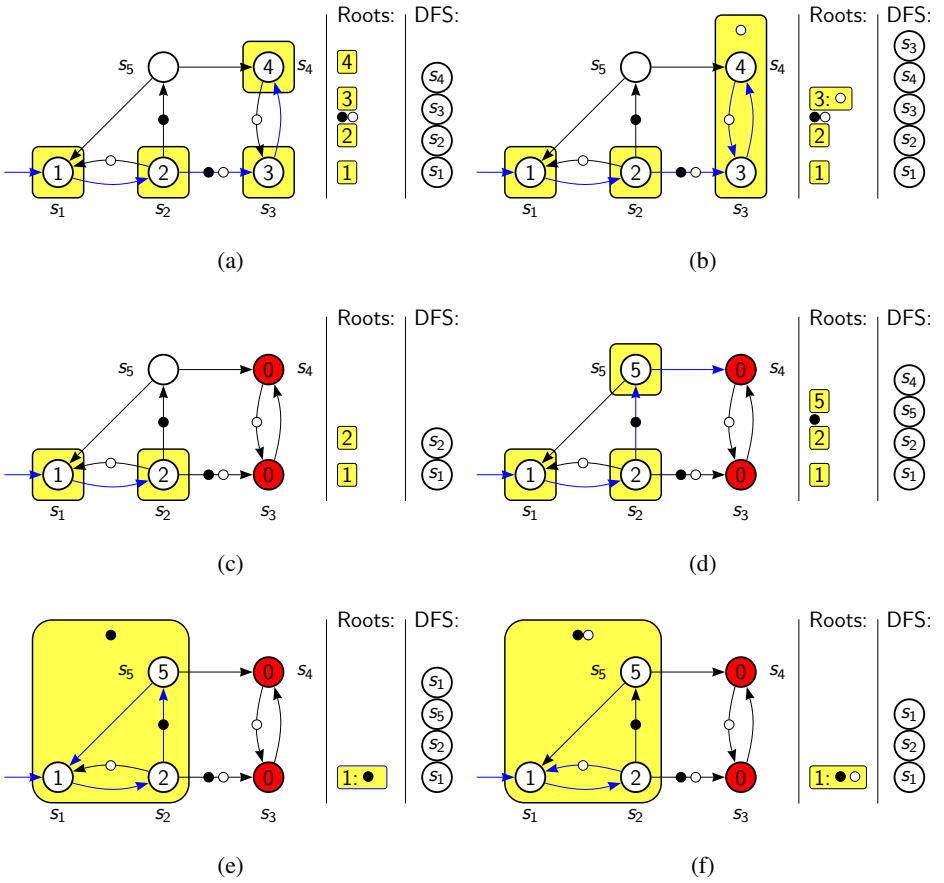


Figure 2.14: Six intermediate steps in a run of algorithm 1. The states $s_1, \ldots, s_5$ are labeled by their value in H. The stack of roots of SCCs (the *root* stack in the algorithm) and the DFS search stack (induced by the recursive calls to `DFSpush()`) are displayed on the side. An interpretation of the SCC stack in term of SCCs is given as yellow blobs on the automaton.
(a) Initially the algorithm performs a DFS search by declaring each newly encountered state as a trivial SCC. (b) When the transition from s_4 to s_3 is processed, the algorithm detects that $H[s_3] \neq 0$ which means the transition creates a cycle and all SCCs between s_4 and s_3 are merged. (c) When the DFS exits the non-accepting $\{s_3, s_4\}$ SCC, it marks all its states as dead ($H[s] = 0$). (d) When the DFS attempt to visit a dead state, it ignores it. (e) Visiting the transition from s_5 to s_1 will merge three SCCs into one, but it does not yet appear to be accepting because the white accepting condition (○)has not been seen. (f) Finally visiting the transition from s_2 back to s_1 will add the white acceptance condition to the current SCC, and the algorithm will stop immediately because it has found an SCC labeled by all acceptance conditions.

root is the DFS rank (H) of the first state of the SCC, *acc* is the set of all acceptance conditions belonging to the SCC, *la* is the acceptance conditions of the transition between the previous and the current SCC, and *rem* contains the fully explored states of the SCC. Figure 2.13 shows how *acc* and *la* are used in the SCC search stack.

1. The algorithm begins by pushing in *SCC* each state visited for the first time (line 4), as a trivial SCC with an empty *acc* set (line 22).

2. Then, when the DFS explores a transition t between two states s and d, if d is in the SCC stack (line 11), therefore t closes a cycle passing through s and d in the product automaton. This cycle "strongly connects" all SCCs pushed in the *SCC* stack between $SCC[i]$ and $SCC[n]$: the two SCCs that respectively contains the states d and s ($SCC[n]$ is the top of the *SCC* stack).

3. All the SCCs between $SCC[i]$ and $SCC[n]$ are merged (line 15) into $SCC[i]$. The merge of acceptance conditions is illustrated by Figure 2.13: a "back" transition t is found between $SCC[n]$ and $SCC[i]$, therefore the latest SCCs (from i to n) are merged.

4. The acceptance conditions of the merged SCC is equal to the union of $SCC[i].acc \cup SCC[i+1].la \cup SCC[i+1].acc \cup \cdots \cup SCC[n].la \cup SCC[n].acc \cup t.la$. If this union is equal to $\mathcal{F}$, then the merged SCC is accepting and the algorithm return *false* (line 16): the product is not empty.

Figure 2.14 illustrates the successive steps of a run of algorithm 1.

2.5.3 Complexity

In the automata-theoretic approach to model checking of a Kripke structure $\mathcal{K}_M$ against an LTL formula φ, the upper bound of the time (and space) complexity is in $O(|\mathcal{K}_M| \times 2^{|\varphi|})$, because φ can be converted into Büchi automaton $A_{\neg\varphi}$ whose size and time of construction is in $O(2^{|\varphi|})$, and the emptiness check is linear with respect to the size of the product automaton $\mathcal{K}_M \otimes A_{\neg\varphi}$. This linear complexity is for instance the complexity of Algorithm 1, which has the same complexity as the algorithm for finding the maximum strong components in a directed graph of Dijkstra [32, 33].

Thus, the upper bound of the model checking complexity is directly deduced from the inequality: $|\mathcal{K}_M \otimes A_{\neg\varphi}| \leq |\mathcal{K}_M| \times |A_{\neg\varphi}|$.

2.6 Conclusion

Automata-theoretic approach is traditionally used for the explicit LTL model checking. In this approach, a Kripke structure $\mathcal{K}_M$ is used to represent the state-space of the model M, and the property to be checked is expressed as an LTL formula φ, then its negation is converted into a Büchi automaton $A_{\neg\varphi}$. The third operation is the synchronization between $\mathcal{K}_M$ and $A_{\neg\varphi}$. This constructs a product automaton $\mathcal{K}_M \otimes A_{\neg\varphi}$ whose language, $\mathcal{L}(\mathcal{K}_M) \cap \mathcal{L}(A_{\neg\varphi})$, is the set of executions of M invalidating φ. The last operation is the emptiness check algorithm that explores the product to

tell whether it accepts or not an infinite word, i.e., a counterexample. The model M satisfies φ iff $\mathscr{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$.

The main problem of model checking is the well known state-space explosion problem. In particular, the performance of the automata-theoretic approach depends (in practice) on the size of the explored part during the emptiness check of the product automaton. This explored part itself depends on three parameters: the automaton $A_{\neg\varphi}$ obtained from the LTL formula φ to be checked, the Kripke structure $\mathcal{K}_M$ representing the state-space of the model M, and the emptiness check algorithm. The fact that this algorithm is performed on-the-fly, potentially avoids building the entire product automaton. Indeed, the states of this product that are not visited by the emptiness check are not generated at all.

In order to reduce the size of the product $\mathcal{K}_M \otimes A_{\neg\varphi}$, many works have attempted to reduce the size of $A_{\neg\varphi}$, either by improving the LTL translation (Section 2.4.6), or by proposing several reductions for the automaton produced by this translation (examples of these reductions are the bisimulation/simulation based reductions presented in Section 2.4.5). However, [78] claims that the size of the product automaton depends more on the "determinism degree" of $A_{\neg\varphi}$ rather than its size.

Another optimization that can reduce the size of $A_{\neg\varphi}$ is to simplify the LTL formula $\neg\varphi$ before translating it into an automaton, several rewriting rules have been proposed [81, 40, 11] to perform LTL simplifications.

According to [27, 49], moving the accepting conditions from the states (as in GBA) to the transitions (TGBA) also reduces the size $A_{\neg\varphi}$. In addition, any algorithm that translates LTL into a Büchi automaton has to deal with generalized Büchi acceptance conditions at some point, for instance the obtained automaton is a GBA or a TGBA, and the process of degeneralizing this generalized automaton to obtain a BA often increases its size (see Section 2.4.2.1). Several emptiness-check algorithms can deal with generalized Büchi acceptance conditions, making such a degeneralization unnecessary and even costly [29].

It is important to note that in this thesis, we only focus on optimizing the property automaton $A_{\neg\varphi}$. We do not consider the techniques whose aim is to reduce the model's state-space $\mathcal{K}_M$, such as the *partial order reduction* implemented in Spin tool [55]. Several partial order reduction techniques have been proposed, as the *stubborn sets* of Valmari [86], the *persistent sets* of Godefroid [50] and the *ample sets* of Peled [69]. However, these techniques require additional knowledge about the model. In addition, the state-space of the model is more difficult to manipulate than the property automaton, which is generally smaller.

In the next chapter, we will present the results of an experimentation that compares the performance of different automata-based approaches to model checking, including those using BA versus TGBA, and another kind of ω-automaton, called Testing Automata (TA), that is specific to represent stutter-invariant LTL formulas.

CHAPTER 3

Evaluation of the Testing Automata Approach

Contents

3.1 Introduction

The previous chapter presented the classical automata-theoretic approach to model checking based on Büchi automata. The main limitation of this approach is the large size of the product automaton $(\mathcal{K}_M \otimes A_{\neg\varphi})$ obtained by synchronizing the Kripke structure of the model $\mathcal{K}_M$ with the Büchi automaton $A_{\neg\varphi}$, which represents the negation of the LTL property to be checked.

Different variants of Büchi automata have been used with the automata-theoretic approach. In Spot [64, 35], the model checking library we used in the experiments presented in this thesis, LTL properties are represented using TGBA (i.e., the variant of Büchi automata with generalized Büchi acceptance conditions *on transitions* rather on states). Indeed, according to [36], it is preferable

to use TGBA for their conciseness. Unfortunately, having a smaller property automaton $A_{\neg\varphi}$ does not always imply a smaller product automaton $\mathcal{K}_M \otimes A_{\neg\varphi}$. Instead of targeting smaller property automata, some people have attempted to build *more deterministic* [78] ones; however even this does not guarantee the product to be smaller [3, 10].

This chapter focuses on another kind of ω-automaton called *Testing Automaton (TA)*. TA are a variant of an "extended" Büchi automata introduced by Hansen et al. [52]. Instead of observing the valuations on states or transitions, the TA transitions only record the changes between these valuations. In addition, TA are less expressive than Büchi automata since they are able to represent only stutter-invariant LTL properties. Also they are often a lot larger than their equivalent Büchi automaton, but their high degree of determinism [52] often leads to a smaller product size [46].

We first provide a detailed presentation of TA and their associated operations for model checking. Then, in order to evaluate the efficiency of LTL model checking using TA, we report and discuss the results of an experimental comparison of three kinds of ω-automata: classical Büchi Automata (BA), Transition-based Generalized Büchi automata (TGBA), and Testing Automata (TA) (this part completes our experiment presented in [5]). Our main motivation is to find the technique that seems the most suitable to check a given *stutter-invariant* property on a given model. This is of interest when a tool offers the choice of several techniques, which is the case for the Spot [64] tool in which I implemented the TA approach.

3.2 Stutter-invariant Languages

For any ω-automaton $\mathcal{A}$, we say that a language $\mathcal{L}(\mathcal{A})$ is *stutter-invariant* if the number of the successive repetitions of any letter of a word $\sigma \in \mathcal{L}(\mathcal{A})$ does not affect the membership of σ to $\mathcal{L}(\mathcal{A})$ [39]. In other words, $\mathcal{L}(\mathcal{A})$ is stutter-invariant *iff* for any finite sequence $u \in \Sigma^*$, any element $\ell \in \Sigma$, and any infinite sequence $v \in \Sigma^\omega$ we have $u\ell v \in \mathcal{L}(\mathcal{A}) \iff u\ell\ell v \in \mathcal{L}(\mathcal{A})$.

Two infinite words w_1 and w_2 are *stuttering equivalent iff* they are equal after removing all repeated letters.

Two languages $\mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{B})$ are *stuttering equivalent iff* any word of $\mathcal{L}(\mathcal{A})$ is stuttering equivalent to a word of $\mathcal{L}(\mathcal{B})$ and vice versa.

Given a stutter-invariant LTL formula φ (Definition 2.3.3) and an ω-automaton $\mathcal{A}_\varphi$ such that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$, then $\mathcal{L}(\mathcal{A}_\varphi)$ is a stutter-invariant language (we also say that $\mathcal{A}_\varphi$ is a stutter-invariant automaton).

3.3 Testing Automata (TA): a natural way to monitor the stuttering

Testing Automata were introduced by Hansen et al. [52] to represent stutter-invariant properties. While a Büchi automaton observes the value of the atomic propositions AP, the basic idea of TA is to only detect the *changes* in these values, making TA particularly suitable for stutter-invariant properties; if a valuation of AP does not change between two consecutive valuations of an execution, the TA can stay in the same state, this kind of transitions are called stuttering transitions. To detect infinite executions that end stuck in the same TA state because they are stuttering,

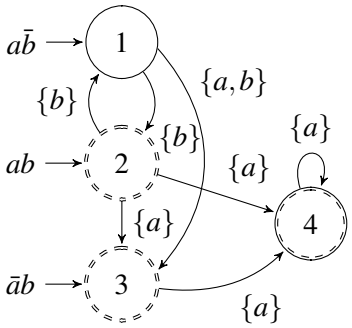


Figure 3.1: A TA $\mathcal{T}$ recognizing the LTL formula $a\mathsf{U}\mathsf{G}b$.

a new kind of accepting states is introduced: *livelock-accepting states*.

If A and B are two valuations, $A \oplus B$ denotes the symmetric set difference, i.e., the set of atomic propositions that differ (e.g., $a\bar{b} \oplus ab = \{b\}$). Technically, this is implemented with an XOR operation (also denoted by the symbol $\oplus$).

Definition 23 (TA)**.** *A TA over the alphabet $\Sigma = 2^{AP}$ is a tuple $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$, where:*

- *Q is a finite set of states,*
- *$I \subseteq Q$ is the set of initial states,*
- *$U : I \to 2^{\Sigma}$ is a function mapping each initial state to a set of valuations (set of possible initial configurations),*
- *$\delta \subseteq Q \times (\Sigma \setminus \emptyset) \times Q$ is the transition relation where each transition (s,k,d) is labeled by a* changeset*: $k \in \Sigma$ is interpreted as a non empty set of atomic propositions whose value must change between states s and d,*
- *$\mathcal{F} \subseteq Q$ is a set of Büchi-accepting states,*
- *$\mathcal{G} \subseteq Q$ is a set of livelock-accepting states.*

An infinite word $\sigma = \ell_0\ell_1\ell_2 \ldots \in \Sigma^{\omega}$ is accepted by $\mathcal{T}$ iff *there exists an infinite sequence $r = (q_0, \ell_0 \oplus \ell_1, q_1)(q_1, \ell_1 \oplus \ell_2, q_2) \ldots (q_i, \ell_i \oplus \ell_{i+1}, q_{i+1}) \ldots \in (Q \times \Sigma \times Q)^{\omega}$ such that:*

- *$q_0 \in I$ with $\ell_0 \in U(q_0)$,*
- *$\forall i \in \mathbb{N}$, either $(q_i, \ell_i \oplus \ell_{i+1}, q_{i+1}) \in \delta$ (the execution progresses in the TA), or $\ell_i = \ell_{i+1} \wedge q_i = q_{i+1}$ (the execution is stuttering and the TA does not progress),*
- *Either, $\forall i \in \mathbb{N}, (\exists j \geq i, \ell_j \neq \ell_{j+1}) \wedge (\exists l \geq i, q_l \in \mathcal{F})$ (the TA is progressing in a Büchi-accepting way), or, $\exists n \in \mathbb{N}, (q_n \in \mathcal{G} \wedge (\forall i \geq n, q_i = q_n \wedge \ell_i = \ell_n))$ (the sequence reaches a livelock-accepting state and then stays on that state because the execution is stuttering).*

The language accepted by $\mathcal{T}$ is the set $\mathscr{L}(\mathcal{T}) \subseteq \Sigma^{\omega}$ of executions it accepts.

To illustrate this definition, let us consider Figure 3.1, representing a TA $\mathcal{T}$ for $a\mathsf{U}\mathsf{G}b$. In this figure, the initial states 1, 2 and 3 are labeled respectively by the set of valuations $U(1) = \{a\bar{b}\}$, $U(2) = \{ab\}$ and $U(3) = \{\bar{a}b\}$. Each transition of $\mathcal{T}$ is labeled with a changeset over the set of atomic propositions $AP = \{a,b\}$. In a TA, states with a double enclosure belong to either $\mathcal{F}$ or $\mathcal{G}$: states in $\mathcal{F} \setminus \mathcal{G}$ have a double plain line, states in $\mathcal{G} \setminus \mathcal{F}$ have a double dashed line (states 2 and 3 of $\mathcal{T}$), and states in $\mathcal{F} \cap \mathcal{G}$ use a mixed dashed/plain style (state 4).

- The infinite word $ab;\bar{a}b;ab;\bar{a}b;ab;\bar{a}b;ab;\ldots$ is accepted by a Büchi accepting run of $\mathcal{T}$. Indeed, a run recognizing such word must start in state 2, then it always changes the value of a, so it has to take transitions labeled by $\{a\}$. For instance it could be the run $2 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4\cdots$ or the run $2 \xrightarrow{\{a\}} 3 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4\cdots$ Both visit the state $4 \in \mathcal{F}$ infinitely often, so they are Büchi accepting.
- The infinite word $ab;\bar{a}b;\bar{a}b;\bar{a}b;\ldots$ is accepted by a livelock accepting run of $\mathcal{T}$. An accepting run starts in state 2, then moves to state 4, and stutters on this livelock-accepting state. Another possible accepting run goes from state 2 to state 3 and stutters in $3 \in \mathcal{G}$.
- The infinite word $ab;a\bar{b};ab;a\bar{b};ab;a\bar{b};\ldots$ is not accepted. It would correspond to a run alternating between states 2 and 1, but such a run is neither Büchi accepting (does not visit any $\mathcal{F}$ state) nor livelock-accepting (it passes through state $2 \in \mathcal{G}$, but does not stay into this state continuously).

Property 2. *The language accepted by a testing automaton is stutter-invariant.*

Proof. This follows from definition of accepted infinite words: a TA may not change its state when an infinite word stutters, so stuttering is always possible. □

3.4 TA Construction

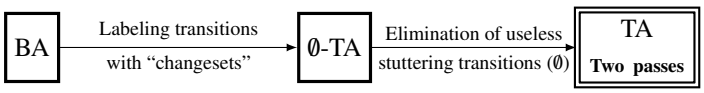


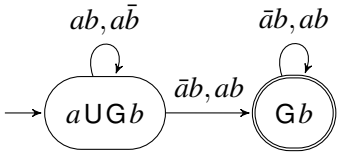
Figure 3.2: The two steps of the construction of a TA from a BA.

A TA is constructed from a BA in two steps as illustrated in Figure 3.2. The first step constructs an intermediate form of TA, called $\emptyset$-TA (for "empty-changesets" TA), which can contain stuttering transitions between two distinct states (i.e., $\delta \subseteq Q \times \Sigma \times Q$ in $\emptyset$-TA). The second step allows to eliminate these useless stuttering transitions and thus to obtain a TA that satisfies the Definition 23 (i.e., $\delta \subseteq Q \times (\Sigma \setminus \emptyset) \times Q$ in TA).

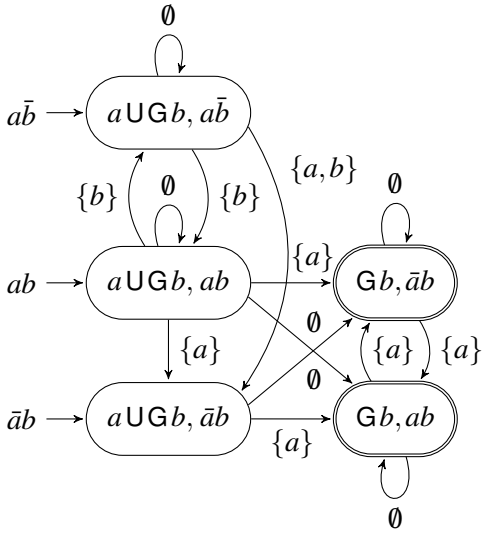
We begin by formally defining an $\emptyset$-TA automaton and how it accepts infinite words:

Definition 24 ($\emptyset$-TA)**.** *An $\emptyset$-TA over the alphabet $\Sigma = 2^{AP}$ is a tuple $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$, where:*

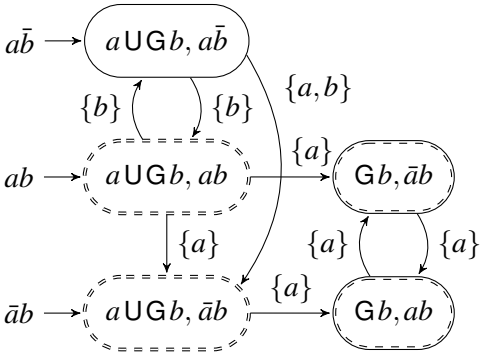
- *Q is a finite set of states,*
- *$I \subseteq Q$ is the set of initial states,*
- *$U : I \to 2^{\Sigma}$ is a function mapping each initial state to a set of valuations,*
- *$\delta \subseteq Q \times \Sigma \times Q$ is the transition relation where each transition (s,k,d) is labeled by a changeset $k \in \Sigma$ interpreted as a set of atomic propositions whose value must change between states s and d,*
- *$\mathcal{F} \subseteq Q$ is a set of Büchi-accepting states,*
- *$\mathcal{G} \subseteq Q$ is a set of livelock-accepting states.*



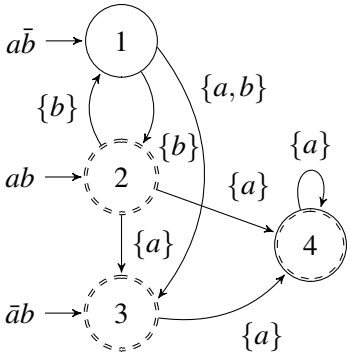
(a) Initial BA for $a\mathsf{U}\mathsf{G}b$.



(b) $\emptyset$-TA after the construction from property 3.



(c) TA after the simplifications from property 4.



(d) TA after bisimulation.

Figure 3.3: Steps of the construction of a TA from a BA. States with a double enclosure belong to either $\mathcal{F}$ or G: states in $\mathcal{F} \setminus \mathcal{G}$ have a double plain line, states in $\mathcal{G} \setminus \mathcal{F}$ have a double dashed line, and states in $\mathcal{F} \cap \mathcal{G}$ use a mixed dashed/plain style.

An infinite word $\sigma = \ell_0\ell_1\ell_2\ldots \in \Sigma^\omega$ *is accepted by* $\mathcal{T}$ iff *there exists an infinite sequence* $r = (q_0, \ell_0 \oplus \ell_1, q_1)(q_1, \ell_1 \oplus \ell_2, q_2)\ldots(q_i, \ell_i \oplus \ell_{i+1}, q_{i+1})\ldots \in (Q \times \Sigma \times Q)^\omega$ *such that:*

- $q_0 \in I$ *with* $\ell_0 \in U(q_0)$,
- $\forall i \in \mathbb{N}$, *either* $(q_i, \ell_i \oplus \ell_{i+1}, q_{i+1}) \in \delta$ *(we are always progressing in* $\emptyset$*-TA),*
- *Either,* $\forall i \in \mathbb{N}, (\exists j \geq i, q_j \in \mathcal{F})$ *(the* $\emptyset$*-TA is progressing in a Büchi-accepting way), or,* $\exists n \in \mathbb{N}, \forall i \geq n, (\ell_i = \ell_n) \wedge (q_i \in \mathcal{G})$ *(a suffix of the infinite word stutters in* $\mathcal{G}$*).*

The language accepted by $\mathcal{T}$ *is the set* $\mathscr{L}(\mathcal{T}) \subseteq \Sigma^\omega$ *of infinite words it accepts.*

Figure 3.3b shows an example of an $\emptyset$-TA. The $\emptyset$-TA does not respect the constraint of TA that prohibits having stuttering transitions between two distinct states. The second difference is that the stuttering transitions are explicitly represented in an $\emptyset$-TA. In other words, an $\emptyset$-TA does not use the implicit stuttering of TA (which consists of having an implicit stuttering self-loop on each state of TA).

3.4.1 From BA to $\emptyset$-TA: Construction of an intermediate $\emptyset$-TA from a Büchi Automaton BA

Geldenhuys and Hansen [46] have shown how to convert a BA into a TA by first converting the BA into an automaton with valuations on the states (called state-labeled Büchi automaton (SLBA)), and then converting this SLBA into an intermediate form of TA (i.e., an $\emptyset$-TA) by computing the difference between the labels of the source and destination states of each transition. The next proposition implements these first steps.

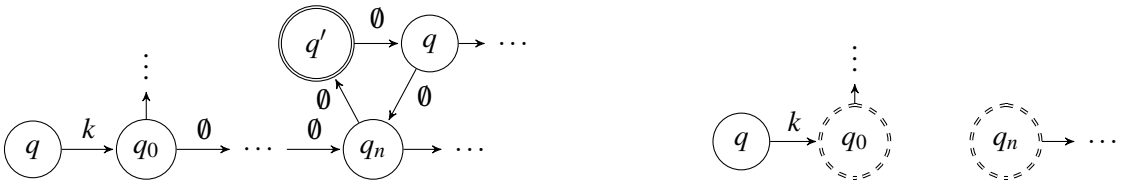
Property 3 (Converting a BA into an $\emptyset$-TA (an intermediate form of TA) [46])**.** *For any BA* $\mathcal{B} = \langle Q_\mathcal{B}, I_\mathcal{B}, \delta_\mathcal{B}, F_\mathcal{B}\rangle$ *over the alphabet* $\Sigma = 2^{AP}$ *and such that* $\mathscr{L}(\mathcal{B})$ *is stutter-invariant, let us define the* $\emptyset$*-TA* $\mathcal{T} = \langle Q_\mathcal{T}, I_\mathcal{T}, U_\mathcal{T}, \delta_\mathcal{T}, F_\mathcal{T}, \emptyset\rangle$ *with:*

- $Q_\mathcal{T} = Q_\mathcal{B} \times \Sigma$,
- $I_\mathcal{T} = I_\mathcal{B} \times \Sigma$,
- $\forall (s,\ell) \in I_\mathcal{T}, U_\mathcal{T}((s,\ell)) = \{\ell\}$,
- $\forall (s,\ell) \in Q_\mathcal{T}, \forall (s',\ell') \in Q_\mathcal{T}$,
  $((s,\ell), \ell \oplus \ell', (s',\ell')) \in \delta_\mathcal{T} \iff ((s,\ell,s') \in \delta_\mathcal{B})$,
- $F_\mathcal{T} = F_\mathcal{B} \times \Sigma$.

Then $\mathscr{L}(\mathcal{B}) = \mathscr{L}(\mathcal{T})$.

The proof of Property 3 is similar to the proof of Property 8, which will be given in Section 5.3.1.

Figure 3.3b shows the result of applying this construction to the example of Büchi automaton shown for $a\,\mathsf{U}\,\mathsf{G}\,b$. This testing automaton does not yet use livelock-accepting states (the $\mathcal{G}$ set). The next property, again from Geldenhuys and Hansen [46], shows how filling $\mathcal{G}$ allows to remove all stuttering transitions (i.e., transitions labeled by $\emptyset$) and therefore build the final form of the TA.



(a) Before stuttering transitions ($\emptyset$) reduction. q' is a Büchi accepting state

(b) After reduction, q_0 and q_n are livelock-accepting states.

Figure 3.4: Elimination of useless stuttering transitions. The states q_0 and q_n are added to the set of livelock-accepting states $\mathcal{G}$.

3.4.2 From $\emptyset$-TA to TA: Elimination of useless stuttering transitions ($\emptyset$) and introducing livelock-acceptance

The second step consists in filling $\mathcal{G}$ to simplify $\mathcal{T}$, the intuition of this second step is illustrated by Figure 3.4. For that, we compute all strongly connected components using only stuttering transitions (i.e., transitions labeled by $\emptyset$). If such a SCC is not trivial (i.e., it contains a cycle) and contains a Büchi-accepting state, then add all its states to $\mathcal{G}$. Then, add to $\mathcal{G}$ any state that can reach $\mathcal{G}$ using only stuttering transitions. Finally remove all stuttering transitions from δ. The following property formalizes this second step.

Property 4 (Filling $\mathcal{G}$ to obtain a TA [46])**.** *Let $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G}\rangle$ be $\emptyset$-TA such that $\mathscr{L}(\mathcal{T})$ is stutter-invariant. By combining the first three of the following operations we can remove all stuttering transitions $(q, \emptyset, q')$ from $\mathcal{T}$ and thus obtain a TA. The fourth simplification can be performed along the way.*

1. *If $Q' \subseteq Q$ is a Strongly Connected Component (SCC) such that $Q' \cap \mathcal{F} \neq \emptyset$ (it is Büchi-accepting), and any two states $q, q' \in Q'$ can be connected using a non-empty sequence of stuttering transitions $(q, \emptyset, q_1) \cdot (q_1, \emptyset, q_2) \cdots (q_n, \emptyset, q') \in \delta^*$, then the automaton $\mathcal{T}' = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \cup Q'\rangle$ is such that $\mathscr{L}(\mathcal{T}') = \mathscr{L}(\mathcal{T})$. Such a component Q' is called an **accepting Stuttering-SCC**.*

2. *If there exists a transition $(q, \emptyset, q') \in \delta$ such that $q' \in \mathcal{G}$, then the $\emptyset$-TA $\mathcal{T}'' = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \cup \{q\}\rangle$ is such that $\mathscr{L}(\mathcal{T}'') = \mathscr{L}(\mathcal{T})$.*

3. *Let $\mathcal{T}^\dagger = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G}^\dagger\rangle$ be the $\emptyset$-TA obtained after repeating the previous two operations as much as possible (i.e., $\mathcal{G}^\dagger$ contains all states that can be added by the above two operations (Figure 3.4b)). Then, because $\mathscr{L}(\mathcal{T})$ and thus $\mathscr{L}(\mathcal{T}^\dagger)$ are stutter-invariant, we can remove all stuttering transitions form $\mathcal{T}^\dagger$ to obtain a TA (since stuttering can be captured by the implicit stuttering of TA and the livelock-accepting states of $\mathcal{G}^\dagger$ after the previous two operations). After this last reduction of stuttering transitions, we obtain the final TA $\mathcal{T}'''$. Formally, the TA $\mathcal{T}''' = \langle Q, I, U, \delta''', \mathcal{F}, \mathcal{G}^\dagger\rangle$ with $\delta''' = \{(q, k, q') \in \delta \mid k \neq \emptyset\}$ is such that $\mathscr{L}(\mathcal{T}''') = \mathscr{L}(\mathcal{T}^\dagger) = \mathscr{L}(\mathcal{T})$.*

4. *Any state from which one cannot reach a Büchi-accepting cycle nor a livelock-accepting state can be removed without changing the automaton's language.*

The proof of Property4 is similar to the proof of Property 9, which will be given in Section 5.3.2.

The resulting TA can be further simplified by merging bisimilar states (two states are bisimilar if the automaton can accept the same infinite words starting for either of these states). This can be achieved using any algorithm based on partition refinement, the same as for Büchi automata presented in section 2.4.5, taking $\{\mathcal{F} \cap \mathcal{G}, \mathcal{F} \setminus \mathcal{G}, \mathcal{G} \setminus \mathcal{F}, \mathcal{Q} \setminus (\mathcal{F} \cup \mathcal{G})\}$ as initial partition.

Figure 3.3 shows how a BA denoting the LTL formula $a \mathsf{U} \mathsf{G} b$ is transformed into a TA by applying prop. 3, prop. 4, and finally merging bisimilar states.

A TA for $\mathsf{GF}a \wedge \mathsf{GF}b$ is too big to be shown: even after simplifications it has 11 states and 64 transitions.

3.4.3 TA Optimizations (that are not yet implemented)

Looking at Figure 3.3 inspires two optimizations. The first one is based on the fact that the construction of testing automata, described in the previous section, generates a lot of bisimilar states such as the two states labeled with $(\mathsf{G}b, \bar{a}b)$ and $(\mathsf{G}b, ab)$. This is because the construction considers all the elements of Σ that are compatible with $\mathsf{G}b$. Had the LTL formula been over $AP = \{a, b, c\}$, e.g., $(a \vee c) \mathsf{U} \mathsf{G} b$, then we would have had four bisimilar states: $(\mathsf{G}b, \bar{a}b\bar{c})$, $(\mathsf{G}b, \bar{a}bc)$, $(\mathsf{G}b, ab\bar{c})$, and $(\mathsf{G}b, abc)$. These states are *necessarily* isomorphic, because they only differ in a and c, some propositions that the formula $\mathsf{G}b$ does not *observe*.

A more efficient way to construct the testing automaton (and to construct the automaton from Figure 3.3d directly) would be to consider only the subset of atomic propositions that are observed by the corresponding state of the Büchi automaton or its descendants (if the state is labeled by an LTL formula, the atomic propositions occurring in this formula give an over-approximation of that set).

A second optimization relies on the fact any state that does not belong to an SCC can be added to $\mathcal{F}$ without changing the language of the automaton (this is also true for Büchi automata). For instance on Figure 3.3 the state labeled $(a \mathsf{U} \mathsf{G} b, \bar{a}b)$ can be added to $\mathcal{F}$. Since this state is not part of any cycle, it cannot occur infinitely often and therefore cannot change the accepted language of the automaton.

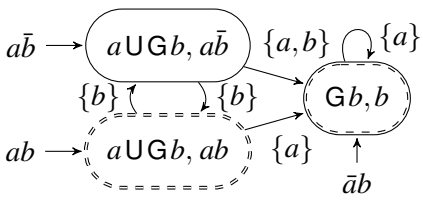


Figure 3.5: Reduced TA for $a \mathsf{U} \mathsf{G} b$.

This change allows further simplifications by bisimulation: the state $(a \mathsf{U} \mathsf{G} b, \bar{a}b)$ is now obviously equivalent to the $(\mathsf{G}b, b)$ state. Figure 3.5 shows the resulting automaton. Note that putting any trivial SCC x in $\mathcal{F}$ before performing bisimulation could hinder the reduction if x was isomorphic to some state not in $\mathcal{F}$. However if x has only successors in $\mathcal{F}$, as in our example, then it can be put safely in $\mathcal{F}$: indeed, it can only be isomorphic to an $\mathcal{F}$-state, or to another trivial SCC that will be added to $\mathcal{F}$. This condition is similar to the one used by Löding before minimizing deterministic weak ω-automata [61].

Unfortunately, these optimizations have not been implemented for this work and we had no time to implement the *simulation* [40, 44] reduction.

However, the bisimulation reduction (section 2.4.5) was implemented for the three approaches

BA, TGBA and TA, and used in the experimental comparisons presented later in this thesis.

3.5 Explicit Model checking using TA

A first difference between the BA and TA approaches resides in the computation of the synchronous product. Indeed, during the computation of the product between a TA $\mathcal{T}$ and a Kripke structure $\mathcal{K}$, $\mathcal{T}$ remains in the same state when $\mathcal{K}$ executes a stuttering step.

The emptiness check also requires a dedicated algorithm because there are two ways to accept an infinite word: Büchi accepting or livelock accepting. In the algorithm sketched by Geldenhuys and Hansen [46], a first pass is used with an heuristic to detect both Büchi and livelock accepting cycles. Unfortunately, in certain cases, this first pass fails to report existent livelock accepting cycles. This implies that when no counterexample is found by the first pass, a second one is required to double-check for possible livelock accepting cycles. Thus, when there is no counterexample (i.e., the property is satisfied), the entire state-space has to be explored twice.

3.5.1 Synchronous Product of a TA with a Kripke structure

For traditional Büchi automata, the product of a BA or a TGBA with a Kripke structure, is also respectively a BA or a TGBA. In the case of testing automata, the product of a Kripke structure and a TA is not a TA. Indeed, while an execution in a TA is allowed to stutter on any state, the execution in a product must execute an explicit stuttering transition [46]. This product automaton can be seen as an $\emptyset$-TA but with a small difference in the way of recognizing Büchi-accepting runs. Indeed, a Büchi-accepting run of the product must visit at least a non-stuttering transition infinitely often.

Chapter 5 will introduce a new type of ω-automata that improve the TA, called TGTA [6], which simplifies the definition of the product and improves its emptiness check (by proposing a single-pass algorithm). Indeed, the $\emptyset$-TGTA used as an intermediate form in the construction of a TGTA will also serve to represent the product between a TGTA and a Kripke structure. In addition, the emptiness check of this product using TGTA will be done in a single pass (in the next section, we will show that the product using TA requires two passes for its emptiness check).

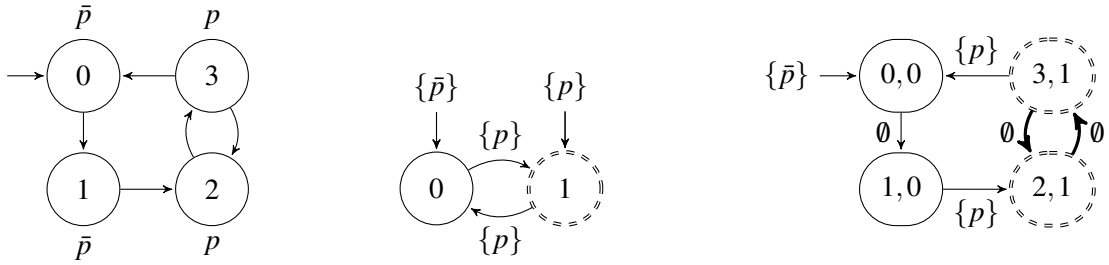
Definition 25. *For a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ and a TA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$, the product $\mathcal{K} \otimes \mathcal{T}$ is a tuple $\langle \mathcal{S}_\otimes, I_\otimes, U_\otimes, \delta_\otimes, \mathcal{F}_\otimes, \mathcal{G}_\otimes \rangle$ where*

- $\mathcal{S}_\otimes = \mathcal{S} \times Q$,
- $I_\otimes = \{(s,q) \in \mathcal{S}_0 \times I \mid l(s) \in U(q)\}$,
- $\forall (s,q) \in I_\otimes, U_\otimes((s,q)) = \{l(s)\}$,
- $\delta_\otimes = \{((s,q),k,(s',q')) \mid (s,s') \in \mathcal{R}, (q,k,q') \in \delta, k = l(s) \oplus l(s')\}$,
  $\cup \{((s,q),\emptyset,(s',q')) \mid (s,s') \in \mathcal{R}, q = q', l(s) = l(s')\}$
- $\mathcal{F}_\otimes = \mathcal{S} \times \mathcal{F}$,
- *and* $\mathcal{G}_\otimes = \mathcal{S} \times \mathcal{G}$.

An execution $\sigma = \ell_0 \ell_1 \ell_2 \ldots \in \Sigma^\omega$ is accepted by $\mathcal{K} \otimes \mathcal{T}$ if there exists an infinite sequence $r = (s_0, \ell_0 \oplus \ell_1, s_1)(s_1, \ell_1 \oplus \ell_2, s_2) \ldots (s_i, \ell_i \oplus \ell_{i+1}, s_{i+1}) \ldots \in (\mathcal{S}_\otimes \times \Sigma \times \mathcal{S}_\otimes)^\omega$ such that:

- $s_0 \in \mathcal{S}^0_\otimes$ *with* $\ell_0 \in U_\otimes(s_0)$,
- $\forall i \in \mathbb{N}, (s_i, \ell_i \oplus \ell_{i+1}, s_{i+1}) \in \delta_\otimes$ *(we are always progressing in the product)*
- *Either,* $\forall i \in \mathbb{N}, (\exists j \geq i, \ell_j \neq \ell_{j+1}) \wedge (\exists l \geq i, s_l \in \mathcal{F}_\otimes)$ *(the automaton is progressing in a Büchi-accepting way), or,* $\exists n \in \mathbb{N}, \forall i \geq n, (\ell_i = \ell_n) \wedge (s_i \in \mathcal{G}_\otimes)$ *(a suffix of the execution stutters in* $\mathcal{G}_\otimes$*).*

We have $\mathscr{L}(\mathcal{K} \otimes \mathcal{T}) = \mathscr{L}(\mathcal{K}) \cap \mathscr{L}(\mathcal{T})$ *by construction.*



(a) A Kripke structure $\mathcal{K}$ (b) A TA $\mathcal{T}$ for $\mathsf{FG}\,p$ (c) $\mathcal{K} \otimes \mathcal{T}$

Figure 3.6: Example of a synchronous product between a Kripke structure $\mathcal{K}$ and a TA $\mathcal{T}$ recognizing the LTL formula $\mathsf{FG}\,p$. The bold cycle of $\mathcal{K} \otimes \mathcal{T}$ is livelock-accepting.

Figure 3.6 shows an example of a synchronous product between a Kripke structure $\mathcal{K}$ and a TA $\mathcal{T}$ recognizing the LTL formula $\mathsf{FG}\,p$. Each state of $\mathcal{K}$ is numbered and labeled with a valuation of atomic propositions (over $AP = \{p\}$) that hold in this state. In the product $\mathcal{K} \otimes \mathcal{T}$, states are labeled with a pairs of the form (s,q) with s is a state of $\mathcal{K}$ and q of $\mathcal{T}$, and the livelock accepting states are denoted by a double dashed circle (as in $\mathcal{T}$). We can notice that this product using TA is smaller than the product of Figure 2.12 using Büchi automata, i.e., the product between the same Kripke structure $\mathcal{K}$ and the TGBA (or BA) recognizing the LTL formula $\mathsf{FG}\,p$.

3.5.2 A two-pass emptiness check algorithm

In this section, we present a two-pass algorithm for the emptiness check of the synchronous product between a TA and a Kripke structure. While the emptiness check proposed by Geldenhuys and Hansen [46] is based on the Tarjan algorithm [82, 83], the emptiness check proposed in this section, consists of Algorithm 2 (`first-pass`) and Algorithm 3 (`second-pass`), which are based on the Dijkstra algorithm for SCCs detection [32, 33].

In model checking approach using TA, the emptiness check requires a dedicated algorithm because according to the Definition 25 (of the different ways to accept a sequence) , there are two ways to detect an accepting cycle in the product:

- Büchi accepting: a cycle containing at least one Büchi-accepting state ($\mathcal{F}$) and at least one non-stuttering transition (i.e., a transition (s,k,s') with $k \neq \emptyset$),
- livelock accepting: a cycle composed only by stuttering transitions and livelock accepting states ($\mathcal{G}$).

A straightforward emptiness check would have the following two passes:

```
1  Input: A product $\mathcal{K}\otimes\mathcal{T} = \langle \mathcal{S}_\otimes, I_\otimes, U_\otimes, \delta_\otimes, \mathcal{F}_\otimes, \mathcal{G}_\otimes\rangle$
2  Result: $\top$ if and only if $\mathscr{L}(\mathcal{K}\otimes\mathcal{T}) = \emptyset$
3  Data: todo: stack of $\langle state \in \mathcal{S}_\otimes, succ \subseteq \delta_\otimes\rangle$
         SCC: stack of $\langle root \in \mathbb{N}, lk \in 2^{AP}, k \in 2^{AP}, acc \subseteq \mathcal{F}_\otimes, rem \subseteq \mathcal{S}_\otimes\rangle$
         H: map of $\mathcal{S}_\otimes \mapsto \mathbb{N}$
         $max \leftarrow 0$
4  begin
5      if ¬ first-pass() then return $\bot$ ;
6      return second-pass()
7  first-pass()
8      foreach $s^0 \in I_\otimes$ do
9          DFSpush1($\emptyset$, $s^0$) while ¬todo.empty() do
10             if todo.top().succ $= \emptyset$ then
11                 DFSpop()
12             else
13                 pick one $\langle s,k,d\rangle$ off todo.top().succif $d \notin H$ then
14                     DFSpush1(k, d)
15                 else if $H[d] > 0$ then
16                     merge1(k, $H[d]$) if $(SCC.top().acc \neq \emptyset) \wedge$
                       $(SCC.top().k \neq \emptyset)$ then return $\bot$;
17                     if $(d \in \mathcal{G}_\otimes) \wedge (SCC.top().k = \emptyset)$ then return $\bot$ ;
18     return $\top$
19 DFSpush1($lk \in 2^{AP}$, $s \in \mathcal{S}_\otimes$)
20     $max \leftarrow max + 1$
21     $H[s] \leftarrow max$
22     if $s \in \mathcal{F}_\otimes$ then
23         SCC.push($\langle max, lk, \emptyset, \{s\}, \emptyset\rangle$)
24     else
25         SCC.push($\langle max, lk, \emptyset, \emptyset, \emptyset\rangle$)
26     todo.push($\langle s, \{\langle q,k,d\rangle \in \delta_\otimes \mid q = s\}\rangle$)
27 merge1($lk \in 2^{AP}$, $t \in \mathbb{N}$)
28     $acc \leftarrow \emptyset$
29     $r \leftarrow \emptyset$
30     $k \leftarrow lk$
31     while $t < SCC.top().root$ do
32         $acc \leftarrow acc \cup SCC$.top().acc
33         $k \leftarrow k \cup SCC$.top().$k \cup SCC$.top().lk
34         $r \leftarrow r \cup SCC$.top().rem
35         SCC.pop()
36     SCC.top().$acc \leftarrow SCC$.top().$acc \cup acc$
37     SCC.top().$k \leftarrow SCC$.top().$k \cup k$
38     SCC.top().$rem \leftarrow SCC$.top().$rem \cup r$
```

Algorithm 2: The first-pass of the Emptiness check algorithm for TA products, see Algorithm 3 for the `second-pass`.

```
1  Data: todo: stack of ⟨state ∈ S⊗, succ ⊆ S⊗⟩
         SCC: stack of ⟨root ∈ ℕ, rem ⊆ S⊗⟩
         H: map of S⊗ ↦ ℕ
         max ← 0; init ← I⊗
2  second-pass()
3      while ¬init.empty() do
4          pick one s⁰ off init if s⁰ ∉ H then DFSpush2(∅, s⁰) ;
5          while ¬todo.empty() do
6              if todo.top().succ = ∅ then
7                  DFSpop()
8              else
9                  pick one d off todo.top().succ if d ∉ H then
10                     DFSpush2(d)
11                 else if H[d] > 0 then
12                     merge2(H[d]) if (d ∈ G⊗) then return ⊥ ;
13     return ⊤
14 DFSpush2(s ∈ S⊗)
15     max ← max + 1
16     H[s] ← max
17     SSCC.push(⟨max, ∅⟩)
18     todo.push(⟨s, {d ∈ S⊗ | (s, ∅, d) ∈ δ⊗}⟩)
19     init ← init ∪ {d ∈ S⊗ | (s, k, d) ∈ δ⊗ ∧ k ≠ ∅}
20 merge2(t ∈ ℕ)
21     r ← ∅
22     while t < SCC.top().root do
23         r ← r ∪ SCC.top().rem
24         SCC.pop()
25     SCC.top().rem ← SCC.top().rem ∪ r
```

Algorithm 3: The second-pass of the TA emptiness check algorithm.

- a first pass to detect Büchi accepting cycles, it corresponds to Algorithm 2 without the test at line 17,
- a second pass presented in Algorithm 3 to detect livelock accepting cycles.

It is not possible to merge these two passes into a single DFS: the first DFS requires the exploration of every transition of the product while the second one must consider only stuttering transitions.

The `first-pass` of Algorithm 2 detects all Büchi-accepting cycles, and with line 17 included in this algorithm, it detects also some livelock-accepting cycles. Since in certain cases it may fail to report some livelock-accepting cycles, a second pass is required to look for possible livelock-accepting cycles.

This `first-pass` is based on the TGBA emptiness check algorithm presented in Algorithm 1 (page 34) with the following changes:

- In each item *scc* of the *SCC* stack: the field *scc.acc* contains the Büchi-accepting states detected in *scc*, *scc.lk* is analogous to *la* in Figure 2.13 but it stores the *change-set* labeling the transition coming from the previous SCC, and *scc.k* contains the union of all *change-sets* in *scc* (lines 33 and 37).
- After each merge, *SCC*.top() is checked for Büchi-acceptance (line 16) or livelock-acceptance (line 17) depending on the emptiness of *SCC*.top().*k*.

Figure 3.6 illustrates how the `first-pass` of Algorithm 2 can fail to detect the livelock accepting cycle in a product $\mathcal{K} \otimes \mathcal{T}$ as defined in Definition 25. In this example, $G_{\mathcal{T}} = \{1\}$ therefore $(3,1)$ and $(2,1)$ are livelock-accepting states, and $C_2 = [(3,1) \rightarrow (2,1) \rightarrow (3,1)]$ is a livelock-accepting cycle.

However, the `first-pass` may miss this livelock-accepting cycle depending on the order in which it processes the outgoing transitions of $(3,1)$. If the transition $t_1 = ((3,1),\{p\},(0,0))$ is processed before $t_2 = ((3,1),\emptyset,(2,1))$, then the cycle $C_1 = [(0,0) \rightarrow (1,0) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (0,0)]$ is detected and the four states are merged in the same SCC before exploring t_2. After this merge (line 16), this SCC is at the top of the SCC stack. Subsequently, when the DFS explores t_2, the merge caused by the cycle C_2 does not add any new state to the SCC, and the *SCC* stack remains unchanged. Therefore, the test line 17 still return false because the union $SCC.top().k$ of all change-sets labeling the transitions of the SCC is not empty (it includes for example t_1's label: $\{p\}$). Finally, `first-pass` algorithm terminates without reporting any accepting cycle, missing C_2.

On the other side, if the `first-pass` had processed t_2 before t_1, it would have merged the states $(3,1)$ and $(2,1)$ in an SCC, and would have detected it to be livelock-accepting.

In general, to report a livelock-accepting cycle, the first-pass computes the union of all change-sets of the SCC containing this cycle. However, this union may include non-stuttering transitions belonging to other cycles of the SCC. In this case, the `second-pass` is required to search for livelock-accepting cycles, ignoring the non-stuttering transitions that may belong to the same SCC.

In the next chapter, we propose a Single-pass Testing Automata STA, which allows to obtain a synchronous product in which such mixing of non-stuttering and stuttering transitions will never occur in SCCs containing livelock-accepting cycles, making the `second-pass` unnecessary.

The `second-pass` (Algorithm 3) is a DFS exploring only stuttering transitions (line 18). To report a livelock-accepting cycle, it detects "stuttering-SCCs" and tests if they contain a livelock-accepting state (line 12).

Ignoring the non-stuttering transitions during the DFS, may lead to miss some parts of the product so any destination of a non stuttering transition is stored in *init* for later exploration (line 19).

In the algorithm proposed by Geldenhuys and Hansen [46], the first pass uses a heuristic to detect livelock-accepting cycles when possible. This heuristic detects more livelock-accepting cycles than Algorithm 2. In certain cases this first pass may still fail to report some livelock-accepting cycles. Yet, this heuristic is very efficient: when counterexamples exist, they are usually caught by the first pass, and the second is rarely needed. However, when properties are satisfied, the second pass is always required.

Note. It is important to say that in the experiments presented in the sequel, we implement Algorithm 2 including the heuristic proposed by Geldenhuys and Hansen [46] to detect more livelock-accepting cycles during the first pass of the TA approach. We don't present the details of this heuristic because we show in the next chapters other solutions that allow to detect all the livelock-accepting cycles during the first pass and therefore remove the second pass.

3.6 Experimental Comparison of TA versus TGBA and BA

This section presents our experimentation of the various types of automata within our tool Spot [64]. We first present the Spot architecture and the way the variation on the model checking algorithm was introduced. Then we present our benchmarks (formulas and models) prior to the description of our experiments.

3.6.1 Implementation on top of Spot

Spot is a model-checking library offering several algorithms that can be combined to build a model checker [36].

Figure 3.7 shows the building blocks we used to implement the three approaches:

- the TGBA and BA approaches that share the same synchronous product construction and emptiness check,

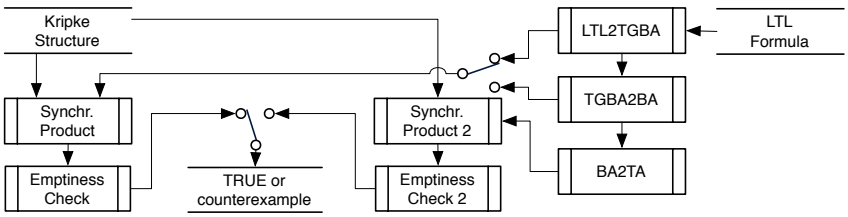


Figure 3.7: The experiment's architecture. Two command-line switches control which one of the three approaches is used to verify an LTL formula on a Kripke structure.

- the dedicated algorithms required by the TA approach.

The construction of TGBA and BA already exists in Spot. In addition, we have implemented all algorithms used in the TA approach (TA construction and TA product and its emptiness check). We have also implemented the bisimulation reduction (section 2.4.5) for three approaches BA, TGBA and TA.

In order to evaluate our approach on "realistic" models, we decided to couple the Spot library with the CheckPN tool [36]. CheckPN implements Spot's Kripke structure interface in order to build on-the-fly a state-space of a Petri net. This Kripke structure is then synchronized with an ω-automaton (TGBA, BA, or TA), and fed to the suitable emptiness check algorithm. The latter algorithm drives the on-the-fly construction: only the explored part of the product (and the associated states of the Kripke structure) will be constructed.

Constructing the state-space on-the-fly is a double-edged optimization. Firstly, it saves memory, because the state-space is computed as it is explored and thus, does not need be stored. Secondly, it also saves time when a property is violated because the emptiness check can stop as soon as it has found a counterexample. However, on-the-fly exploration is costlier than browsing an explicit graph: an emptiness check algorithm such as the one for TA [52] that does two traversals of the full state-space in the worst case (e.g. when the property holds) will pay twice the price of that construction.

In the CheckPN implementation of the Kripke structure, the Petri Net marking are compressed to save memory. The marking of a state has to be uncompressed every time we compute its successors, or when we compute the value of the atomic properties on this state. These two operations often occur together, so there is a one-entry cache that prevents the marking from being uncompressed twice in a row.

3.6.2 Benchmark Inputs

We selected some Petri net models and formulas to compare these approaches.

Case Studies The following two bigger models, were taken from actual case studies. They come with some *dedicated* properties to check.

PolyORB models the core of the μbroker component of a middleware [56] in an implementation using a Leader/Followers policy [71]. It is a Symmetric Net and, since CheckPN processes P/T nets only, it was unfolded into a P/T net. The resulting net, for a configuration involving three sources of data, three simultaneous jobs and two threads (one leader, one follower) is composed of 189 places and 461 transitions. Its state space contains 61 662 states. The authors propose to check that once a job is issued from a source, it must be processed by a thread (no starvation). It corresponds to:

$$\Phi_1 = \mathsf{G}(MSrc_1 \rightarrow \mathsf{F}(DOSrc_1)) \wedge \mathsf{G}(MSrc_2 \rightarrow \mathsf{F}(DOSrc_2)) \wedge \mathsf{G}(MSrc_3 \rightarrow \mathsf{F}(DOSrc_3))$$

MAPK models a biochemical reaction: Mitogen-activated protein kinase cascade [53]. For a scaling value of 8 (that influences the number of tokens in the initial marking), it contains 22 places and 30 transitions. Its state-space contains 6.11×10^6 states. The authors propose to check that from the initial state, it is necessary to pass through states *RafP*, *MEKP*, *MEKPP* and *ERKP*

in order to reach *ERKPP*. In LTL:

$$\Phi_2 = \neg((\neg RafP)\,\mathsf{U}\,MEKP) \wedge \neg((\neg MEKP)\,\mathsf{U}\,MEKPP) \wedge \\ \neg((\neg MEKPP)\,\mathsf{U}\,ERKP) \wedge \neg((\neg ERKP)\,\mathsf{U}\,ERKPP)$$

Toy Examples A second class of models were selected from the Petri net literature [19, 59]: the flexible manufacturing system (FMS), the Kanban system, the Peterson algorithm, the slotted-ring system, the dining philosophers and the Round-robin mutex [21]. All these models have a parameter n. For the dining philosophers, the Peterson algorithm, the Round-robin, and the slotted-ring, the models are composed of n 1-safe subnets. For FMS and Kanban, n only influences the number of tokens in the initial marking. In our experiments, we selected the following 12 models instances:

- $n = 4$ and $n = 5$ for Peterson, FMS and Kanban,
- $n = 6$ and $n = 5$ for slotted-ring,
- $n = 9$ and $n = 10$ for dining philosophers,
- $n = 14$ and $n = 15$ for Round-robin.

Types of Formulas As suggested by Geldenhuys and Hansen [46], the type of formula may affect the performances of the various algorithms. In addition to the formulas Φ_1 and Φ_2 above, we consider two classes of formulas:

- *RND*: randomly generated LTL formulas (without X operator). Since random formulas are very often trivial to verify (the emptiness check needs to explore only a handful of states), we selected for each model only those formulas requiring more than one second of CPU for the emptiness check in all approaches.

- *WFair*: properties of the form $(\bigwedge_{i=1}^{n} \mathsf{GF}\, p_i) \rightarrow \varphi$, where φ is a randomly generated LTL formula. This represents the verification of φ under the weak-fairness hypothesis $\bigwedge_{i=1}^{n} \mathsf{GF}\, p_i$. The automaton representing such a formula has at least n acceptance conditions which means that the BA will in the worst case be $n+1$ times bigger than the TGBA. For the formulas we generated for our experiments we have $n = 3.51$ on the average.

All formulas were translated into automata using Spot, which was shown experimentally to be very good at this job [74, 34].

For each selected model instance, we generated:

- Verified formulas (i.e., no counterexample in the product): 100 random and 100 weak-fairness,
- Violated formulas (i.e., a counterexample exists): 100 random and 100 weak-fairness.

We consequently have a total 5600 pairs of (model, formula): 2800 violated formulas and 2800 verified formulas.

3.6.3 Results

Table 3.1 and Table 3.2 show how BA, TGBA and TA approaches deal with toy models and random formulas.

			Verified properties (no counterexample)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	BA	avg	13	502	0	2791685	14311851	2791685	14311851	18968
		max	82	2827	7	11010896	108553439	11010896	108553439	80529
	TGBA	avg	12	460	1	2753024	14166594	**2753024**	**14166594**	**18656**
		max	54	2816	6	11010896	108553439	11010896	108553439	80455
	TA	avg	83	4324	12	**2600539**	**10293714**	5201077	20587428	34992
		max	509	37096	242	9384663	42614845	18769326	85229690	124863
Ring6	BA	avg	13	448	0	1467181	10533598	1467181	10533598	2422
		max	49	3788	4	4272666	50547840	4272666	50547840	11538
	TGBA	avg	12	411	0	1398021	10139043	**1398021**	**10139043**	**2325**
		max	49	3788	4	4269384	50547840	4269384	50547840	10181
	TA	avg	69	3118	9	**1097667**	**6606130**	2195333	13212259	3748
		max	509	25773	151	2851152	19625864	5702304	39251728	9688
FMS5	BA	avg	10	267	0	2077187	15144149	2077187	15144149	3074
		max	31	2612	4	9132417	89397363	9132417	89397363	16452
	TGBA	avg	9	245	0	2038054	15037218	**2038054**	**15037218**	**3017**
		max	28	2612	4	9132417	89397363	9132417	89397363	15594
	TA	avg	48	1643	5	**1401286**	**11000449**	2802572	22000898	5347
		max	282	11007	100	6109887	54324903	12219774	108649806	26238
Kanban5	BA	avg	9	185	0	3494358	33955856	3494358	33955856	5424
		max	94	1994	4	22360464	282849140	22360464	282849140	43179
	TGBA	avg	8	166	0	3356053	32800737	**3356053**	**32800737**	**5253**
		max	60	1994	4	20253072	258315134	20253072	258315134	44281
	TA	avg	44	1659	3	**2433853**	**23389805**	4867707	46779609	9192
		max	277	21251	82	15272712	161364553	30545424	322729106	61873
Philo10	BA	avg	16	705	1	4952039	29080163	4952039	29080163	9841
		max	81	5995	7	18399098	121269824	18399098	121269824	43970
	TGBA	avg	14	636	1	4668178	27564474	**4668178**	**27564474**	**9352**
		max	81	5397	6	17947837	119545256	17947837	119545256	40273
	TA	avg	71	4092	20	**2334154**	**19893200**	4668308	39786400	14494
		max	412	55321	232	8378151	74240975	16756302	148481950	53359
Robin15	BA	avg	16	717	1	2776200	31544574	2776200	31544574	9345
		max	100	7680	4	15198075	238617470	15198075	238617470	71576
	TGBA	avg	15	664	0	2682881	30981263	**2682881**	**30981263**	**9192**
		max	82	5792	4	13129644	225905117	13129644	225905117	62328
	TA	avg	95	6048	15	**2027251**	**17168855**	4054503	34337710	12401
		max	481	43547	143	8169472	68645888	16338944	137291776	48675

Table 3.1: Comparison of the three approaches on toy examples with **random formulæ, when counterexamples do not exist.**

			Violated properties (a counterexample exists)							
			Automaton			Full product		Emptiness check		
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	BA	avg	17	738	1	8 591 425	45 706 072	608 443	2 310 272	3 932
		max	70	3 568	5	50 366 303	365 239 596	3 599 887	21 728 436	24 720
	TGBA	avg	14	646	0	8 118 953	42 865 329	600 828	2 325 266	3 892
		max	47	3 392	5	50 297 469	364 396 821	3 632 979	22 095 045	24 665
	TA	avg	90	5 683	15	**8 592 222**	**33 972 863**	**546 562**	**1 886 028**	**3 667**
		max	656	62 947	132	56 166 663	224 108 070	3 178 506	11 478 152	21 823
Ring6	BA	avg	20	832	1	3 378 684	28 026 221	876 552	5 897 370	1 369
		max	64	4 460	3	11 012 779	123 266 244	6 792 681	70 374 257	15 784
	TGBA	avg	17	713	1	3 163 763	26 260 802	854 229	5 805 080	1 329
		max	51	3 788	4	9 855 589	108 310 156	5 914 415	65 020 741	13 999
	TA	avg	125	7 462	14	**2 846 468**	**18 131 253**	**650 592**	**3 881 701**	**1 170**
		max	472	41 068	162	8 960 960	62 840 996	4 923 301	36 659 892	9 895
FMS5	BA	avg	16	553	1	10 949 858	101 710 172	1 765 510	11 843 265	2 319
		max	104	4 524	3	42 512 695	432 252 072	14 455 789	120 456 128	25 056
	TGBA	avg	14	496	1	10 174 866	95 061 682	1 602 719	10 636 471	2 148
		max	96	4 200	3	40 672 843	404 072 123	11 075 871	110 774 947	18 644
	TA	avg	90	4 112	9	**8 418 689**	**70 784 299**	**1 051 522**	**7 742 600**	**1 956**
		max	390	22 702	79	31 864 749	267 656 316	8 518 894	63 178 516	16 558
Kanban5	BA	avg	15	391	1	11 646 608	127 363 073	1 204 137	9 094 855	1 518
		max	87	2 696	4	43 038 083	587 543 197	13 023 468	185 410 581	25 805
	TGBA	avg	13	355	0	10 952 627	119 272 618	1 153 850	8 728 297	1 473
		max	87	2 184	3	43 038 083	587 543 197	13 023 468	185 410 581	26 750
	TA	avg	75	2 870	6	**8 429 948**	**83 120 910**	**670 472**	**5 873 480**	**1 224**
		max	473	30 375	92	31 602 068	331 833 407	7 673 335	77 364 100	14 239
Philo10	BA	avg	14	641	0	20 491 935	206 666 412	1 551 357	7 673 496	2 713
		max	74	5 928	4	84 068 722	1 377 479 362	9 032 250	53 881 112	17 065
	TGBA	avg	13	590	0	19 118 012	195 258 228	1 518 296	7 522 623	2 651
		max	74	5 928	3	84 068 722	1 377 479 362	9 032 250	53 881 112	15 880
	TA	avg	92	5 690	14	**16 012 398**	**148 019 289**	**900 840**	**6 499 314**	**2 545**
		max	356	46 498	170	66 535 322	696 331 784	7 342 016	64 470 840	25 536
Robin15	BA	avg	24	1 113	1	5 866 595	71 526 501	1 719 685	18 722 554	5 549
		max	84	5 928	6	17 192 350	282 229 101	13 627 374	191 356 248	53 159
	TGBA	avg	21	965	1	5 514 573	67 871 153	1 676 314	18 343 280	5 475
		max	74	5 928	7	15 127 431	281 880 971	13 627 374	191 356 248	52 962
	TA	avg	164	11 366	21	**4 869 334**	**41 343 229**	**1 249 161**	**10 420 582**	**4 052**
		max	854	62 419	107	11 791 872	104 488 704	7 767 216	74 241 202	28 352

Table 3.2: Comparison of the three approaches on toy examples with **random formulæ, when counterexamples exist.**

			Verified properties (no counterexample)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	BA	avg	9	93	0	3 530 202	17 259 383	3 530 202	17 259 383	23 976
		max	32	494	2	14 572 104	105 106 990	14 572 104	105 106 990	99 998
	TGBA	avg	4	52	0	3 071 854	15 745 483	**3 071 854**	**15 745 483**	**21 163**
		max	17	357	2	9 551 018	87 453 872	9 551 018	87 453 872	70 107
	TA	avg	50	564	1	**3 018 533**	**11 771 410**	6 037 065	23 542 820	41 176
		max	184	3 362	3	10 750 205	42 228 050	21 500 410	84 456 100	146 637
Ring6	BA	avg	10	104	0	1 565 890	9 845 113	1 565 890	9 845 113	2 345
		max	34	494	2	4 945 096	44 945 800	4 945 096	44 945 800	9 306
	TGBA	avg	5	64	0	1 121 590	7 920 270	**1 121 590**	**7 920 270**	**1 903**
		max	25	357	2	3 873 576	38 047 692	3 873 576	38 047 692	7 910
	TA	avg	58	682	1	**1 118 511**	**6 494 556**	2 237 021	12 989 111	3 697
		max	183	3 353	3	3 383 372	22 481 284	6 766 744	44 962 568	10 865
FMS5	BA	avg	9	84	1	3 966 306	28 568 464	3 966 306	28 568 464	5 596
		max	25	340	2	14 865 917	130 390 060	14 865 917	130 390 060	24 590
	TGBA	avg	4	41	0	3 030 750	23 589 355	**3 030 750**	**23 589 355**	**4 674**
		max	12	212	2	11 882 973	114 787 553	11 882 973	114 787 553	23 008
	TA	avg	48	515	1	**2 259 230**	**17 600 147**	4 518 460	35 200 294	8 482
		max	171	4 256	3	9 764 223	79 707 273	19 528 446	159 414 546	36 382
Kanban5	BA	avg	8	62	0	3 784 743	33 114 074	3 784 743	33 114 074	5 190
		max	32	221	2	15 768 592	198 359 739	15 768 592	198 359 739	29 290
	TGBA	avg	4	33	1	3 098 615	29 435 137	**3 098 615**	**29 435 137**	**4 712**
		max	21	168	2	13 014 212	187 004 177	13 014 212	187 004 177	26 143
	TA	avg	36	306	1	**2 083 777**	**19 156 589**	4 167 555	38 313 178	7 883
		max	140	2 049	2	11 039 112	106 534 190	22 078 224	213 068 380	48 724
Philo10	BA	avg	9	80	0	5 478 383	33 455 338	5 478 383	33 455 338	11 245
		max	31	547	1	16 695 369	140 088 618	16 695 369	140 088 618	41 598
	TGBA	avg	4	35	0	4 255 979	**26 793 678**	**4 255 979**	**26 793 678**	**9 064**
		max	21	216	2	12 974 557	112 561 242	12 974 557	112 561 242	39 806
	TA	avg	47	501	1	**3 229 690**	27 631 036	6 459 379	55 262 071	20 439
		max	289	8 199	6	10 987 384	94 141 317	21 974 768	188 282 634	64 593
Robin15	BA	avg	9	98	0	2 262 431	20 634 259	2 262 431	20 634 259	6 703
		max	38	608	1	5 849 069	76 754 676	5 849 069	76 754 676	20 953
	TGBA	avg	5	59	0	1 782 133	17 996 764	**1 782 133**	**17 996 764**	**5 749**
		max	16	337	1	5 029 881	70 344 648	5 029 881	70 344 648	19 263
	TA	avg	54	618	1	**1 508 082**	**12 171 559**	3 016 163	24 343 119	9 099
		max	239	3 706	4	3 944 448	32 391 168	7 888 896	64 782 336	23 018

Table 3.3: Comparison of the three approaches on toy examples with **weak-fairness formulæ, when counterexamples do not exist.**

			Violated properties (a counterexample exists)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	BA	avg	12	184	1	9 604 699	42 508 151	654 497	2 230 146	4 265
		max	74	1 338	2	34 771 276	235 328 943	7 043 181	27 266 804	46 628
	TGBA	avg	5	78	0	**3 731 947**	**17 751 951**	**522 414**	**1 759 041**	**3 438**
		max	18	351	2	13 508 730	79 589 415	3 696 481	16 733 420	24 769
	TA	avg	95	1 400	1	10 950 499	42 790 086	641 997	2 123 401	4 368
		max	350	5 968	8	30 354 888	120 985 972	7 637 347	29 621 420	51 378
Ring6	BA	avg	18	291	1	4 076 877	30 760 035	1 312 236	7 496 631	1 822
		max	70	2 248	3	6 924 114	96 086 800	3 706 323	34 372 404	6 998
	TGBA	avg	7	123	1	**2 000 787**	**15 868 857**	**890 567**	5 736 568	**1 410**
		max	38	1 168	3	4 670 124	53 246 768	2 766 283	26 917 302	6 101
	TA	avg	128	2 009	2	3 456 785	21 069 783	905 280	**5 156 350**	1 583
		max	388	12 783	13	6 152 232	43 283 696	2 747 578	17 464 827	4 995
FMS5	BA	avg	12	139	0	13 634 313	123 924 884	1 623 286	8 873 430	1 807
		max	46	550	2	38 283 058	436 545 323	11 138 596	82 531 874	14 207
	TGBA	avg	5	68	0	**7 978 129**	**75 701 408**	1 372 877	7 544 397	1 607
		max	23	370	2	27 570 312	293 219 919	6 676 584	53 451 994	9 961
	TA	avg	71	871	1	10 007 988	81 736 556	**845 087**	**5 687 307**	**1 471**
		max	245	3 766	5	27 733 464	249 573 778	6 425 987	51 962 113	11 856
Kanban5	BA	avg	12	133	1	11 528 567	115 461 763	1 541 122	10 510 834	1 738
		max	34	501	2	34 735 366	497 454 792	12 460 358	113 423 964	16 932
	TGBA	avg	5	66	0	**7 040 236**	**73 867 374**	1 344 870	9 279 129	1 562
		max	17	305	2	25 423 161	388 961 954	7 950 633	80 041 971	11 336
	TA	avg	69	849	1	8 102 548	77 515 606	**809 384**	**6 986 502**	**1 440**
		max	237	3 655	5	32 732 392	329 125 090	5 501 242	53 375 472	10 359
Philo10	BA	avg	12	148	0	20 724 516	190 232 774	1 144 023	5 417 407	1 849
		max	44	688	3	52 231 021	996 567 139	7 775 968	45 330 784	13 909
	TGBA	avg	5	69	0	**10 515 366**	**102 622 810**	833 663	**3 762 217**	**1 357**
		max	25	392	3	24 496 745	402 197 001	5 017 174	32 273 634	9 802
	TA	avg	81	1 027	1	16 779 360	150 030 607	**743 065**	5 019 598	1 936
		max	267	4 339	6	43 186 049	394 172 486	6 703 064	55 546 036	21 005
Robin15	BA	avg	21	356	1	7 007 406	78 083 493	1 916 523	16 640 621	5 468
		max	73	2 248	2	23 830 436	327 588 082	8 911 386	101 808 220	28 268
	TGBA	avg	9	157	1	**4 006 015**	**46 910 526**	1 486 839	14 360 747	4 691
		max	44	1 168	2	14 802 883	195 152 925	5 666 009	78 112 291	21 619
	TA	avg	143	2 467	2	5 832 674	48 201 707	**1 347 956**	**10 786 002**	**4 257**
		max	466	12 783	12	18 204 684	151 155 748	6 486 142	53 115 008	18 506

Table 3.4: Comparison of the three approaches on toy examples with **weak-fairness formulæ, when counterexamples exist.**

				Verified properties (no counterexample)								
				Automaton			Full product		Emptiness check			
				st.	tr.	T_φ	st.	tr.	st.	tr.	T	
PolyORB 3/3/2	RND	BA	avg	18	781	1	111 714	337 573	111 714	337 573	612	
			max	136	7 680	5	373 039	2 138 028	373 039	2 138 028	2 097	
		TGBA	avg	16	715	1	109 158	328 976	**109 158**	**328 976**	**601**	
			max	136	5 792	4	373 039	2 138 028	373 039	2 138 028	2 207	
		TA	avg	94	5 768	17	**94 283**	**205 464**	188 566	410 928	1 037	
			max	481	43 547	196	261 318	604 061	522 636	1 208 122	2 847	
	WFair	BA	avg	11	157	0	132 102	369 787	132 102	369 787	713	
			max	71	1 294	2	799 355	2 630 204	799 355	2 630 204	4 339	
		TGBA	avg	5	72	0	96 435	277 426	**96 435**	**277 426**	**524**	
			max	18	337	2	368 898	1 450 005	368 898	1 450 005	2 049	
		TA	avg	79	1 106	1	**120 479**	**262 456**	240 958	524 913	1 302	
			max	351	5 920	9	633 208	1 408 943	1 266 416	2 817 886	6 806	
	Φ_1	BA	–	7	576	1	345 241	760 491	**345 241**	**760 491**	**1 675**	
		TGBA	–	7	576	1	345 241	760 491	**345 241**	**760 491**	**1 688**	
		TA	–	80	14 590	8	**342 613**	**742 815**	685 226	1 485 630	3 391	
MAPK 8	RND	BA	avg	12	374	1	2 894 799	25 769 242	2 894 799	25 769 242	5 477	
			max	79	3 813	5	13 636 352	147 555 158	13 636 352	147 555 158	29 292	
		TGBA	avg	10	333	1	2 808 973	25 212 509	**2 808 973**	**25 212 509**	**5 325**	
			max	64	3 813	5	13 636 352	147 555 158	13 636 352	147 555 158	28 979	
		TA	avg	50	2 239	9	**1 724 291**	**20 203 618**	3 448 582	40 407 237	9 591	
			max	336	22 614	146	8 469 258	108 847 708	16 938 516	217 695 416	53 824	
	WFair	BA	avg	10	82	0	4 290 714	38 013 034	4 290 714	38 013 034	7 887	
			max	46	437	2	17 600 440	177 748 756	17 600 440	177 748 756	33 063	
		TGBA	avg	5	39	0	3 898 645	34 822 961	**3 898 645**	**34 822 961**	**7 344**	
			max	21	198	2	14 452 198	162 156 912	14 452 198	162 156 912	31 117	
		TA	avg	44	407	1	**2 112 810**	**24 492 299**	4 225 619	48 984 598	12 092	
			max	171	1 920	3	6 110 748	75 624 744	12 221 496	151 249 488	40 347	
	Φ_2	BA	–	6	165	0	46 494	302 350	**46 494**	**302 350**	**45**	
		TGBA	–	6	165	1	46 494	302 350	**46 494**	**302 350**	**50**	
		TA	–	9	293	2	**33 376**	**289 235**	66 752	578 470	95	

Table 3.5: Comparison of the three approaches for the **case studies when counterexamples do not exist.**

				Violated properties (a counterexample exists)								
				Automaton			Full product		Emptiness check			
				st.	tr.	T_φ	st.	tr.	st.	tr.	T	
PolyORB 3/3/2	RND	BA	avg	15	839	1	137 958	397 495	54 412	130 427	294	
			max	74	7 004	11	1 520 649	5 898 222	192 692	514 905	1 088	
		TGBA	avg	14	760	1	132 232	380 305	53 901	128 960	291	
			max	65	7 004	9	1 295 661	5 045 814	192 692	514 905	1 049	
		TA	avg	81	6 223	29	**135 237**	**296 081**	**56 518**	**126 300**	**310**	
			max	540	59 971	631	1 377 784	2 976 709	243 950	547 459	1 324	
	WFair	BA	avg	11	150	0	190 987	472 205	112 918	279 490	600	
			max	43	788	2	604 611	1 951 062	328 442	1 419 816	1 794	
		TGBA	avg	5	66	0	**96 849**	**258 704**	**69 053**	**177 391**	**372**	
			max	22	516	2	432 240	1 584 372	193 264	1 031 898	1 110	
		TA	avg	82	1 157	1	201 879	434 747	113 705	250 573	612	
			max	262	4 636	6	560 685	1 194 198	254 328	594 751	1 330	
MAPK 8	RND	BA	avg	15	578	1	29 980 338	454 492 066	1 939 885	13 406 649	2 890	
			max	61	4 398	2	141 058 746	2 921 365 220	13 284 206	120 933 281	24 053	
		TGBA	avg	13	513	1	27 266 223	405 363 157	1 850 695	12 575 125	2 702	
			max	59	4 298	2	91 059 214	1 963 331 216	13 284 206	111 916 150	21 771	
		TA	avg	88	4 818	13	**22 458 466**	**298 950 743**	**967 363**	**9 259 668**	**2 346**	
			max	334	35 401	164	90 049 281	1 300 904 178	5 241 327	61 655 512	13 808	
	WFair	BA	avg	13	176	0	34 857 814	493 530 217	2 249 726	16 186 296	3 515	
			max	46	541	2	111 139 060	1 700 666 096	15 349 474	123 057 668	26 758	
		TGBA	avg	6	81	0	**20 904 101**	**300 416 331**	1 962 765	13 896 236	3 129	
			max	23	414	2	101 813 441	1 681 905 871	9 463 562	72 266 873	16 928	
		TA	avg	88	1 212	1	27 535 941	358 622 897	**1 112 324**	**11 462 361**	**2 869**	
			max	245	3 749	4	93 441 682	1 258 076 980	5 203 746	59 642 346	14 489	

Table 3.6: Comparison of the three approaches for the **case studies when counterexamples exist.**

Table 3.3 and Table 3.4 show toy models against weak-fairness formulas.

Table 3.5 and Table 3.6 show the results of the two cases studies against random, weak-fairness, and dedicated formulas issued from the studies.

These tables separate cases where formulas are verified from cases where they are violated. In the former (Table 3.1, 3.3 and 3.5), no counterexample are found and the full state-space had to be explored; in the latter (Table 3.2, 3.4 and 3.6) the on-the-fly exploration of the state-space stops as soon as the existence of a counterexample is computed.

All values shown in all tables are averaged over 100 different formulas (except for the lines Φ_1 and Φ_2 in Table 3.5, where only one formula is used). For instance we checked Peterson5 against 100 random formulas that had no counterexample, and against 100 random formulas that had a counterexample. The average and maximum are computed separately on these two sets of formulas.

Column-wise, these tables show the average and maximum sizes (states and transitions) of: (1) the automata $A_{\neg\varphi_i}$ expressing the properties φ_i; (2) the products $A_{\neg\varphi_i} \otimes \mathcal{K}$ of the property with the model; and (3) the subset of this product that was actually explored by the emptiness check. For verified properties, the emptiness check of TGBA and BA always explores the full product so these sizes are equal, while the emptiness check of TA always performs two passes on the full product so it shows double values. On violated properties, the emptiness check aborts as soon as it finds a counterexample, so the explored size is usually significantly smaller than the full product.

The emptiness check values show a third column labeled "T": this is the time in $\frac{1}{100^e}$ of seconds (a.k.a. centiseconds) spent doing that emptiness check, including the on-the-fly computation of the subset of the product that is explored. The time spent constructing the property automata $A_{\neg\varphi_i}$ is shown in column "T_φ" (it is negligible compared to that of the emptiness check "T").

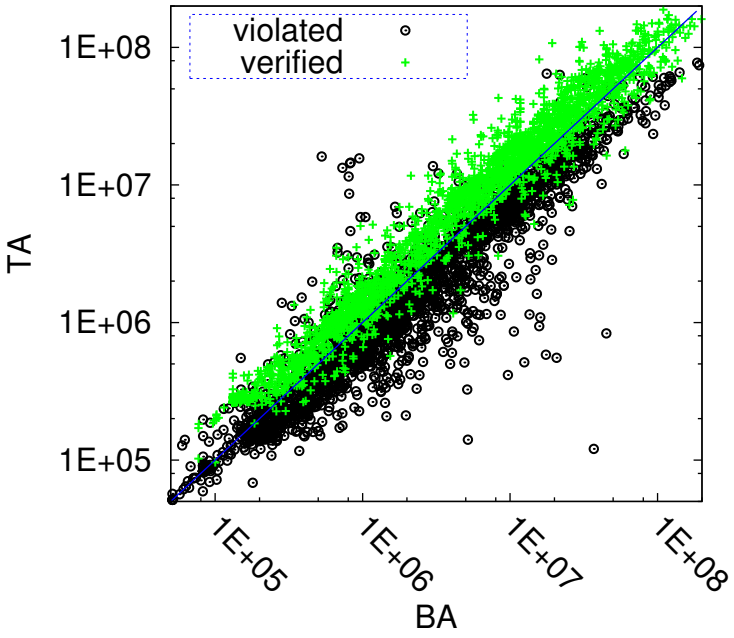
Figure 3.8 compares the number of visited transitions when running the emptiness check; plotting TA against BA and TGBA. This gives an idea of their relative performance. Each point corresponds to one of the 5600 evaluated formulas (2800 violated with counterexample as black circles, and 2800 verified having no counterexample as green crosses). Each point below the diagonal is in favor of TA while others are in favor of the other approach. Axes are displayed using a logarithmic scale.

Figure 3.9 compares the number of visited transitions between BA and TGBA. Each point below the diagonal is in favor of TGBA (this clearly shows that BA are less efficient than TGBA).

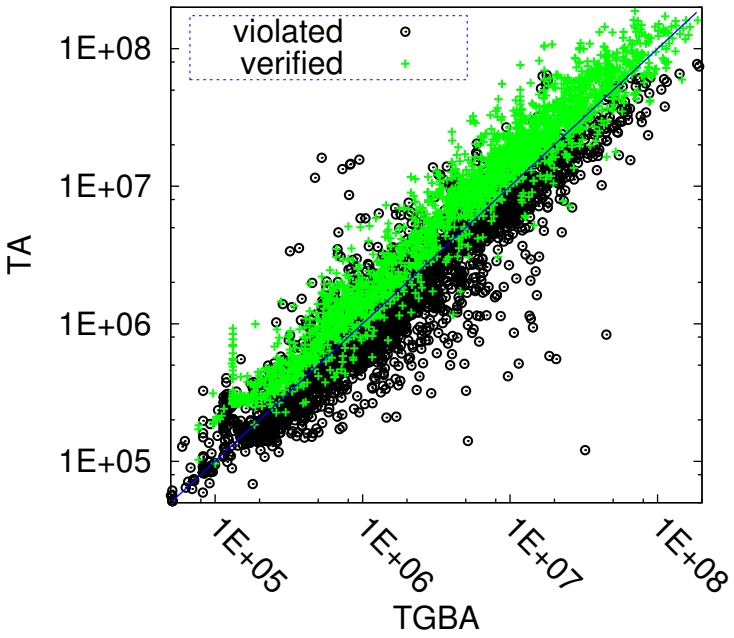
All these experiments were run on a 64bit Linux system running on an Intel(R) 64-bit Xeon(R) @2.00GHz, with 10GB of RAM.

3.6.4 Discussion (TA two-pass emptiness check problem)

Although the state-space of cases studies can be very different from those of random state-spaces [68], a first look at our results confirms two facts already observed by Geldenhuys and Hansen using random state-spaces [46]: (1) although the TA constructed from properties are usually a lot larger than BA, the average size of the full product is smaller thanks to the more deterministic nature of the TA. (2) For violated properties, the TA approach explores less states and transitions on the average than the BA.

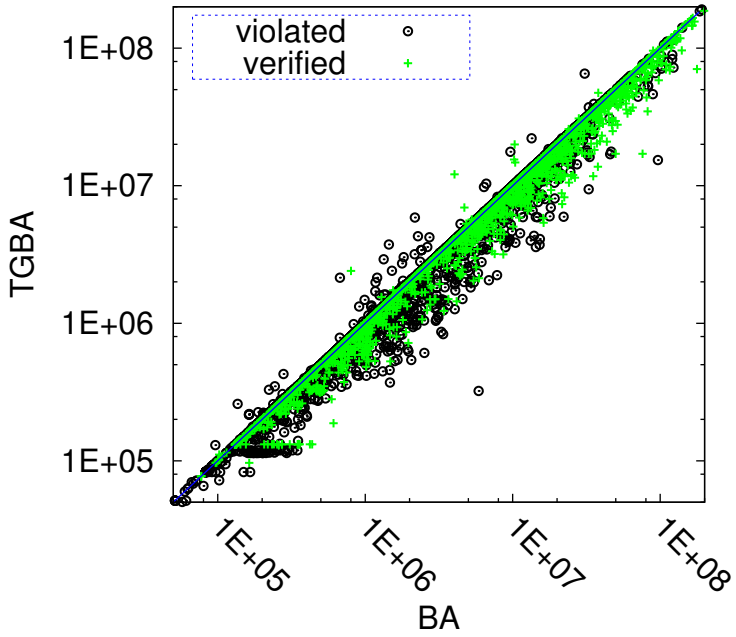


(a) TA against BA approaches



(b) TA against TGBA approaches

Figure 3.8: Performance (number of transitions explored by the emptiness check) of TA against BA and TGBA.



(a) TGBA against BA approaches

Figure 3.9: Performance (number of transitions explored by the emptiness check) of TGBA against BA.

We complete this picture by showing execution times, by separating verified properties from violated properties, and by also evaluating the TGBA approach.

On verified properties, the results are very straightforward to interpret: the BA are slightly worse than the TGBA because the degeneralization increases the size of the property automaton. In fact, the average number of acceptance conditions needed in random formulas (Table 3.1 and 3.5) is so close to 1 that the degeneralization barely changes the sizes of the automata. With weak-fairness formulas (Table 3.3 and 3.5), the number of acceptance conditions is greater, so TGBA are favored over BA. Surprisingly, both TGBA and BA, although they are not tailored to *stutter-invariant* properties like TA, appear more effective to prove that a *stutter-invariant* property is verified. In the three tables, although the full product of the TA approach is smaller than the other approaches, it has to be explored twice (as explained in section 3.4): the emptiness-check consequently explores more states and transitions. This double exploration is not enough to explain the big runtime differences. Two other subtle implementation details of the synchronous products contribute to the time difference:

- To synchronize a transition of a Kripke structure with a transition (or a state in case of stuttering) of a TA, we must compute the symmetric difference (i.e., the changeset) $l(s) \oplus l(d)$ between the labels of the source and destination states. The same synchronization in the TGBA and BA approaches requires to know only the source label.

  Computing these labels is a costly operation in CheckPN because Petri net marking are compressed in memory to save space. Although we implemented some (limited) cache to

alleviate the number of such label computation, profiling measures revealed the TA approach was 3 times slower than the TGBA and BA approaches, but that labels where computed 9 times more.

- A second implementation difference is that the transitions of testing automata are labeled by elements of Σ, while in the Spot implementation of TGBA and BA, the transitions are labeled by elements of 2^{Σ}. That means that once $l(s) \oplus l(d) \in \Sigma$ has been computed, we can use a hash table to immediately find matching transitions of the testing automaton. In the TGBA and BA implementations, we linearly scan the list of transitions of the property automaton until we find one compatible with $l(s)$.

In order to protect the results against the influence of various optimizations, implementation tricks, and the central processor and memory architecture, Geldenhuys and Hansen [46] have decided (in their work about TA) to report only the number of explored states and transitions, and not the number of bytes and milliseconds consumed by their implementations. They found that the number of states gives a reliable indication of the memory required, and, similarly, the number of transitions a reliable indication of the time consumption. That is why in our discussion (in this chapter and in the next chapters), we focus on the number of explored states and transitions more than time consumption.

On violated properties, it is harder to interpret these tables because the emptiness check returns as soon as it finds a counterexample. Changing the order in which non-deterministic transitions of the property automaton are iterated is enough to change the number of states and transitions to be explored before a counterexample is found: in the best case the transition order leads the emptiness check straight to an accepting cycle; in the worst case, the algorithm explores the whole product until it finally finds an accepting cycle. Although the emptiness check algorithms for the three approaches share the same routines to explore the automaton, they are all applied to different kinds of property automata, and thus provide different transition orders.

This ordering luckiness explains why the BA approach sometimes outperforms the TGBA one.

We believe that the TA, since they are more deterministic, are less sensible to this ordering. They also explore a smaller state-space on the average. This smaller exploration is not always tied to a good runtime because of the extra computation of labels discussed in this section. Again, looking at the average number of transitions explored by the emptiness check indicates that the TA approach would outperform the others if labels computation would have been cheap.

3.7 Conclusion

Geldenhuys and Hansen evaluated the performance of the BA and TA approaches with small random Kripke structures checked against LTL formulas taken from the literature [46]. In this work, we have completed their experiments by using actual models and different kinds of formulas (random formulas not trivially verifiable, random formulas expressing weak-fairness formulas, and a couple of real formulas), by evaluating the TGBA approach, and by distinguishing violated formulas and verified formulas in the benchmark.

In TA approach, an unfortunate consequence of having two different ways of accepting infinite words (livelock or Büchi), is that the emptiness-check algorithm required during model checking

must perform two passes on the whole state-space in the worst case.

For verified formulas, we found that the state-space reduction achieved by the TA approach was not enough to compensate for the two-pass emptiness check this approach requires. It is therefore better to use the TGBA approach to prove that a *stutter-invariant* formula is verified and TA approach in an earlier "debugging phase".

When the formulas are violated, the TA approach usually processes less transitions than the BA approach and TGBA to find a counterexample. This approach should therefore be a valuable help to debug models (i.e. when counterexamples are *expected*). This is especially true on random formulas. With weak-fairness formulas, generalized automata are advantaged and are able to beat the TA on the average in 4 of our 8 examples (Peterson5, Philo10, Ring6, PolyORB 3/2/2).

In the next chapter, we propose some optimizations in order to omit the second pass in TA approach, in particular when no livelock-accepting states is encountered during the first pass. We also propose STA (*Single-pass Testing Automata*), a transformation of TA that never requires such a second pass.

In Chapter 5, we propose a single-pass and generalized testing automata, called TGTA (*Transition-based Generalized Testing Automata*). TGTA combines ideas from TA and TGBA. The basic idea is to have a form of testing automata with transition-based generalized acceptance conditions, which allows us to modify the automata construction in order to remove the second pass of the emptiness check of the product.

We also noted that making a product between a TA and a Kripke structure is not a good idea when computing the label of the Kripke structure is expensive. A more efficient model checker using testing automata should probably represent the model using labeled Kripke structure in which the transition labels represent the symmetric difference (i.e., the changeset) between the label of the source and destination state. This optimization is exploited in the chapter 6 concerning the symbolic model checking using TGTA.

Part II

Improving Testing Automata for Explicit LTL Model Checking

CHAPTER 4

Improving the Testing Automata Approach

Contents

4.1 Introduction

In the previous chapter, we evaluated the use of Testing Automata (TA) for the model checking of stutter-invariant properties. We have shown that the TA approach is efficient when the formula to be verified is violated (i.e., a counterexample exists). This is not the case when the property is verified since the entire state-space has to be visited twice to check for each acceptance mode of a TA (Büchi-acceptance or livelock-acceptance).

In this chapter we improve the TA approach in two ways. First, we propose optimizations of the emptiness check algorithm that avoid the second pass when it is possible. Second, we propose a transformation of TA into a normal form that never requires such a second pass, called *Single-pass Testing Automata* (STA).

Although STA are more constrained than TA, we provide a transformation that automatically translates the latter into the former. Then we prove the correctness of the single pass emptiness check for STA. We also present an optimization that allows to build a smaller STA.

We have implemented the algorithms of STA approach in Spot, our model checking library. We are thus able to compare them with the "traditional" algorithms we used on Testing Automata

(TA) and Transition-based Generalized Büchi Automata (TGBA). These experiments show that STA compete well on our examples.

4.2 Improving the Emptiness check by avoiding the second pass in particular cases

In the evaluation of TA approach presented in the previous chapter, we have implemented the heuristic for livelock detection proposed by Geldenhuys and Hansen [46] to avoid the second pass of the emptiness check algorithm. Unfortunately, this heuristic fails in certain cases to detect livelock cycles. In addition to this heuristic, we add the following improvements to the first pass in order to avoid the second pass in the following particular cases:

1. In the previous chapter, we have shown that the second pass is only used to detect livelock-accepting cycles. Therefore, if no livelock-accepting state is visited during the first pass (i.e., the product does not contain livelock-accepting states), then the second pass can be disabled: this can be done by simply adding a variable *Gseen* in Algorithm 2 (page 49), where *Gseen* is a flag that records if a livelock-accepting state is detected during the exploration of the product by the first pass. In the experiments presented at the end of this chapter, this optimization greatly improves the performance of the TA approach in the cases where the formula is verified.

2. A cycle detected in the product during the first pass is also accepted if it contains a livelock-accepting state (s,q) of the product such that q has no successor in the TA. Indeed, from this state, a run of the product can only execute stuttering transitions. Therefore, a cycle containing this state, is composed only by stuttering transitions: it is a livelock-accepting cycle.

This second optimization may detect some livelock-accepting cycles, but it misses livelock-accepting cycles that are mixed with non-stuttering transitions in the same SCC, as discussed in section 3.5.2 (page 48) of the previous chapter.

We now introduce a class of TA where the second optimization will always detect any livelock-accepting cycle during the `first-pass`.

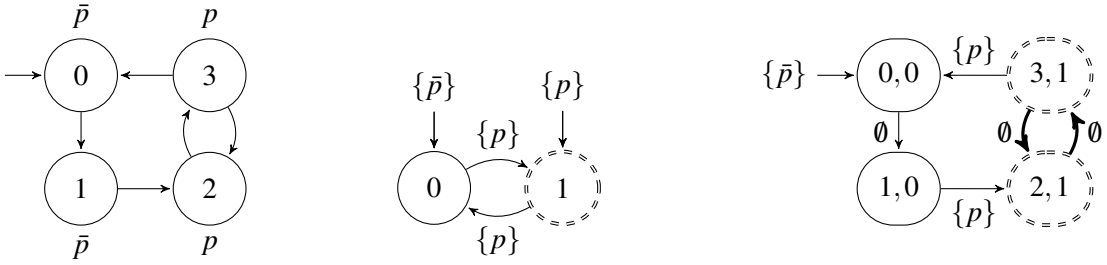
4.3 Converting a TA into a Single-pass Testing Automaton (STA)

In this section, we introduce STA, a transformation of TA into a normal form such that livelock-accepting states have no successors, and therefore STA approach does not need the second pass of the emptiness check of TA approach. This contribution improves the efficiency of the model checking (this will be experimentally evaluated in section 4.4). In addition, STA simplify the implementation (and the optimization) of the emptiness check algorithm as it renders unnecessary the implementation of the second pass.

4.3.1 Single-pass Testing Automata (STA)

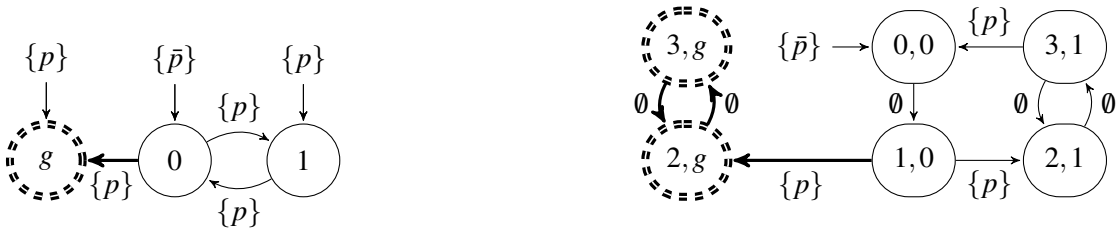
Definition 26 (STA)**.** *A Single-pass Testing Automaton (STA) is a TA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$ over Σ such that $\delta \cap (\mathcal{G} \times 2^{\Sigma} \times Q) = \emptyset$. In other words, an STA is a TA in which every livelock-accepting state has no successors.*

4.3.2 Construction of an STA from a TA



(a) a Kripke structure $\mathcal{K}$ (b) a TA $\mathcal{T}$ for $\mathsf{FG}\,p$ (c) $\mathcal{K} \otimes \mathcal{T}$

Figure 4.1: Example of a product between a Kripke structure $\mathcal{K}$ and a TA $\mathcal{T}$ of $\mathsf{FG}\,p$. The bold cycle of $\mathcal{K} \otimes \mathcal{T}$ is livelock-accepting.



(a) an STA $\mathcal{T}^{+}$ for $\mathsf{FG}\,p$ (b) $\mathcal{K} \otimes \mathcal{T}^{+}$

Figure 4.2: Impact on the product of using STA $\mathcal{T}^{+}$ instead of TA $\mathcal{T}$. Bold states and transitions are addition relative to Figure 4.1.

Property 5 formalizes the construction of an STA from a TA. We can transform a TA into an STA by adding an unique livelock-accepting state g (i.e., in STA, $G = \{g\}$), and adding a transition (q,k,g) for any transition (q,k,q') that goes into a livelock-accepting state $q' \in \mathcal{G}$ of the original automaton. In addition, if q' has no successors then q' can be removed, since it is bisimilar to the new state g.

Property 5. *Let $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$ be a TA, the equivalent STA is $\mathcal{T}' = \langle Q', I', U', \delta'_{\oplus}, \mathcal{F}, \{g\} \rangle$ where*

- *$Q' = (Q \setminus G_{\emptyset}) \cup \{g\}$ where $G_{\emptyset} = \{q \in \mathcal{G} \mid (\{q\} \times \Sigma \times Q) \cap \delta = \emptyset\}$ is the set of states of $\mathcal{G}$ that have no successors, and $g \notin Q$ is a new state,*
- *$I' = I \cup \{g\}$ if $\mathcal{G} \cap I \neq \emptyset$, $I' = I$ otherwise,*
- *$\delta'_{\oplus} = (\delta \setminus (Q \times \Sigma \times G_{\emptyset})) \cup \{(q,k,g) \mid (q,k,q') \in \delta, q' \in \mathcal{G}\}$,*

- $\forall q \in I, U'(q) = U(q)$ *and* $U'(g) = \bigcup_{q \in (\mathcal{G} \cap I)} U(q)$.

is such that $\mathscr{L}(\mathcal{T}') = \mathscr{L}(\mathcal{T})$.

Figure 4.2a shows how the TA from Figure 4.1b was transformed into an STA using Property 5. The idea behind this transformation is that any livelock-accepting execution of $\mathcal{T}$ will be mapped to an execution of $\mathcal{T}^+$ that is captured by the new state g. The new g state has an impact on the size of the product (Figure 4.2b), but the strongly connected components of this new product no longer mix non-stuttering transitions and livelock-accepting cycles: this renders the `second-pass` of Algorithm 3 useless. However, adding an artificial state in STA will increase the size of the product automaton to be explored by the emptiness check (i.e. the `first-pass` of Algorithm 2) . In the next chapter, we present a new improvement of TA, called TGTA that removes the `second-pass` without adding an artificial state.

The objective of STA is to isolate in the product the exploration of the parts that are composed only by livelock-accepting states and stuttering transitions, like the bold part of the product represented in the Figure 4.2b. In this kind of sub-products, it is easy to find all livelock accepting cycles.

The STA emptiness check algorithm is the `first-pass` of the TA emptiness check algorithm without the `second-pass` procedure. In other words, in STA approach, the emptiness check is only Algorithm. 2 (page 49) without line 6.

4.3.3 Correctness of the one-pass emptiness check using STA

In the following lemmas, $\mathcal{K}$, $\mathcal{T}$, $\mathcal{T}^+$ denote respectively a Kripke structure, a TA and an STA.

The `first-pass` is an SCC-based algorithm, it computes the set of all MSCCs (i.e., Maximal SCCs) of the product automaton. Therefore, in order to prove that the `first-pass` is sufficient to detect all livelock-accepting cycles, we prove that in $\mathcal{K} \otimes \mathcal{T}^+$, searching for all livelock-accepting cycles is equivalent to searching for all MSCCs that are only composed of stuttering transitions and livelock-accepting states. In Algorithm. 2, line 17 allows to detect this kind of MSCCs.

Lemma 1. *In a product automaton* $\mathcal{K} \otimes \mathcal{T}$*:* ***if*** *one MSCC M contains a product state* (s,q) *such that q is a livelock-accepting state that has no successors in* $\mathcal{T}$*,* ***then*** *M is only composed of stuttering transitions and livelock-accepting states.*

Proof. q has no successors in the TA $\mathcal{T}$, therefore from q, a run of $\mathcal{T}$ can only execute stuttering transitions: it stays in the same livelock-accepting state q. Consequently, all product states of M are connected by stuttering transitions. In addition, they have the same livelock-accepting state as TA component (q), therefore by Definition 25 all states of M are livelock-accepting. □

Lemma 2. *In a product automaton* $\mathcal{K} \otimes \mathcal{T}^+$*: one MSCC M contains a livelock-accepting state* ***if and only if*** *M is only composed of stuttering transitions and livelock-accepting states.*

Proof. $(\Longrightarrow)$ If an MSCC M contains a livelock-accepting state (s,q) of $\mathcal{K} \otimes \mathcal{T}^+$, then q is a livelock-accepting state that has no successors in $\mathcal{T}^+$ because in an STA every livelock-accepting state has no successors. The proof follows from Lemma 1 applied to $\mathcal{K} \otimes \mathcal{T}^+$.
$(\Longleftarrow)$ Any state of M is livelock-accepting.

□

The difference between lemma 1 and lemma 2 is that the livelock-accepting states of STA have no successors, while those of a TA can. Therefore, the following lemma is true only for STA.

Lemma 3. *In the product automaton $\mathcal{K} \otimes \mathcal{T}^+$: there exists at least one livelock-acceptance cycle C* ***if and only if*** *there exists at least one non trivial MSCC M such that M is only composed of stuttering transitions and livelock-accepting states.*

Proof. ($\Longrightarrow$): C contains at least one livelock-accepting state, therefore applying Lemma 2 with M is the MSCC containing C allows us to conclude.
($\Longleftarrow$): M is non-trivial (it contains at least two states or a single state with a self-loop), therefore M contains at least one non-trivial cycle only composed of stuttering transitions and livelock-accepting states. This cycle is the livelock-accepting cycle C. □

In Algorithm. 2, the `first-pass` computes all MSCCs and line 17 allows to detect only the MSCCs satisfying Lemma 3. Therefore the STA emptiness check algorithm reports one cycle **if and only if** this cycle is a livelock-accepting cycle or a Büchi-accepting cycle.

Emptiness check optimizations According to Lemma 2, it is sufficient to verify that an MSCC contains a livelock-accepting state at line 17 of Algorithm. 2. Therefore, computing *SCC*.top().k is not necessary for detecting livelock-accepting cycles.

4.3.4 STA optimization

The goal of this optimization is to reduce the number of transitions in STA, by exploiting a special property of livelock-accepting states that are also Büchi-accepting. We begin by introducing a definition to distinguish these particular states:

Definition 27 (Fully-accepting state)**.** *In a TA (or in a product of a TA with a Kripke structure), a state that is both livelock-accepting and Büchi-accepting, is called* fully-accepting *state.*

Definition 26 previously proposed for STA introduces a constraint on livelock-accepting states. This constraint (which we call in the following "livelock-constraint") allowed us to remove the `second-pass` of the emptiness check. However, in order to transform a TA into an STA that satisfies this "livelock-constraint", we have to add an artificial state g and artificial transitions having g as destination (Property 5).

In this optimization, we start by introducing Property 6 that allows us to deduce that the fully-accepting states do not require the second pass of the emptiness check. Then, we propose a new STA definition that removes the "livelock-constraint" for the fully-accepting states and therefore reduces the number of artificial transitions added during the transformation of a TA into an STA.

The goal of the following Lemma 4 and Property 6 is is to prove that every cycle C that contains a fully-accepting state s_f is an accepting cycle . In addition, C is necessarily included in an MSCC M detected by the `first-pass` of the emptiness check.

We remind that in a product of a TA with a Kripke structure, the definition of an accepting cycle does not depend only on its states but also on its transitions:

- On the one hand, any cycle that contains a livelock-accepting state, is considered as an accepting cycle if it is only composed of stuttering transitions.

- On the other hand, a cycle that contains a Büchi-accepting state, must contains a non-stuttering transition, to be considered an accepting cycle.

In the following lemma 4, we distinguish a particular kind of cycles that can be reported as accepting without checking if they contain or not non-stuttering transitions:

Lemma 4. *In a product automaton $\mathcal{K} \otimes \mathcal{T}$: **if** a cycle C contains a fully-accepting state s_f, **then** C is an accepting cycle.*

Proof. There are two cases depending on the transitions composing the cycle C:

- If C contains at least one non-stuttering transition, then using the fact that s_f is Büchi-accepting, we deduce that C is a Büchi-accepting cycle.
- Otherwise, if C contains only stuttering transitions, in this case we exploit the fact that s_f is a livelock-accepting state to deduce that C is a livelock-accepting cycle.

□

In addition, these particular accepting cycles also verify the following property:

Property 6. *In a product automaton $\mathcal{K} \otimes \mathcal{T}$: for every accepting cycle C containing a fully-accepting state s_f, the MSCC M that includes C is detected by the `first-pass` of Algorithm. 2.*

Proof. There are two cases depending on the transitions of the MSCC M:

- If M includes C and contains at least one non-stuttering transition, then M is detected by the line 16 of the `first-pass`, because M contains a Büchi-accepting state s_f and M contains non-stuttering transitions.
- Otherwise, if M includes C and is only composed of stuttering transitions, then M is detected by the line 17 of the `first-pass`, because M contains a livelock-accepting state s_f and M is only composed of stuttering transitions).

□

We deduce from Property 6 that the "livelock-constraint" is not necessary for fully-accepting states. This leads us to propose the following optimized definition of STA:

Definition 28. *A Single-pass Testing Automaton (STA) is a TA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$ over Σ such that $\delta \cap ((\mathcal{G} \setminus \mathcal{F}) \times 2^{\Sigma} \times Q) = \emptyset$. In other words, an STA is a TA in which every state in $\mathcal{G} \setminus \mathcal{F}$ has no successors.*

Consequently, during the TA to STA transformation described by Property 5, it was unnecessary to add artificial transitions (q,k,g) for any transition (q,k,q') where $q' \in (\mathcal{G} \cap \mathcal{F})$ (i.e., q' is a fully-accepting state).

This optimization of Property 5 is formalized by the following Property 7:

Property 7. *Let $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, \mathcal{G} \rangle$ be a TA, the equivalent STA is $\mathcal{T}' = \langle Q', I', U', \delta'_{\oplus}, \mathcal{F}, \mathcal{G}' \rangle$ where*

- *$\mathcal{G}' = (\mathcal{G} \cap \mathcal{F}) \cup \{g\}$ where $g \notin Q$ is a new state,*
- *$Q' = (Q \setminus G_{\emptyset}) \cup \{g\}$ where $G_{\emptyset} = \{q \in \mathcal{G} \mid (\{q\} \times \Sigma \times Q) \cap \delta = \emptyset\}$ is the set of states of $\mathcal{G}$ that have no successors,*

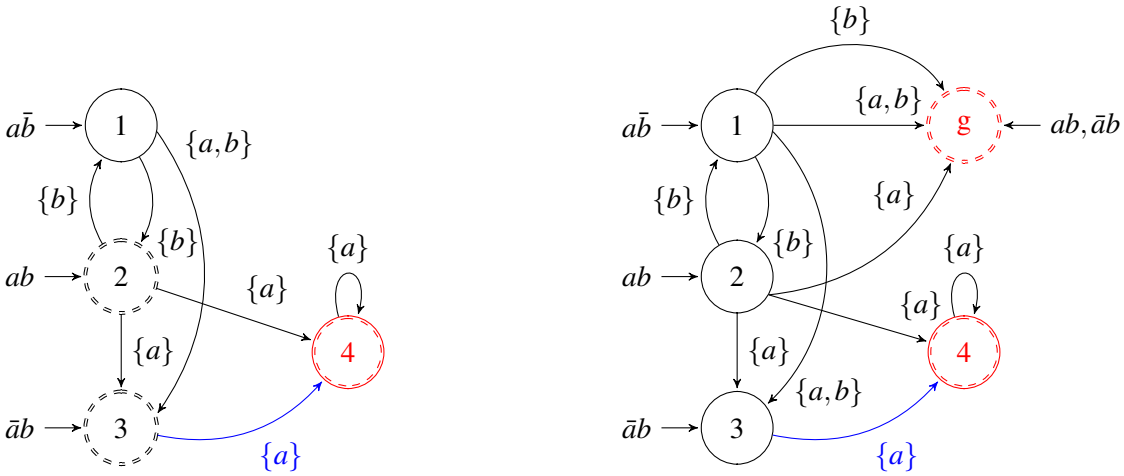


Figure 4.3: Transformation of a TA recognizing $a \mathsf{U} \mathsf{G} b$ into an optimized STA (according to Property 7). The state 4 is a fully-accepting state

- $I' = I \cup \{g\}$ *if* $\mathcal{G} \cap I \neq \emptyset$, $I' = I$ *otherwise,*
- $\delta'_{\oplus} = (\delta \setminus (Q \times \Sigma \times G_{\emptyset})) \cup \{(q,k,g) \mid (q,k,q') \in \delta, q' \in ((G \setminus F) \cup G_{\emptyset})\}$,
- $\forall q \in I, U'(q) = U(q)$ *and* $U'(g) = \bigcup_{q \in (\mathcal{G} \cap I)} U(q)$.

is such that $\mathscr{L}(\mathcal{T}') = \mathscr{L}(\mathcal{T})$.

Property 7 is illustrated by figure 4.3 that shows the transformation of a TA into an STA satisfying Definition 28. The state 4 is a fully-accepting state in the TA, therefore in the STA we don't add an artificial transition from state 4 to g.

4.4 Experimental evaluation of the TA improved emptiness check and of STA

This section presents our experimentation conducted under the same conditions as the previous chapter (see section 3.6), i.e., within the same tools Spot and CheckPN and using the same benchmark inputs (formulas and models).

4.4.1 Implementation

Figure 4.4 shows the building blocks we used to implement the different approaches. The TGBA and BA approaches share the same synchronous product and emptiness check, while a dedicated algorithms are required by the TA and STA approaches.

For TA approach, we have implemented the first-pass improvements of section 4.2 that allow to avoid the second-pass in more cases than the implementation of the previous chapter.

For STA approach, our STA emptiness check implementation shares the same first-pass with the TA algorithm and then disables the second-pass. We also disable the heuristic [46] livelock detection for the STA because it is useless.

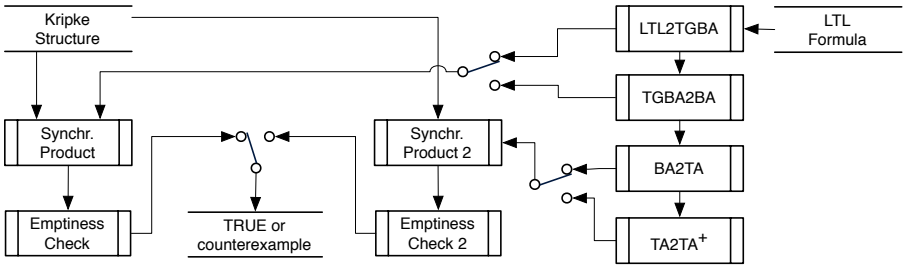


Figure 4.4: The experiment's architecture. The transformation of TA into STA is represented by the block "TA2TA$^+$". Two command-line switches control which one of the different approaches is used to verify an LTL formula on a Kripke structure.

4.4.2 Results

Table 4.1 and Table 4.2 shows how the TGBA, TA and STA approaches deal with the toy models and random formulas already presented in Section 3.6.2. We omit data for BA since they are outperformed by TGBA (see Section 3.6.3). Table 4.5 and Table 4.6 show toy models against weak-fairness formulas.

Table 4.3 and Table 4.4 show the results of the two cases studies against random, weak-fairness, and dedicated formulas issued from the studies.

All values shown in tables are averaged over 100 different formulas: we checked each model against 100 random and 100 weak-fairness formulas that had no counterexample, and against 100 random and 100 weak-fairness formulas that had a counterexample. The average and maximum are computed separately on each model against each set of formulas.

The cases with formulas verified are separated from cases with violated ones. For verified formulas (Table 4.1, 4.5 and 4.3), no counterexample is found and the full state-space has to be explored. For violated formulas (Table 4.2, 4.6 and 4.4) the on-the-fly exploration of the state-space stopped as soon as the existence of a counterexample could be computed.

Column-wise, these tables show the average and maximum sizes (states and transitions) of: (1) the automata $A_{\neg\varphi_i}$ expressing the properties φ_i; (2) the products $A_{\neg\varphi_i} \otimes \mathcal{K}$ of the property with the model; and (3) the subset of this product that was actually explored by the emptiness check. For verified properties, the emptiness check of TGBA and STA always explores the full product, so the sizes shown in the columns of the product and the emptiness check are equal, while the emptiness check of TA must in many cases perform two passes on the full product (see the linear cloud of green crosses below the diagonal of Figure 4.5a), so the columns of the product and the emptiness check in TA approach show different values. On violated properties, the emptiness check aborts as soon as it finds a counterexample, so the explored size is usually significantly smaller than the full product.

The emptiness check values show a third column labeled "T": this is the time in $\frac{1}{100^e}$ of seconds (a.k.a. centiseconds) spent doing that emptiness check, including the on-the-fly computation of the subset of the product that is explored. The time spent constructing the property automata from the formulas, shown in column "T_φ" (in centiseconds), is negligible compared to that of the emptiness check.

			Verified properties (no counterexample)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T	
Peterson5	TGBA	avg	12	460	1	2753024	14166594	2753024	14166594	18656	
		max	54	2816	6	11010896	108553439	11010896	108553439	80455	
	TA	avg	83	4324	12	**2600539**	**10293714**	4498653	17832896	30334	
		max	509	37096	240	9384663	42614845	18769326	85229690	122867	
	STA	avg	83	5131	13	2620919	10388576	**2620919**	**10388576**	**17784**	
		max	509	48968	292	9867135	45471783	9867135	45471783	65036	
Ring6	TGBA	avg	12	411	0	1398021	10139043	1398021	10139043	2325	
		max	49	3788	4	4269384	50547840	4269384	50547840	10181	
	TA	avg	69	3118	9	**1097667**	**6606130**	2155839	12981954	3744	
		max	509	25773	150	2851152	19625864	5702304	39251728	11149	
	STA	avg	70	3768	10	1229116	7453297	**1229116**	**7453297**	**2238**	
		max	509	33578	170	2950460	20815264	2950460	20815264	5635	
FMS5	TGBA	avg	9	245	0	2038054	15037218	2038054	15037218	3017	
		max	28	2612	4	9132417	89397363	9132417	89397363	15594	
	TA	avg	48	1643	5	**1401286**	**11000449**	1496882	11732337	3084	
		max	282	11007	98	6109887	54324903	6368802	54324903	14894	
	STA	avg	49	1962	6	1413836	11090638	**1413836**	**11090638**	**2889**	
		max	283	13230	118	6109887	54324903	6109887	54324903	12928	
Kanban5	TGBA	avg	8	166	0	3356053	32800737	3356053	32800737	5253	
		max	60	1994	4	20253072	258315134	20253072	258315134	44281	
	TA	avg	44	1659	3	**2433853**	**23389805**	**2433853**	**23389805**	4871	
		max	277	21251	80	15272712	161364553	15272712	161364553	33129	
	STA	avg	44	1873	4	**2433853**	**23389805**	**2433853**	**23389805**	**4824**	
		max	277	22324	87	15272712	161364553	15272712	161364553	31731	
Philo10	TGBA	avg	14	636	1	4668178	27564474	4668178	27564474	9352	
		max	81	5397	6	17947837	119545256	17947837	119545256	40273	
	TA	avg	71	4092	20	**2334154**	**19893200**	**2334154**	**19893200**	7742	
		max	412	55321	235	8378151	74240975	8378151	74240975	27233	
	STA	avg	72	4904	22	**2334154**	**19893200**	**2334154**	**19893200**	**7593**	
		max	413	67836	260	8378151	74240975	8378151	74240975	27799	
Robin15	TGBA	avg	15	664	0	2682881	30981263	2682881	30981263	9192	
		max	82	5792	4	13129644	225905117	13129644	225905117	62328	
	TA	avg	95	6048	15	**2027251**	**17168855**	3928111	33295175	12207	
		max	481	43547	149	8169472	68645888	16338944	137291776	48584	
	STA	avg	96	7158	17	2303685	19603937	**2303685**	**19603937**	**7257**	
		max	482	47171	175	8755200	75356160	8755200	75356160	27955	

Table 4.1: Comparison of the three approaches on toy examples with **random formulæ, when counterexamples do not exist.**

			Violated properties (a counterexample exists)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T	
Peterson5	TGBA	avg	14	646	0	8 118 953	42 865 329	600 828	2 325 266	3 892	
		max	47	3 392	5	50 297 469	364 396 821	3 632 979	22 095 045	24 665	
	TA	avg	90	5 683	15	**8 592 222**	**33 972 863**	**546 562**	**1 886 029**	**3 683**	
		max	656	62 947	134	56 166 663	224 108 070	3 178 506	11 478 151	22 221	
	STA	avg	91	6 824	17	9 035 627	35 735 598	590 613	2 053 508	3 955	
		max	657	69 867	146	56 483 288	225 386 395	3 644 229	13 374 186	25 569	
Ring6	TGBA	avg	17	713	1	3 163 763	26 260 802	854 229	5 805 080	1 329	
		max	51	3 788	4	9 855 589	108 310 156	5 914 415	65 020 741	13 999	
	TA	avg	125	7 462	15	**2 846 468**	**18 131 253**	**646 844**	**3 858 721**	**1 198**	
		max	472	41 068	162	8 960 960	62 840 996	4 923 301	36 659 892	10 659	
	STA	avg	126	8 960	17	3 160 137	20 284 936	728 890	4 365 527	1 291	
		max	473	52 838	180	9 204 480	67 482 964	5 041 865	38 542 138	10 459	
FMS5	TGBA	avg	14	496	1	10 174 866	95 061 682	1 602 719	10 636 471	2 148	
		max	96	4 200	3	40 672 843	404 072 123	11 075 871	110 774 947	18 644	
	TA	avg	90	4 112	9	**8 418 689**	**70 784 299**	1 044 896	7 693 244	1 968	
		max	390	22 702	78	31 864 749	267 656 316	8 518 894	63 178 516	16 437	
	STA	avg	91	4 855	11	9 021 110	75 667 584	**1 033 999**	**7 665 018**	**1 979**	
		max	391	26 803	84	34 082 475	286 582 247	7 336 271	61 123 648	15 463	
Kanban5	TGBA	avg	13	355	0	10 952 627	119 272 618	1 153 850	8 728 297	1 473	
		max	87	2 184	3	43 038 083	587 543 197	13 023 468	185 410 581	26 750	
	TA	avg	75	2 870	6	**8 429 948**	**83 120 910**	**670 172**	**5 871 206**	**1 260**	
		max	473	30 375	94	31 602 068	331 833 407	7 673 335	77 363 942	15 562	
	STA	avg	76	3 353	7	9 248 939	90 946 642	670 376	5 873 036	1 241	
		max	474	38 397	99	34 070 884	364 831 218	7 673 335	77 363 941	14 607	
Philo10	TGBA	avg	13	590	0	19 118 012	195 258 228	1 518 296	7 522 623	2 651	
		max	74	5 928	3	84 068 722	1 377 479 362	9 032 250	53 881 112	15 880	
	TA	avg	92	5 690	14	**16 012 398**	**148 019 289**	900 960	6 499 587	2 558	
		max	356	46 498	164	66 535 322	696 331 784	7 342 016	64 470 840	23 509	
	STA	avg	93	6 676	16	17 277 606	159 696 150	**870 692**	**6 437 341**	**2 528**	
		max	357	55 042	187	70 986 407	759 461 806	7 340 801	64 469 273	23 022	
Robin15	TGBA	avg	21	965	1	5 514 573	67 871 153	1 676 314	18 343 280	5 475	
		max	74	5 928	7	15 127 431	281 880 971	13 627 374	191 356 248	52 962	
	TA	avg	164	11 366	21	**4 869 334**	**41 343 229**	**1 248 873**	**10 418 682**	**4 110**	
		max	854	62 419	115	11 791 872	104 488 704	7 767 216	74 241 202	27 244	
	STA	avg	165	13 727	23	5 452 751	46 556 753	1 497 531	12 536 773	4 671	
		max	855	80 513	134	12 072 448	108 812 032	8 860 848	85 541 043	30 942	

Table 4.2: Comparison of the three approaches on toy examples with **random formulæ, when counterexamples exist.**

				Verified properties (no counterexample)								
				Automaton			Full product		Emptiness check			
				st.	tr.	T_φ	st.	tr.	st.	tr.	T	
PolyORB 3/3/2	RND	TGBA	avg	16	715	1	109 158	328 976	109 158	328 976	601	
			max	136	5 792	4	373 039	2 138 028	373 039	2 138 028	2 207	
		TA	avg	94	5 768	17	**94 283**	**205 464**	158 401	346 203	867	
			max	481	43 547	183	261 318	604 061	522 636	1 208 122	2 753	
		STA	avg	95	6 854	20	103 955	227 203	**103 955**	**227 203**	**570**	
			max	482	47 171	248	279 251	687 246	279 251	687 246	1 534	
	WFair	TGBA	avg	5	72	0	96 435	277 426	**96 435**	277 426	**524**	
			max	18	337	2	368 898	1 450 005	368 898	1 450 005	2 049	
		TA	avg	79	1 106	1	**120 479**	**262 456**	121 096	263 776	652	
			max	351	5 920	8	633 208	1 408 943	633 208	1 408 943	3 359	
		STA	avg	80	1 145	1	120 483	262 464	120 483	**262 464**	651	
			max	352	6 072	8	633 208	1 408 943	633 208	1 408 943	3 373	
	Φ_1	TGBA	–	7	576	1	345 241	760 491	**345 241**	**760 491**	**1 688**	
		TA	–	80	14 590	8	**342 613**	**742 815**	685 226	1 485 630	3 374	
		STA	–	81	17 110	12	348 499	760 788	348 499	760 788	1 716	
MAPK 8	RND	TGBA	avg	10	333	1	2 808 973	25 212 509	2 808 973	25 212 509	5 325	
			max	64	3 813	5	13 636 352	147 555 158	13 636 352	147 555 158	28 979	
		TA	avg	50	2 239	9	**1 724 291**	**20 203 618**	**1 724 291**	**20 203 618**	4 852	
			max	336	22 614	149	8 469 258	108 847 708	8 469 258	108 847 708	25 029	
		STA	avg	51	2 702	10	**1 724 291**	**20 203 618**	**1 724 291**	**20 203 618**	**4 813**	
			max	337	26 522	170	8 469 258	108 847 708	8 469 258	108 847 708	25 445	
	WFair	TGBA	avg	5	39	0	3 898 645	34 822 961	3 898 645	34 822 961	7 344	
			max	21	198	2	14 452 198	162 156 912	14 452 198	162 156 912	31 117	
		TA	avg	44	407	1	**2 112 810**	**24 492 299**	**2 112 810**	**24 492 299**	6 126	
			max	171	1 920	3	6 110 748	75 624 744	6 110 748	75 624 744	20 653	
		STA	avg	44	422	1	**2 112 810**	**24 492 299**	**2 112 810**	**24 492 299**	**6 067**	
			max	172	2 012	3	6 110 748	75 624 744	6 110 748	75 624 744	18 566	
	Φ_2	TGBA	–	6	165	1	46 494	302 350	46 494	302 350	50	
		TA	–	9	293	2	**33 376**	**289 235**	**33 376**	**289 235**	**50**	
		STA	–	10	415	1	**33 376**	**289 235**	**33 376**	**289 235**	51	

Table 4.3: Comparison of the three approaches for the **case studies when counterexamples do not exist.**

				Violated properties (a counterexample exists)								
				Automaton			Full product		Emptiness check			
				st.	tr.	T_φ	st.	tr.	st.	tr.	T	
PolyORB 3/3/2	RND	TGBA	avg	14	760	1	132 232	380 305	**53 901**	128 960	**291**	
			max	65	7 004	9	1 295 661	5 045 814	192 692	514 905	1 049	
		TA	avg	81	6 223	29	**135 237**	**296 081**	56 518	**126 300**	311	
			max	540	59 971	574	1 377 784	2 976 709	243 950	547 459	1 284	
		STA	avg	82	7 771	33	148 699	325 993	58 989	131 829	324	
			max	541	72 599	759	1 438 600	3 109 093	253 761	570 216	1 373	
	WFair	TGBA	avg	5	66	0	**96 849**	**258 704**	**69 053**	**177 391**	**372**	
			max	22	516	2	432 240	1 584 372	193 264	1 031 898	1 110	
		TA	avg	82	1 157	1	201 879	434 747	113 705	250 573	615	
			max	262	4 636	5	560 685	1 194 198	254 328	594 751	1 358	
		STA	avg	83	1 197	1	202 820	436 769	114 335	251 969	617	
			max	263	4 674	5	560 685	1 194 198	254 328	594 751	1 392	
MAPK 8	RND	TGBA	avg	13	513	1	27 266 223	405 363 157	1 850 695	12 575 125	2 702	
			max	59	4 298	2	91 059 214	1 963 331 216	13 284 206	111 916 150	21 771	
		TA	avg	88	4 818	13	**22 458 466**	**298 950 743**	**965 567**	**9 245 311**	**2 392**	
			max	334	35 401	165	90 049 281	1 300 904 178	5 241 327	61 655 512	14 227	
		STA	avg	89	5 658	15	23 784 395	315 527 818	967 140	9 251 558	2 389	
			max	335	42 297	184	91 186 725	1 328 333 534	5 241 327	61 655 512	14 256	
	WFair	TGBA	avg	6	81	0	20 904 101	300 416 331	1 962 765	13 896 236	3 129	
			max	23	414	2	101 813 441	1 681 905 871	9 463 562	72 266 873	16 928	
		TA	avg	88	1 212	1	**27 535 941**	**358 622 897**	1 112 324	11 462 361	**3 002**	
			max	245	3 749	5	93 441 682	1 258 076 980	5 203 746	59 642 346	16 992	
		STA	avg	89	1 252	1	28 387 255	372 261 767	**1 110 154**	**11 456 800**	**3 007**	
			max	246	3 796	4	94 454 379	1 279 166 220	5 203 741	59 642 337	15 180	

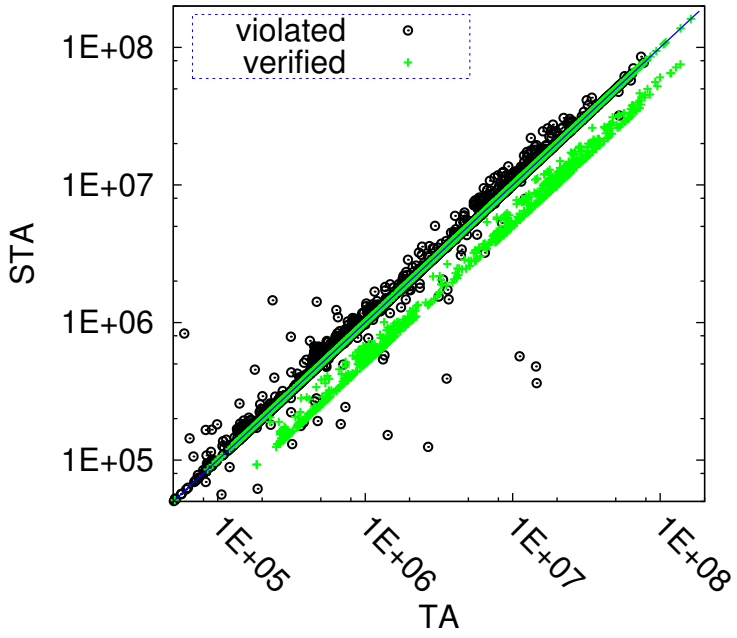
Table 4.4: Comparison of the three approaches for the **case studies when counterexamples exist.**

			Verified properties (no counterexample)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T	
Peterson5	TGBA	avg	4	52	0	3071854	15745483	3071854	15745483	21163	
		max	17	357	2	9551018	87453872	9551018	87453872	70107	
	TA	avg	50	564	1	**3018533**	**11771410**	3436223	13401691	23597	
		max	184	3362	4	10750205	42228050	11867218	47116180	78154	
	STA	avg	51	589	1	3018776	11772694	**3018776**	**11772694**	**20870**	
		max	185	3456	4	10750205	42228050	10750205	42228050	77007	
Ring6	TGBA	avg	5	64	0	1121590	7920270	**1121590**	7920270	**1903**	
		max	25	357	2	3873576	38047692	3873576	38047692	7910	
	TA	avg	58	682	1	**1118511**	**6494556**	1774991	10328239	3046	
		max	183	3353	3	3383372	22481284	6766744	44962568	11037	
	STA	avg	59	712	1	1126059	6547031	1126059	**6547031**	2024	
		max	184	3447	4	3386972	22595612	3386972	22595612	6062	
FMS5	TGBA	avg	4	41	0	3030750	23589355	3030750	23589355	4674	
		max	12	212	2	11882973	114787553	11882973	114787553	23008	
	TA	avg	48	515	1	**2259230**	**17600147**	**2259230**	**17600147**	4589	
		max	171	4256	3	9764223	79707273	9764223	79707273	18521	
	STA	avg	49	534	1	**2259230**	**17600147**	**2259230**	**17600147**	**4561**	
		max	172	4367	4	9764223	79707273	9764223	79707273	18365	
Kanban5	TGBA	avg	4	33	1	3098615	29435137	3098615	29435137	4712	
		max	21	168	2	13014212	187004177	13014212	187004177	26143	
	TA	avg	36	306	1	**2083777**	**19156589**	**2083777**	**19156589**	4112	
		max	140	2049	3	11039112	106534190	11039112	106534190	23453	
	STA	avg	36	315	1	**2083777**	**19156589**	**2083777**	**19156589**	**4059**	
		max	141	2120	3	11039112	106534190	11039112	106534190	22420	
Philo10	TGBA	avg	4	35	0	4255979	26793678	4255979	26793678	9064	
		max	21	216	2	12974557	112561242	12974557	112561242	39806	
	TA	avg	47	501	1	**3229690**	**27631036**	**3229690**	**27631036**	**10439**	
		max	289	8199	6	10987384	94141317	10987384	94141317	38378	
	STA	avg	48	524	1	**3229690**	**27631036**	**3229690**	**27631036**	10576	
		max	290	8346	6	10987384	94141317	10987384	94141317	34352	
Robin15	TGBA	avg	5	59	0	1782133	17996764	1782133	17996764	5749	
		max	16	337	1	5029881	70344648	5029881	70344648	19263	
	TA	avg	54	618	1	**1508082**	**12171559**	1802576	14591484	5731	
		max	239	3706	4	3944448	32391168	7098368	61216768	22074	
	STA	avg	54	642	1	1520057	12265094	**1520057**	**12265094**	**4897**	
		max	240	3763	4	3944448	32391168	3944448	32391168	12200	

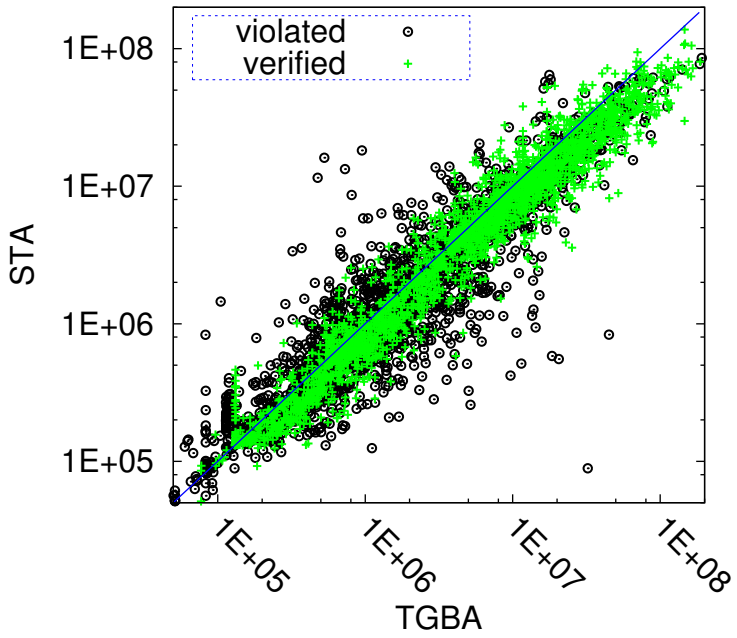
Table 4.5: Comparison of the three approaches on toy examples with **weak-fairness formulæ, when counterexamples do not exist.**

			Violated properties (a counterexample exists)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	TGBA	avg	5	78	0	**3 731 947**	**17 751 951**	**522 414**	**1 759 041**	**3 438**
		max	18	351	2	13 508 730	79 589 415	3 696 481	16 733 420	24 769
	TA	avg	95	1 400	2	10 950 499	42 790 086	641 997	2 123 401	4 370
		max	350	5 968	13	30 354 888	120 985 972	7 637 347	29 621 420	51 473
	STA	avg	95	1 448	1	10 951 464	42 793 386	642 215	2 124 081	4 362
		max	351	6 120	9	30 354 888	120 985 972	7 637 348	29 621 421	51 487
Ring6	TGBA	avg	7	123	1	**2 000 787**	**15 868 857**	**890 567**	5 736 568	**1 410**
		max	38	1 168	3	4 670 124	53 246 768	2 766 283	26 917 302	6 101
	TA	avg	128	2 009	2	3 456 785	21 069 783	905 265	**5 156 327**	1 621
		max	388	12 783	14	6 152 232	43 283 696	2 747 578	17 464 827	4 793
	STA	avg	129	2 063	2	3 459 344	21 101 063	907 368	5 174 029	1 614
		max	389	12 873	14	6 160 248	43 438 480	2 747 578	17 464 827	5 352
FMS5	TGBA	avg	5	68	0	**7 978 129**	**75 701 408**	1 372 877	7 544 397	1 607
		max	23	370	2	27 570 312	293 219 919	6 676 584	53 451 994	9 961
	TA	avg	71	871	1	10 007 988	81 736 556	848 557	5 707 348	1 522
		max	245	3 766	5	27 733 464	249 573 778	6 425 987	51 962 113	12 307
	STA	avg	71	900	1	10 128 248	82 964 077	**841 164**	**5 667 421**	**1 511**
		max	246	3 766	4	27 733 464	249 573 778	6 425 987	51 962 113	11 806
Kanban5	TGBA	avg	5	66	0	**7 040 236**	**73 867 374**	1 344 870	9 279 129	1 562
		max	17	305	2	25 423 161	388 961 954	7 950 633	80 041 971	11 336
	TA	avg	69	849	1	8 102 548	77 515 606	809 467	6 987 095	1 474
		max	237	3 655	3	32 732 392	329 125 090	5 501 242	53 375 472	10 218
	STA	avg	70	875	1	8 355 726	80 541 627	**788 150**	**6 773 391**	**1 442**
		max	238	3 710	3	33 147 912	335 772 570	5 498 955	53 371 193	10 147
Philo10	TGBA	avg	5	69	0	**10 515 366**	**102 622 810**	833 663	**3 762 217**	**1 357**
		max	25	392	3	24 496 745	402 197 001	5 017 174	32 273 634	9 802
	TA	avg	81	1 027	1	16 779 360	150 030 607	742 819	5 019 286	1 943
		max	267	4 339	5	43 186 049	394 172 486	6 703 064	55 546 036	20 284
	STA	avg	81	1 068	1	16 801 072	150 313 327	**727 226**	4 926 991	1 920
		max	268	4 386	6	43 194 029	394 299 774	6 703 064	55 546 036	20 201
Robin15	TGBA	avg	9	157	1	**4 006 015**	**46 910 526**	1 486 839	14 360 747	4 691
		max	44	1 168	2	14 802 883	195 152 925	5 666 009	78 112 291	21 619
	TA	avg	143	2 467	2	5 832 674	48 201 707	**1 348 406**	**10 788 993**	**4 327**
		max	466	12 783	14	18 204 684	151 155 748	6 486 142	53 115 008	18 844
	STA	avg	144	2 529	2	5 841 364	48 285 515	1 354 923	10 843 123	4 324
		max	467	12 873	13	18 204 684	151 155 748	6 486 142	53 115 008	19 093

Table 4.6: Comparison of the three approaches on toy examples with **weak-fairness formulæ, when counterexamples exist.**



(a) STA against TA approaches



(b) STA against TGBA approaches

Figure 4.5: Performance (number of transitions explored by the emptiness check) of STA against TA and TGBA.

Figure 4.5 compares the number of visited transitions when running the emptiness check; plotting STA against TA and TGBA. This gives an idea of their relative performance. Each point corresponds to one of the 5600 evaluated formulas (2800 violated with counterexample as black circles, and 2800 verified having no counterexample as green crosses). Each point below the diagonal is in favor of STA while others are in favor of the other approach. Axes are displayed using a logarithmic scale.

No comparison is presented with BA since they are less efficient than TGBA (see the experimental evaluation presented in section 3.6.3).

All these experiments were run on a 64bit Linux system running on an Intel(R) 64-bit Xeon(R) @2.00GHz, with 10GB of RAM.

4.4.3 Discussion

Before discussing the performance of different approaches, we recall that in our implementation using CheckPN tool, the cost of computing labels in the Kripke structure is higher (despite the fact that we use a cache). This increases the computing time of products in TA and STA approaches (more than in the approaches TGBA and BA), because TA and STA approaches query two labels by transition of the Kripke structure (to compute an xor between source label and destination label) while other approaches query only one label.

In an implementation where computing labels is cheap, the execution time should be proportional to the number of transitions explored by the emptiness check, so it is important to not consider only the execution time provided by our experiments.

On verified properties, the results are very straightforward to interpret when looking at the number of states and transitions explored by the emptiness check.

The emptiness check optimizations proposed in Section 4.2 improve the performance of the TA approach. This can be observed by comparing the results in this chapter and the results of TA approach in the previous chapter (see Section 3.6.3: for verified properties, the number of transitions visited by the emptiness check was always twice the number of transitions in the product).

TA outperform TGBA except for both Random and weak-fairness properties against Peterson, Ring, Robin and PolyORB.

STA significantly improve TA in all cases where a second pass was necessary. In these cases, the STA approach, with its single-pass emptiness check, is a clear improvement over TA. These cases where the STA approach is twice faster than TA's, appear as a linear cloud of green crosses below the diagonal in the scatter plot of Figure 4.5a (we recall that the axes are displayed using a logarithmic scale) Otherwise, they have the same performance because if no livelock-acceptance states are detected in the product then the TA and STA approaches explore exactly the same product (these cases correspond to the green crosses on the diagonal).

In the scatter plot comparing STA against TGBA, the green crosses appear on both side of the diagonal, with much more points where STA is better. Furthermore, in the results tables, if we observe in more details the average number of states/transitions explored during the emptiness check, STA outperform TGBA in all cases except for weak-fairness formulas against Ring and PolyORB. In this cases, TGBA benefits from the large number of acceptance conditions generated when translating weak-fairness formulas.

On violated properties, it is harder to interpret the results because they depend on the order in which non-deterministic transitions of the property automaton are explored. In the best case, the order of transitions leads the emptiness check straight to a counterexample; in the worst case, the algorithm explores the whole product until it finally finds a counterexample.

The different kinds of property automata TGBA, TA and STA provide different orders of transitions and therefore change the number of states and transitions to be explored by the emptiness check before a counterexample is found.

If we analyze in more details the results tables, we observe that:

- For the (full) product size, TA and STA produce a smaller product on the average than TGBA for random formulas. However, for weak-fairness formulas, TGBA produces the smallest product on the average.
- For the emptiness check, looking at the average number of transitions explored by the emptiness check (and taking into account the extra computation of labels discussed previously), indicates that TA and STA approaches outperform the TGBA approach except for weak-fairness formulas against Peterson, Philo and PolyORB. Even for STA where an artificial state and non-deterministic transitions are added, no significant overhead is noticed.

4.5 Conclusion

In a preliminary work presented in the previous chapter, we experiment LTL model checking of stuttering-insensitive properties with various techniques: Büchi automata (BA), Transition-based Generalized Büchi Automata and Testing Automata [46]. At this time, conclusions were that TA has good performance for violated properties (i.e. when a counterexample was found). However, this was not the case when no counterexample was computed since the entire product had to be visited twice to check for each acceptance mode of a TA (Büchi acceptance or livelock-acceptance).

This chapter extends the above work in two ways. First, it introduces a modified emptiness check algorithm that avoids the second pass when it is useless (i.e., no livelock-acceptance state is detected in the product). Second, it proposes a transformation of TA into STA that avoids the need for a second pass (in all cases).

Both new algorithms have been implemented in Spot, our model checking library and used on several benchmark models including large models issued from case studies. Experimentation with Spot reported that, STA remain good for violated properties, and also beat TA and TGBA in most cases when properties exhibit no counterexample.

In the next chapter, we introduce a new kind of automata that combines ideas from TGBA and STA. The goal is to have a form of single-pass testing automata without adding any artificial state and with generalized acceptance conditions on transitions like a TGBA. These generalized acceptance conditions allow to improve the performance when checking weak-fairness formulas, as observed in the experimentation with better performance of TGBA for this kind of formulas.

CHAPTER 5

Transition-based Generalized Testing Automata (TGTA): A Single-pass and Generalized New Automata

Contents

5.1 Introduction

In Chapter 3, we have shown that Testing Automata (TA) are better than Büchi Automata when the formula to be verified is violated (i.e., a counterexample is found), but this is not the case when the property is verified since the entire product have to be visited twice to check for each acceptance mode of a TA. Then, in order to improve the TA approach, we proposed STA in Chapter 4. STA is a transformation of TA into a normal form that does not need the second pass of the emptiness

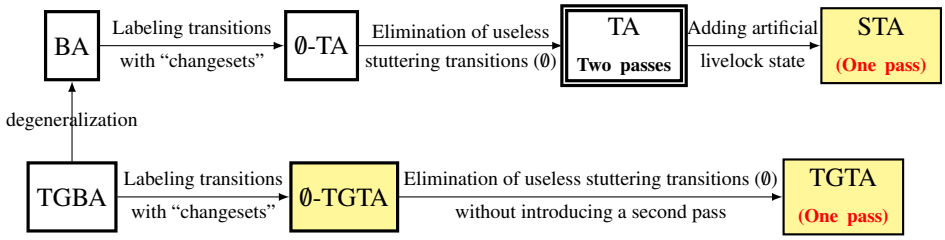


Figure 5.1: An overview of the relations between the different variants of ω-automata presented in this thesis.

check. Unfortunately, STA can increase the size of the product automaton, because in order to remove the second pass, the transformation from TA into STA adds an artificial state.

This chapter introduces a new type of ω-automata for stutter-invariant properties, called Transition-based Generalized Testing Automata (TGTA) [6], that mixes features from both TA and TGBA.

The basic idea of TGTA is to build an improved form of testing automata with generalized acceptance conditions on transitions, which allows us to modify the automata construction in order to remove the second pass of the emptiness check of the product. The constructed TGTA represents all the stuttering-transitions using only self-loops.

TGTA combines the advantages of TA and TGBA and it is better than STA because TGTA allows to remove the second pass without adding an artificial state.

In addition, as seen in Figure 5.1, a TA is built from a BA while a TGTA is built directly from a TGBA. Therefore, compared to TA, TGTA can benefit from the fact that TGBA are smaller than BA. Indeed, TGBA are more concise [49, 36] because they use generalized acceptance conditions on transitions, especially for weak-fairness formulas, as was already observed in Section 3.6.3 page 54.

Another advantage of TGTA compared to TA, is that the implementation of TGTA approach does not require a dedicated emptiness check, it reuses the same algorithm used for TGBA (and BA), and the counterexample constructed by this algorithm is also reported as a counterexample for the TGTA approach. We are thus able to compare TGTA with the "traditional" algorithms we used on TA, BA and TGBA. The results of these experimental comparisons show that TGTA compete well on our examples: the TGTA approach is statistically more efficient than the other evaluated approaches, especially when no counterexample is found (i.e., the property is verified) because it does not require a second pass.

5.2 Transition-based Generalized Testing Automata (TGTA)

The following definition of TGTA combines features from both TGBA and TA. From TGBA, we inherit the use of transition-based generalized acceptance conditions. From TA, we take the idea of labeling transitions with changesets, however we remove the use of livelock-acceptance (because it may require a two-pass emptiness check), and we remove the implicit stuttering (in TGTA, δ explicitly represents the stuttering transitions, but we will see in the following that our constructed TGTA satisfies a stuttering-normalization constraint that optimizes the representation

of these stuttering transitions).

The resulting Chimera, accepts only stuttering-insensitive languages like TA, and inherits advantages from both TA and TGBA: it has a simple one-pass emptiness-check procedure (the same as the one for TGBA), and can benefit from reductions based on the stuttering of the properties pretty much like a TA. Livelock acceptance states, which are no longer supported, can be emulated using states with stuttering self-loops labeled by the set of all acceptance conditions $\mathcal{F}$ (these particular self-loops are called "accepting stuttering self-loops" in the following).

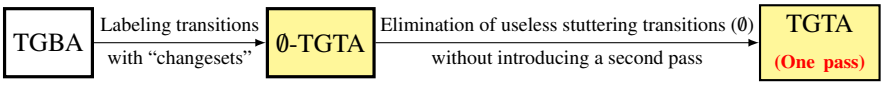


Figure 5.2: The two steps of the construction of a TGTA from a TGBA.

5.2.1 ∅-TGTA

Before defining TGTA, we begin by defining an intermediate form called ∅-TGTA ("empty-changesets" TGTA). This intermediate form is a generalized Testing Automaton that allows to represent any LTL property. Then, we define TGTA as a normal form of ∅-TGTA used to only represent stutter-invariant properties (LTL$\setminus$X). In Section 5.3 we will show that a TGTA is constructed in two steps as illustrated in Figure 5.2. The first step transforms a TGBA into an ∅-TGTA by labeling transitions with changesets. The second step transforms an ∅-TGTA into a TGTA by normalizing the representation of stuttering transitions. This normalization benefits from the hypothesis that the LTL property is stutter-invariant to remove all stuttering transitions that are not self-loops in TGTA.

In addition to being used in the construction of TGTA, ∅-TGTA will also be used to represent the product between a TGTA and a Kripke structure in Section 5.4.1.

Definition 29 (∅-TGTA)**.** *An ∅-TGTA over the alphabet Σ is a tuple $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$ where:*

- *Q is a finite set of states,*
- *$I \subseteq Q$ is a set of initial states,*
- *$U : I \to 2^{\Sigma}$ is a function mapping each initial state to a set of symbols of Σ,*
- *$\mathcal{F}$ is a finite set of acceptance conditions,*
- *$\delta \subseteq Q \times \Sigma \times 2^{\mathcal{F}} \times Q$ is the transition relation, where each element (q,k,F,q') represents a transition from state q to state q' labeled by a* changeset *k interpreted as a (possibly empty) set of atomic propositions whose values change between q and q', and the set of acceptance conditions $F \in 2^{\mathcal{F}}$,*

An infinite word $\sigma = \ell_0\ell_1\ell_2\ldots \in \Sigma^{\omega}$ is accepted by $\mathcal{T}$ if there exists an infinite path $r = (q_0, \ell_0 \oplus \ell_1, F_0, q_1)(q_1, \ell_1 \oplus \ell_2, F_1, q_2)(q_2, \ell_2 \oplus \ell_3, F_2, q_3)\ldots \in \delta^{\omega}$ where:

- *$q_0 \in I$ with $\ell_0 \in U(q_0)$ (the infinite word is recognized by the path),*
- *$\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$ (each acceptance condition is visited infinitely often).*

(a) An $\emptyset$-TGTA for $\mathsf{FG}\,p$

(b) A TGTA for $\mathsf{FG}\,p$

Figure 5.3: An $\emptyset$-TGTA (left) and a TGTA (right) for the LTL property $\varphi = \mathsf{FG}\,p$, with acceptance conditions indicated by the black dot $\bullet$.

The language accepted by $\mathcal{T}$ is the set $\mathscr{L}(\mathcal{T}) \subseteq \Sigma^\omega$ of infinite words it accepts.

Figure 5.3a shows an $\emptyset$-TGTA recognizing the LTL formula $\mathsf{FG}\,p$. Acceptance sets are represented using dots as in TGBAs. Transitions are labeled by changesets: e.g., the transition $(0,\{p\},1)$ means that the value of p changes between states 0 and 1. Initial valuations are shown above initial arrows: $U(0) = \{\bar{p}\}$, $U(1) = \{p\}$ and $U(2) = \{p\}$. Any infinite path in this example is accepted if it visits infinitely often, the acceptance transition indicated by the black dot $\bullet$: i.e., the stuttering self-loop $(2,\emptyset,\bullet,2)$. As an illustration, the infinite word $\bar{p};p;p;p;\ldots$ is accepted by the run $⓪\xrightarrow{\{p\}}①\overset{\emptyset}{\bullet\!\!\rightarrow}②\overset{\emptyset}{\bullet\!\!\rightarrow}②\cdots$ Indeed, a run recognizing such an infinite word must start in state 0 (because only $U(0) = \{\bar{p}\}$), then it changes the value of p, so it has to move to state 1 because from state 0 only the transition $(0,\{p\},1)$ is labeled by $\{p\}$. In the next step, the value of p does not change and the run must execute a stuttering transition among $(1,\emptyset,1)$ or $(1,\emptyset,2)$. To be accepted, it must eventually move to state 2 (rather than remain in state 1), and finally stay on state 2 by executing infinitely the accepting stuttering self-loop $(2,\emptyset,\bullet,2)$.

5.2.2 TGTA

A TGTA is a normal form of $\emptyset$-TGTA satisfying a structural constraint on stuttering transitions, called "stuttering-normalization constraint", its objective is to force the TGTA to represent the stuttering transitions using only stuttering self-loops.

Definition 30 (TGTA)**.** *A TGTA is an $\emptyset$-TGTA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}\rangle$ such that $\mathscr{L}(\mathcal{T})$ is stutter-invariant and the transition relation δ has to satisfy the following* stuttering-normalization constraint*:*

1. *All stuttering transitions are self-loops, and*
2. *and every state has a stuttering self-loop.*

Formally, the stuttering-normalization constraint can be expressed by the following equivalence: $\forall (q,q') \in Q^2 : \big(\exists F \in 2^{\mathcal{F}}, (q,\emptyset,F,q') \in \delta\big) \iff (q = q')$

Figure 5.3b shows a TGTA recognizing the LTL formula $\mathsf{FG}\,p$. We note that this TGTA satisfies the stuttering-normalization constraint. Indeed, we have that the set of stuttering transitions is $\{(0,\emptyset,\emptyset,0),(1,\emptyset,\emptyset,1),(2,\emptyset,\bullet,2)\}$, i.e., a stuttering self-loop on each state. In this TGTA, the

infinite word $\bar{p};p;p;p;\ldots$ is accepted by the run $\textcircled{0}\xrightarrow{\{p\}}\textcircled{2}\overset{\emptyset}{\bullet\!\!\rightarrow}\textcircled{2}\overset{\emptyset}{\bullet\!\!\rightarrow}\textcircled{2}\cdots$ because the value p only changes between the first two steps. Indeed, a run recognizing such an infinite word must start in state 0 (because only $U(0)=\{\bar{p}\}$), then it changes the value of p, so it has to take transitions labeled by $\{p\}$, i.e., $(0,\{p\},1)$ or $(0,\{p\},2)$. To be accepted, it must move to state 2 (rather than state 1), and finally stay on state 2 by executing infinitely the accepting stuttering self-loop $(2,\emptyset,\bullet,2)$.

In this work, we define and use TGTA only for stutter-invariant LTL properties. Indeed, we need this restriction to build a TGTA that satisfies the stuttering-normalization constraint (in particular , we will show in section 5.3.2 how we exploited this restriction to remove all the stuttering transitions that are not self-loops).

In the next section, we present in detail the formalization of the different steps used to build a TGTA that satisfies Definition 30.

5.3 TGTA Construction

Let us now describe how to build a TGTA starting from a TGBA of a stutter-invariant LTL property. The construction is inspired by the one presented in Section 3.4 (page 42) that constructs a TA from a BA.

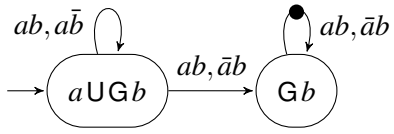
Similar to TA construction, a TGTA is built in two steps as illustrated in Figure 5.2, the first one builds an intermediate $\emptyset$-TGTA from a TGBA. Then, the second step builds the final form of TGTA by removing the useless stuttering transitions of the $\emptyset$-TGTA. This simplification of stuttering transitions does not introduce livelock-accepting states in TGTA (this represents a crucial difference between TA and TGTA because livelock-accepting states require a second pass in the emptiness check of the product using TA).

Figure 5.4d shows a TGTA constructed for $a\,\mathsf{U}\,\mathsf{G}\,b$ in the same way as we did for Figure 3.3d (page 43) . The only accepting runs are those that see $\bullet$ infinitely often. The reader can verify that all the infinite words taken as example in section 3.4 are still accepted, but not always with the same runs (for instance $ab;\bar{a}b;\bar{a}b;\bar{a}b;\ldots$ is accepted by the run $2,4,4,4,\ldots$, but not by the run $2,3,3,3,\ldots$). This difference is due to the way we emulate livelock-accepting states, as we will describe later (in Property 9 page 93).

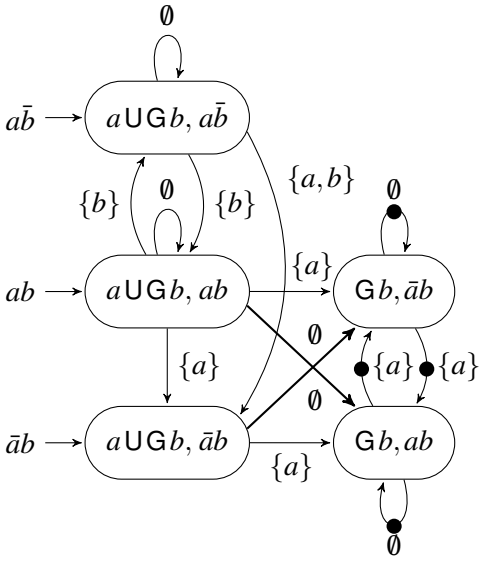
5.3.1 From TGBA to $\emptyset$-TGTA: Construction of an intermediate $\emptyset$-TGTA from a TGBA

This first step is similar to the first step of the TA construction and the following first property is the counterpart of Property 3 (i.e., transforming a BA into an $\emptyset$-TA presented in Section 3.4 page 42). We construct an $\emptyset$-TGTA from a TGBA by moving labels to states, and labeling each transition by the set difference between the labels of its source and destination states. While doing so, we keep the generalized acceptance conditions on the transitions. An example of a constructed $\emptyset$-TGTA is shown on Figure 5.4b.

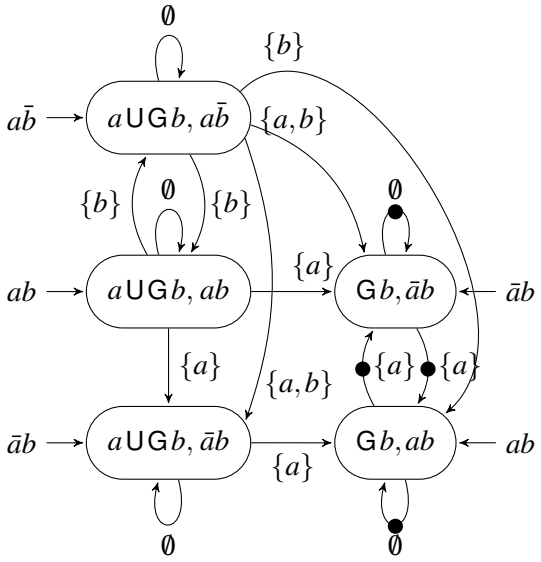
Property 8 (Converting TGBA into $\emptyset$-TGTA)**.** *For any TGBA $\mathcal{G}=\langle Q_{\mathcal{G}},I_{\mathcal{G}},\delta_{\mathcal{G}},\mathcal{F}\rangle$ over the alphabet $\Sigma=2^{AP}$ and such that $\mathscr{L}(\mathcal{G})$ is stutter-invariant, let us define the $\emptyset$-TGTA $\mathcal{T}=$*



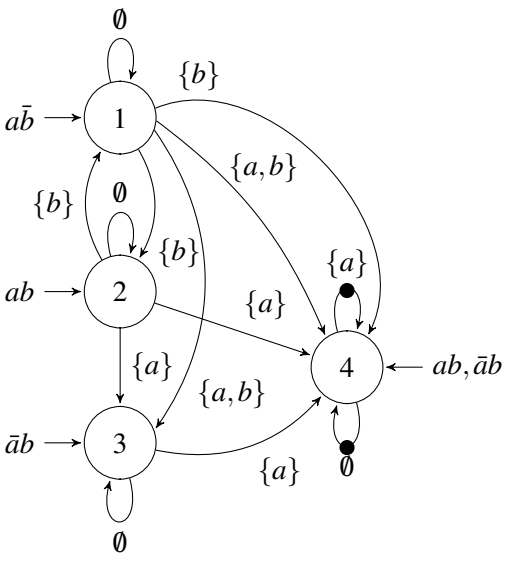
(a) Initial TGBA for $a\mathsf{U}\mathsf{G}b$.



(b) $\emptyset$-TGTA obtained by property 8.



(c) TGTA after simplifications by property 9.



(d) TGTA after bisimulation.

Figure 5.4: TGTA obtained after various steps while translating the TGBA representing $a\mathsf{U}\mathsf{G}b$, into a TGTA with $\mathcal{F} = \{\bullet\}$.

$\langle Q_\mathcal{T}, I_\mathcal{T}, U_\mathcal{T}, \delta_\mathcal{T}, \mathcal{F}\rangle$ *with* $Q_\mathcal{T} = Q_\mathcal{G} \times \Sigma$, $I_\mathcal{T} = I_\mathcal{G} \times \Sigma$ *and*

(i) $\forall (q,\ell) \in I_\mathcal{T}, U_\mathcal{T}((q,\ell)) = \{\ell\}$

(ii) $\forall (q,\ell) \in Q_\mathcal{T}, \forall (q',\ell') \in Q_\mathcal{T}, \big((q,\ell), \ell \oplus \ell', F, (q',\ell')\big) \in \delta_\mathcal{T} \iff ((q,\ell,F,q') \in \delta_\mathcal{G})$

Then $\mathscr{L}(\mathcal{G}) = \mathscr{L}(\mathcal{T})$.

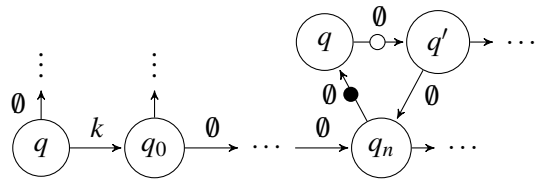
Proof of property 8.
$(\subseteq)$ Let $\sigma_1 = \ell_0\ell_1\ell_2\ldots \in \mathscr{L}(\mathcal{G})$ be an infinite word accepted by $\mathcal{G}$.
By Definition 16, σ_1 is recognized by a path $(q_0,\ell_0,F_0,q_2)(q_2,\ell_1,F_1,q_2)\ldots \in \delta_\mathcal{G}^\omega$ of $\mathcal{G}$, such that $q_0 \in I$, and $\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$. By applying (ii) and (i), we can see that there exists a corresponding path $((q_0,\ell_0),\ell_0 \oplus \ell_1, F_0, (q_1,\ell_1))((q_1,\ell_1),\ell_1 \oplus \ell_2, F_1, (q_2,\ell_2))\ldots \in \delta_\mathcal{T}^\omega$ of $\mathcal{T}$ such that $(q_0,\ell_0) \in I_\mathcal{T}$, $\ell_0 \in U_\mathcal{T}((q_0,\ell_0))$, and still $\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$. By Definition 30 we therefore have $\sigma_1 \in \mathscr{L}(\mathcal{T})$.
$(\supseteq)$ Let $\sigma_2 = w_0w_1w_2\ldots \in \mathscr{L}(\mathcal{T})$ be an infinite word accepted by $\mathcal{T}$.
By Definition 30, σ_2 is recognized by a path $((q_0,\ell_0), w_0 \oplus w_1, F_0, (q_1,\ell_1))((q_1,\ell_1), w_1 \oplus w_2, F_1, (q_2,\ell_2))$
$\ldots \in \delta_\mathcal{T}^\omega$ of $\mathcal{T}$ such that $(q_0,\ell_0) \in I_\mathcal{T}$, $w_0 \in U_\mathcal{T}((q_0,\ell_0))$, and $\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$. Of course we have $w_i \oplus w_{i+1} = \ell_i \oplus \ell_{i+1}$ but this does not suffice to imply that $\ell_i = w_i$. However (i) tells us that $w_0 \in U_\mathcal{T}((q_0,\ell_0)) = \{\ell_0\}$ so $w_0 = \ell_0$, and since $w_i \oplus w_{i+1} = \ell_i \oplus \ell_{i+1}$ it follows that $w_i = \ell_i$. By applying (ii) can now find a corresponding path $(q_0,\ell_0,F_0,q_2)(q_2,\ell_1,F_1,q_2)\ldots \in \delta_\mathcal{G}^\omega$ of $\mathcal{G}$, such that $q_0 \in I, \forall i \in \mathbb{N}, (w_i = \ell_i)$, and $\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$. By Definition 16 we therefore have $\sigma_2 \in \mathscr{L}(\mathcal{G})$. □

5.3.2 From ∅-TGTA to TGTA: Elimination of useless stuttering-transitions (∅) without introducing livelock-acceptance

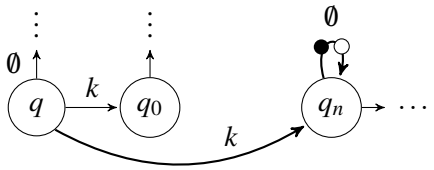
The next property is the pendent of Property 4 (page 45) to simplify the automaton by removing stuttering transitions and thus obtain the final form of the TGTA. Here we cannot remove self-loop transitions labeled by $\emptyset$ (i.e., stuttering self-loops), but we can remove all others. The intuition behind this simplification is illustrated in Figure 5.5a: q_0 is reachable from state q by a non-stuttering transition, but q_0 can reach an accepting stuttering-cycle by following only stuttering transitions. In the context of TA we would have to declare q_0 as being a livelock-accepting state. For TGTA, we replace the accepting stuttering-cycle by adding a self-loop labeled by all acceptance conditions on q_n, then the predecessors of q_0 are connected to q_n as in Figure 5.5b.

In the last step of the following construction, in order to maintain the same accepted (stuttering) language, we add a stuttering self-loop to each state before removing all stuttering transitions between every two distinct states.

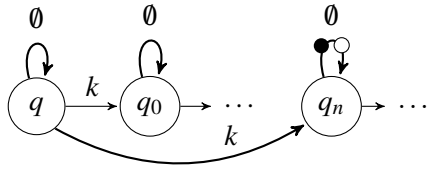
Property 9 (Elimination of useless stuttering transitions of ∅-TGTA to build a TGTA)**.** *Let* $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}\rangle$ *be a* $\emptyset$*-TGTA such that* $\mathscr{L}(\mathcal{T})$ *is stutter-invariant. By combining the first three of the following operations, we can remove all stuttering transitions that are not self-loop (see Figure 5.5) and therefore obtain a TGTA. The fourth operation can be performed along the way for further (classical) simplifications.*



(a) ∅-TGTA: Before reduction of stuttering transitions (∅).



(b) After reduction of stuttering transitions (∅).



(c) TGTA: After removing all stuttering transitions and adding stuttering self-loops on all states.

Figure 5.5: Elimination of useless stuttering transitions of ∅-TGTA to obtain a TGTA.

1. *If $Q \subseteq Q$ is a SCC such that any two states $q, q' \in Q$ can be connected using a sequence of stuttering transitions $(q,\emptyset,F_0,r_1)(r_1,\emptyset,F_1,r_2)\cdots(r_n,\emptyset,F_n,q') \in \delta^*$ with $F_0 \cup F_1 \cup \cdots \cup F_n = \mathcal{F}$, then we can add an accepting stuttering self-loop $(q,\emptyset,\mathcal{F},q)$ on each state $q \in Q$. I.e., the ∅-TGTA $\mathcal{T}' = \langle Q, I, U, \delta \cup \{(q,\emptyset,\mathcal{F},q) \mid q \in Q\}, \mathcal{F} \rangle$ is such that $\mathscr{L}(\mathcal{T}') = \mathscr{L}(\mathcal{T})$. Let us call such a component Q an **accepting Stuttering-SCC**.*

2. *If there exists an accepting Stuttering-SCC Q and a sequence of stuttering-transitions: $(q_0,\emptyset,F_1,q_1)(q_1,\emptyset,F_2,q_2)\cdots(q_{n-1},\emptyset,F_n,q_n) \in \delta^*$ such that $q_n \in Q$ and $q_0, q_1, \ldots q_{n-1} \notin Q$ (as shown in Figure 5.5a), then:*

   - *For any non-stuttering transition, $(q,k,F,q_0) \in \delta$ going to q_0 and such that $k \neq \emptyset$ (and $(q,k,F,q_n) \notin \delta$), the ∅-TGTA $\mathcal{T}'' = \langle Q, I, U, \delta \cup \{(q,k,F,q_n)\}, \mathcal{F} \rangle$ is such that $\mathscr{L}(\mathcal{T}'') = \mathscr{L}(\mathcal{T})$.*
   - *If $q_0 \in I$, the ∅-TGTA $\mathcal{T}'' = \langle Q, I \cup \{q_n\}, U'', \delta, \mathcal{F} \rangle$ with $\forall q \neq q_n, U''(q) = U(q)$ and $U''(q_n) = U(q_n) \cup U(q_0)$, is such that $\mathscr{L}(\mathcal{T}'') = \mathscr{L}(\mathcal{T})$.*

3. *Let $\mathcal{T}^\dagger = \langle Q, I^\dagger, U^\dagger, \delta^\dagger, \mathcal{F} \rangle$ be the ∅-TGTA obtained after repeating the previous two operations as much as possible (i.e., $\mathcal{T}^\dagger$ contains all the transitions and initial states that can be added by the above two operations (Figure 5.5b)). Then, we can add non-accepting stuttering self-loops $(q,\emptyset,\emptyset,q)$ to all states that did not have an accepting stuttering self-loop (Figure 5.5c), because $\mathcal{T}$ describes a stuttering invariant property. Also we can remove all stuttering transitions that are not self-loops since stuttering can be captured by self-loops after the previous two operations. After this last reduction of stuttering transitions, we obtain the final TGTA. More formally, the TGTA $\mathcal{T}''' = \langle Q, I^\dagger, U^\dagger, \delta''', \mathcal{F} \rangle$ with $\delta''' = \{(q,k,F,q') \in \delta^\dagger \mid k \neq \emptyset \vee (q = q' \wedge F = \mathcal{F})\} \cup \{(q,\emptyset,\emptyset,q) \mid (q,\emptyset,\mathcal{F},q) \notin \delta^\dagger\}$ is such that $\mathscr{L}(\mathcal{T}''') = \mathscr{L}(\mathcal{T}^\dagger) = \mathscr{L}(\mathcal{T})$.*

4. *Any state from which one cannot reach a Büchi-accepting cycle can be removed from the automaton without changing its language.*

Here again, an additional optimization is to merge bisimilar states, this can be achieved using the same algorithm used to simplify a TA, taking Q as initial partition and taking into account the acceptance conditions of the outgoing transitions. All these steps are shown in Figure 5.4.

Proof of property 9.

1. $(\mathcal{T}' \supseteq \mathcal{T})$ Obvious because we are only adding transitions. $(\mathcal{T}' \subseteq \mathcal{T})$ Let $\delta' = \delta \cup \{(q,\emptyset,\mathcal{F},q) \mid q \in Q\}$. Consider an accepting infinite word $\sigma = \ell_0\ell_1\ell_2\ldots \in \mathscr{L}(\mathcal{T}')$ recognized by an accepting path π' on $\mathcal{T}'$. Any transition of π' that is not in δ is a self-loop $(q,\emptyset,\mathcal{F},q)$ that has been added to δ' because an accepting stuttering-SCC exists in δ around q: so any $(q,\emptyset,\mathcal{F},q) \in \delta'$ can be replaced by a sequence of stuttering transitions $(q,\emptyset,G_0,r_1)(r_1,\emptyset,G_1,r_2)\ldots(r_n,\emptyset,G_n,q) \in \delta^*$ such that $G_0 \cup G_1 \cup \ldots G_n = \mathcal{F}$. The path $\pi \in \delta^\omega$ obtained by replacing all such transitions is an accepting path of $\mathcal{T}$ that recognizes a word that is stuttering equivalent to σ. Since $\mathscr{L}(\mathcal{T})$ is stuttering-insensitive, it must also contain σ.

2. $(\mathcal{T}'' \supseteq \mathcal{T})$ Obvious for the same reason. $(\mathcal{T}'' \subseteq \mathcal{T})$ We consider the case where q_0 is non initial (the initial case is similar). Let $\delta'' = \delta \cup \{(s,k,F,q_n)\}$. Consider an accepting infinite word $\sigma = \ell_0\ell_1\ell_2\ldots \in \mathscr{L}(\mathcal{T}'')$ recognized by a path π'' on $\mathcal{T}''$. Let π be the path on $\mathcal{T}$ obtained by replacing in π'' any occurrence of $(s,k,F,q_n) \in (\delta'' \setminus \delta)$ by the sequence $(s,k,F,q_0)(q_0,\emptyset,F_1,q_1)(q_1,\emptyset,F_2,q_2)\cdots(q_{n-1},\emptyset,F_n,q_n) \in \delta^*$. The path $\pi \in \delta^\omega$ is also an accepting path of $\mathcal{T}$ that recognizes a word that is stuttering equivalent to σ. Since $\mathscr{L}(\mathcal{T})$ is stuttering-insensitive, it must also contain σ.

3. $\mathscr{L}(\mathcal{T}^\dagger) = \mathscr{L}(\mathcal{T})$ by application of the previous two properties, therefore $\mathscr{L}(\mathcal{T}^\dagger)$ is a stuttering-insensitive language. $\mathscr{L}(\mathcal{T}''')$ is also a stuttering-insensitive language because $\mathcal{T}'''$ is obtained from $\mathcal{T}^\dagger$ that recognizes a stuttering-insensitive language, by adding stuttering self-loops on all its states before removing all stuttering transitions that are not self-loops.

   To prove that two stuttering-insensitive languages are equal, it is sufficient to verify that they contain the same words of the following two forms:

   - $\sigma = \ell_0\ell_1\ell_2\ldots$ with $\forall i \in \mathbb{N}, \ell_i \oplus \ell_{i+1} \neq \emptyset$ (non-stuttering words), or
   - $\sigma = \ell_0\ell_1\ell_2\ldots(\ell_n)^\omega$ with $\forall i < n, \ell_i \oplus \ell_{i+1} \neq \emptyset$ (terminal stuttering words)

   All other accepted words can be generated by duplicating letters in the above words.

   Since we have only touched stuttering transitions, it is clear that the non-stuttering words of $\mathscr{L}(\mathcal{T})$ are the non-stuttering words of $\mathscr{L}(\mathcal{T}''')$.

   We now consider the case of a terminal stuttering word $\sigma = \ell_0\ell_1\ell_2\ldots(\ell_n)^\omega$ with $\forall i < n, \ell_i \oplus \ell_{i+1} \neq \emptyset$.
   $(\mathcal{T}''' \subseteq \mathcal{T}^\dagger)$ The path π''' that recognizes σ in $\mathcal{T}'''$ has the form $(q_0,\ell_0 \oplus \ell_1,F_0,q_1)$ $(q_1,\ell_1 \oplus \ell_2,F_1,q_2)\ldots(q_n,\emptyset,\mathcal{F},q_n)^\omega$ where all transitions are necessarily from $\mathcal{T}^\dagger$ because

we have only added in $\mathcal{T}'''$ transitions of the form $(q,\emptyset,\emptyset,q)$. π''' is thus also an accepting path of $\mathcal{T}^\dagger$ and $\sigma \in \mathscr{L}(\mathcal{T}^\dagger)$.
$(\mathcal{T}''' \supseteq \mathcal{T}^\dagger)$ The path $\pi^\dagger$ that recognizes σ in $\mathcal{T}^\dagger$ does only stutter after ℓ_n. Because this is an accepting path, it has a lasso-shape (i.e., a finite path starting from an initial state with a cycle at the end), where the cyclic part is only stuttering and accepting. Let us denote it $\pi^\dagger = (q_0,\ell_0 \oplus \ell_1,F_0,q_1)(q_1,\ell_1 \oplus \ell_2,F_1,q_2)\dots(q_{n-1},\ell_{n-1} \oplus \ell_n,F_{n-1},q_n)(q_n,\emptyset,F_n,q_{n+1})\dots[(q_m,\emptyset,F_m,q_{m+1})\dots(q_l,\emptyset,F_l,q_m)]^\omega$, with
$\forall i < n, \ell_i \oplus \ell_{i+1} \neq \emptyset$.
Thanks to property 9.1, the accepting cycle $[(q_m,\emptyset,F_m,q_{m+1})\dots(q_l,\emptyset,F_l,q_m)]$ of $\pi^\dagger$ can be replaced by an accepting self-loop $(q_m,\emptyset,\mathcal{F},q_m)$. And thanks to property9.2, the transitions from q_{n-1} to q_m can be replaced by a single transition $(q_{n-1},\ell_{n-1} \oplus \ell_n,F_{n-1},q_m)$. The resulting path $\pi''' = (q_0,\ell_0 \oplus \ell_1,F_0,q_1)(q_1,\ell_1 \oplus \ell_2,F_1,q_2)\dots(q_{n-1},\ell_{n-1} \oplus \ell_n,F_{n-1},q_m)(q_m,\emptyset,\mathcal{F},q_m)^\omega$ is an accepting path of $\mathcal{T}'''$ that accepts σ, so $\sigma \in \mathscr{L}(\mathcal{T}''')$.

4. This is a classical optimization on Büchi automata.

□

5.4 Explicit Model checking using TGTA

As for the other variants of ω-automata, the automata-theoretic approach using TGTA has two important operations: the construction of a TGTA $\mathcal{T}$ recognizing the negation of the stutter-invariant LTL property φ and the emptiness check of the product $(\mathcal{K} \otimes \mathcal{T})$ of the Kripke structure $\mathcal{K}$ with $\mathcal{T}$. Currently, the TGTA $\mathcal{T}$ is built from a TGBA obtained from the translation of φ. In future work we plan to implement a direct translation from LTL$\setminus$X to TGTA, but the construction presented above is enough to show the benefits of using TGTAs, and makes it easier to understand how TGTAs relates to TGBAs.

5.4.1 Synchronous Product of a TGTA with a Kripke Structure

The product of a TGTA $\mathcal{T}$ with a Kripke structure $\mathcal{K}$ is an $\emptyset$-TGTA $(\mathcal{K} \otimes \mathcal{T})$ whose language is the intersection of both languages, i.e, $\mathscr{L}(\mathcal{K} \otimes \mathcal{T}) = \mathscr{L}(\mathcal{K}) \cap \mathscr{L}(\mathcal{T})$.

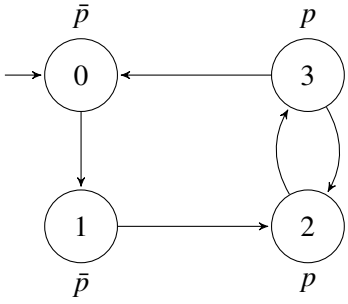
Comparing this definition with the previous two products (for TGBA and TA) shows the double inheritance of TGTA. This product is similar to the product between a TA and a Kripke structure (Definition 25 page 47), except that it does not deal with livelock acceptance states and implicit stuttering. It is also similar to the product of a TGBA with a Kripke structure (Definition 22 page 31), except for the use of changesets on transitions, and the initial labels (U).

Definition 31. *For a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ and a TGTA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$, the product $\mathcal{K} \otimes \mathcal{T}$ is a $\emptyset$-TGTA $\langle \mathcal{S}_\otimes, I_\otimes, U_\otimes, \delta_\otimes, \mathcal{F}_\otimes \rangle$ where*

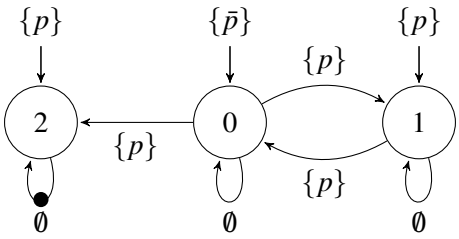
- $\mathcal{S}_\otimes = \mathcal{S} \times Q$,
- $I_\otimes = \{(s,q) \in \mathcal{S}_0 \times I \mid l(s) \in U(q)\}$,
- $\forall (s,q) \in I_\otimes, U_\otimes((s,q)) = \{l(s)\}$,

- $\delta_{\otimes} = \{((s,q),k,F,(s',q')) \mid (s,s') \in \mathcal{R}, (q,k,F,q') \in \delta, k = (l(s) \oplus l(s'))\}$,
- $\mathcal{F}_{\otimes} = \mathcal{F}$.

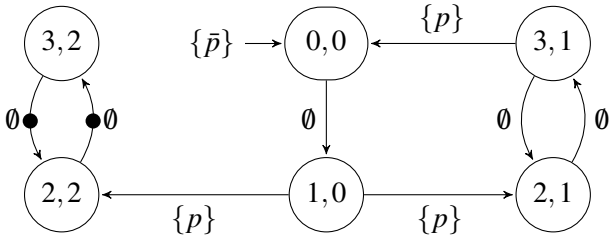
Property 10. *We have $\mathscr{L}(\mathcal{K} \otimes \mathcal{T}) = \mathscr{L}(\mathcal{K}) \cap \mathscr{L}(\mathcal{T})$ by construction.*



(a) A Kripke structure $\mathcal{K}$



(b) A TGTA $\mathcal{T}$ for $\mathsf{FG}\,p$



(c) Synchronous Product $\mathcal{K} \otimes \mathcal{T}$

Figure 5.6: Example of a Synchronous Product $\mathcal{K} \otimes \mathcal{T}$ between a Kripke structure $\mathcal{K}$ and a TGTA $\mathcal{T}$ recognizing the LTL formula $\mathsf{FG}\,p$, with acceptance conditions indicated by the black dot $\bullet$.

Figure 5.6 shows an example of a Synchronous Product between a Kripke structure $\mathcal{K}$ and a TGTA $\mathcal{T}$ recognizing the LTL formula $\mathsf{FG}\,p$. Each state of $\mathcal{K}$ is numbered and labeled with the set of atomic propositions (of $\Sigma = \{p\}$) that hold in this state. In the TGTA representing the product $\mathcal{K} \otimes \mathcal{T}$, the states are labeled with pairs of the form (s,q) where $s \in \mathcal{K}$ and $q \in \mathcal{T}$.

In this example, we can notice that this product using TGTA is smaller than the product using Büchi automata presented in Figure 2.12 (page 32) ,i.e., the product between the same Kripke structure $\mathcal{K}$ and the TGBA $\mathcal{A}$ recognizing the LTL formula $\mathsf{FG}\,p$. Indeed, the synchronization of the TGTA $\mathcal{T}$ with the stuttering parts of $\mathcal{K}$ produces a smaller product than in the case of TGBA $\mathcal{A}$ (compare the (sub-)product of $\mathcal{T}$ and $\mathcal{A}$ with the stuttering cycle of $\mathcal{K}$, i.e, the cycle between the states 2 and 3).

5.4.2 Emptiness check (the same as TGBA)

Since a product of a TGTA with a Kripke structure is an $\emptyset$-TGTA, we only need an emptiness check algorithm for an $\emptyset$-TGTA automaton. An $\emptyset$-TGTA can be seen as a TGBA whose transitions are labeled by changesets instead of valuations of atomic propositions. When checking a TGBA for

emptiness, we are looking for an accepting cycle that is reachable from an initial state. When checking an $\emptyset$-TGTA for emptiness, we are looking exactly for the same thing. Therefore, because emptiness check algorithms do not look at transitions labels, the same emptiness check algorithm used for the product using TGBA (Algorithm 1) can also be used for the product using TGTA.

This is a nice feature of TGTA, not only because it gives us a one-pass emptiness check, but also because it eases the implementation of the TGTA approach in our tool Spot or in any other TGBA-based model checker. We only need to implement the conversion of TGBA to TGTA and the product between a TGTA and a Kripke structure. We discuss our implementation in the next section.

5.5 Experimental evaluation of TGTA

In order to evaluate the TGTA approach against the TGBA (BA) and TA approaches, an experimentation was conducted under the same conditions as Section 3.6, i.e., within the same CheckPN tool on top of Spot and using the same benchmark Inputs (formulas and models) used in the experimental comparison of BA, TGBA, TA and STA, see Section 3.6.2 page 53.

5.5.1 Implementation

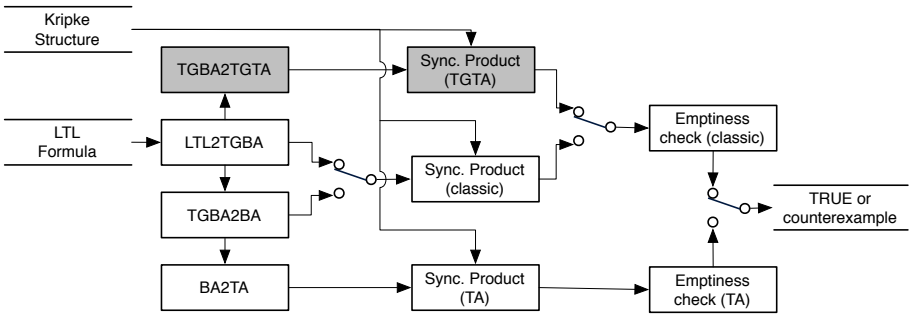


Figure 5.7: The experiment's architecture in SPOT. Three command-line switches control which one of the approaches is used to verify an LTL formula on a Kripke structure. The new components required by the TGTA approach are outlined in Gray.

Figure 5.7 shows the building blocks we used to implement the three approaches. The automaton used to represent the property to check has to be synchronized with a Kripke structure representing the model. Depending on the kind of automaton (TGBA, BA, TA, TGTA), this synchronous product is implemented differently. Only the TGBA and BA approaches can share the same product implementation. Approaches TA and TGTA require a dedicated product computation.

The TGBA, BA, and TGTA approaches share the same emptiness check, while a dedicated algorithm is required by the TA approach. In Figure 5.7, no direct translation is provided from LTL to TGTA (this is also true for BA and TA). This could be investigated in future work, the need being, so far, to assess their effectiveness before optimizing the translation process.

The time spent doing the conversion from LTL to TGBA and then to TGTA (bisimulation included) is measured in the benchmark of the next section (see tables column "T_φ" in centiseconds). This translation process is almost instantaneous, and even if its runtime could be improved (for instance with a direct translation from LTL to TGTA) it is clearly a non significant part of the run time of the different model checking approaches, where all the time is spent performing the emptiness check of the product (built on-the-fly) between the Kripke structure and the property automaton.

5.5.2 Results

Table 5.1 and Table 5.2 shows how for TGBA, TA and TGTA approaches deal with toy models and random formulas. We omit data for BA since they are always outperformed by TGBA. Table 5.5 and Table 5.6 show toy models against weak-fairness formulas.

Table 5.3 and Table 5.4 show the results of the two cases studies against random, weak-fairness, and dedicated formulas issued from the studies.

These tables separate cases where formulas are verified from cases where they are violated. In the former (Tables 5.1, 5.5 and 5.3), no counterexample are found and the full state-space had to be explored; in the latter (Tables 5.2, 5.6 and 5.4) the on-the-fly exploration of the state-space stopped as soon as the existence of a counterexample could be computed.

All values shown in all tables are averaged over 100 different formulas. Indeed, we checked each model against 100 random and 100 weak-fairness formulas that had no counterexample, and against 100 random and 100 weak-fairness formulas that had a counterexample. The average and maximum are computed separately on each model against each set of formulas.

Column-wise, these tables show the average and maximum sizes (states and transitions) of: (1) the automata $A_{\neg\varphi_i}$ expressing the properties φ_i; (2) the products $A_{\neg\varphi_i} \otimes \mathcal{K}$ of the property with the model; and (3) the subset of this product that was actually explored by the emptiness check. The emptiness check values show a third column labeled "T": this is the time (in hundredth of seconds, a.k.a. centiseconds) spent doing that emptiness check, including the on-the-fly computation of the subset of the product that is explored. In the same way, the column "T_φ" shows the time (in centiseconds) spent constructing the property automata $A_{\neg\varphi_i}$ from the formulas (this time is negligible compared to that of the emptiness check).

Figure 5.8 compares the number of visited transitions when running the emptiness check; plotting TGTA against TA and TGBA. This gives an idea of their relative performance. Each point corresponds to one of the 5600 evaluated formulas (2800 violated with counterexample as black circles, and 2800 verified having no counterexample as green crosses). Each point below the diagonal is in favor of TGTA while others are in favor of the other approach. Axes are displayed using a logarithmic scale. No comparison is presented with BA since they are less efficient than TGBA, according to the experimental evaluation presented in section 3.6.3 page 54.

All these experiments were run on a 64bit Linux system running on an Intel(R) 64-bit Xeon(R) @2.00GHz, with 10GB of RAM.

			Verified properties (no counterexample)							
			Automaton			Full product		Emptiness check		
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	TGBA	avg	12	460	1	2753024	14166594	2753024	14166594	18656
		max	54	2816	6	11010896	108553439	11010896	108553439	80455
	TA	avg	83	4324	12	2600539	10293714	4498653	17832896	30334
		max	509	37096	240	9384663	42614845	18769326	85229690	122867
	TGTA	avg	67	4584	14	**2474307**	**9810328**	**2474307**	**9810328**	**16630**
		max	349	49259	290	9596713	43944762	9596713	43944762	62911
Ring6	TGBA	avg	12	411	0	1398021	10139043	1398021	10139043	2325
		max	49	3788	4	4269384	50547840	4269384	50547840	10181
	TA	avg	69	3118	9	1097667	6606130	2155839	12981954	3744
		max	509	25773	150	2851152	19625864	5702304	39251728	11149
	TGTA	avg	57	3283	10	**1075198**	**6646513**	**1075198**	**6646513**	**1884**
		max	349	31401	182	2853568	20290856	2853568	20290856	5247
FMS5	TGBA	avg	9	245	0	2038054	15037218	2038054	15037218	3017
		max	28	2612	4	9132417	89397363	9132417	89397363	15594
	TA	avg	48	1643	5	1401286	11000449	1496882	11732337	3084
		max	282	11007	98	6109887	54324903	6368802	54324903	14894
	TGTA	avg	38	1713	6	**1338609**	**10551757**	**1338609**	**10551757**	**2621**
		max	247	13150	125	6109887	54324903	6109887	54324903	12570
Kanban5	TGBA	avg	8	166	0	3356053	32800737	3356053	32800737	5253
		max	60	1994	4	20253072	258315134	20253072	258315134	44281
	TA	avg	44	1659	3	2433853	23389805	2433853	23389805	4871
		max	277	21251	80	15272712	161364553	15272712	161364553	33129
	TGTA	avg	36	1779	4	**2285091**	**22038994**	**2285091**	**22038994**	**4385**
		max	210	21896	89	14059472	149569931	14059472	149569931	34534
Philo10	TGBA	avg	14	636	1	4668178	27564474	4668178	27564474	9352
		max	81	5397	6	17947837	119545256	17947837	119545256	40273
	TA	avg	71	4092	20	2334154	19893200	2334154	19893200	7742
		max	412	55321	235	8378151	74240975	8378151	74240975	27233
	TGTA	avg	55	4221	23	**2047915**	**17440829**	**2047915**	**17440829**	**6650**
		max	269	49089	256	7381980	61901565	7381980	61901565	24212
Robin15	TGBA	avg	15	664	0	2682881	30981263	2682881	30981263	9192
		max	82	5792	4	13129644	225905117	13129644	225905117	62328
	TA	avg	95	6048	15	2027251	17168855	3928111	33295175	12207
		max	481	43547	149	8169472	68645888	16338944	137291776	48584
	TGTA	avg	85	6725	17	**2042586**	**17645326**	**2042586**	**17645326**	**6370**
		max	414	46843	137	8097792	70340609	8097792	70340609	25213

Table 5.1: Comparison of the three approaches on toy examples with **random formulæ, when counterexamples do not exist.**

			Violated properties (a counterexample exists)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	TGBA	avg	14	646	0	8 118 953	42 865 329	600 828	2 325 266	3 892
		max	47	3 392	5	50 297 469	364 396 821	3 632 979	22 095 045	24 665
	TA	avg	90	5 683	15	8 592 222	33 972 863	546 562	1 886 029	3 683
		max	656	62 947	134	56 166 663	224 108 070	3 178 506	11 478 151	22 221
	TGTA	avg	68	5 389	16	**7 915 691**	**31 411 013**	**535 126**	**1 850 064**	**3 540**
		max	321	40 978	142	55 395 527	221 326 872	3 385 859	12 455 412	22 971
Ring6	TGBA	avg	17	713	1	3 163 763	26 260 802	854 229	5 805 080	1 329
		max	51	3 788	4	9 855 589	108 310 156	5 914 415	65 020 741	13 999
	TA	avg	125	7 462	15	2 846 468	18 131 253	646 844	**3 858 721**	1 198
		max	472	41 068	162	8 960 960	62 840 996	4 923 301	36 659 892	10 659
	TGTA	avg	99	7 247	15	**2 605 476**	**17 026 945**	**640 074**	3 911 536	**1 085**
		max	383	39 636	156	8 712 424	65 764 753	5 438 836	40 403 912	10 615
FMS5	TGBA	avg	14	496	1	10 174 866	95 061 682	1 602 719	10 636 471	2 148
		max	96	4 200	3	40 672 843	404 072 123	11 075 871	110 774 947	18 644
	TA	avg	90	4 112	9	8 418 689	70 784 299	1 044 896	7 693 244	1 968
		max	390	22 702	78	31 864 749	267 656 316	8 518 894	63 178 516	16 437
	TGTA	avg	72	4 005	10	**7 629 575**	**64 829 082**	**934 129**	**6 888 487**	**1 709**
		max	325	23 548	82	29 733 270	247 258 461	7 193 693	60 083 469	14 726
Kanban5	TGBA	avg	13	355	0	10 952 627	119 272 618	1 153 850	8 728 297	1 473
		max	87	2 184	3	43 038 083	587 543 197	13 023 468	185 410 581	26 750
	TA	avg	75	2 870	6	8 429 948	83 120 910	670 172	5 871 206	1 260
		max	473	30 375	94	31 602 068	331 833 407	7 673 335	77 363 942	15 562
	TGTA	avg	63	2 986	7	**7 919 885**	**79 262 186**	**631 528**	**5 523 476**	**1 113**
		max	321	25 589	104	31 771 853	346 223 237	7 673 335	77 364 390	15 411
Philo10	TGBA	avg	13	590	0	19 118 012	195 258 228	1 518 296	7 522 623	2 651
		max	74	5 928	3	84 068 722	1 377 479 362	9 032 250	53 881 112	15 880
	TA	avg	92	5 690	14	16 012 398	148 019 289	900 960	6 499 587	2 558
		max	356	46 498	164	66 535 322	696 331 784	7 342 016	64 470 840	23 509
	TGTA	avg	76	5 797	16	**14 738 127**	**138 827 585**	**804 739**	**5 945 101**	**2 273**
		max	352	37 193	147	66 964 226	712 347 867	6 062 800	52 054 854	18 939
Robin15	TGBA	avg	21	965	1	5 514 573	67 871 153	1 676 314	18 343 280	5 475
		max	74	5 928	7	15 127 431	281 880 971	13 627 374	191 356 248	52 962
	TA	avg	164	11 366	21	4 869 334	41 343 229	**1 248 873**	**10 418 682**	4 110
		max	854	62 419	115	11 791 872	104 488 704	7 767 216	74 241 202	27 244
	TGTA	avg	133	11 564	22	**4 637 571**	**40 182 518**	1 283 417	10 882 928	**3 999**
		max	485	77 546	140	11 726 848	97 390 592	8 539 312	82 390 194	27 829

Table 5.2: Comparison of the three approaches on toy examples with **random formulæ, when counterexamples exist.**

				Verified properties (no counterexample)							
				Automaton			Full product		Emptiness check		
				st.	tr.	T_φ	st.	tr.	st.	tr.	T
PolyORB 3/3/2	RND	TGBA	avg	16	715	1	109158	328976	109158	328976	601
			max	136	5792	4	373039	2138028	373039	2138028	2207
		TA	avg	94	5768	17	94283	205464	158401	346203	867
			max	481	43547	183	261318	604061	522636	1208122	2753
		TGTA	avg	78	5890	19	**97432**	**213975**	**97432**	**213975**	**530**
			max	414	46843	241	296747	677842	296747	677842	1587
	WFair	TGBA	avg	5	72	0	96435	277426	96435	277426	524
			max	18	337	2	368898	1450005	368898	1450005	2049
		TA	avg	79	1106	1	120479	262456	121096	263776	652
			max	351	5920	8	633208	1408943	633208	1408943	3359
		TGTA	avg	34	556	1	**88010**	**191692**	**88010**	**191692**	**475**
			max	120	2591	4	345486	798084	345486	798084	1828
	Φ_1	TGBA	–	7	576	1	345241	760491	**345241**	760491	**1688**
		TA	–	80	14590	8	**342613**	**742815**	685226	1485630	3374
		TGTA	–	79	17153	12	345277	753798	345277	**753798**	1697
MAPK 8	RND	TGBA	avg	10	333	1	2808973	25212509	2808973	25212509	5325
			max	64	3813	5	13636352	147555158	13636352	147555158	28979
		TA	avg	50	2239	9	1724291	20203618	1724291	20203618	4852
			max	336	22614	149	8469258	108847708	8469258	108847708	25029
		TGTA	avg	40	2364	10	**1574850**	**18534045**	**1574850**	**18534045**	**4289**
			max	279	25085	180	8469258	108847708	8469258	108847708	27737
	WFair	TGBA	avg	5	39	0	3898645	34822961	3898645	34822961	7344
			max	21	198	2	14452198	162156912	14452198	162156912	31117
		TA	avg	44	407	1	2112810	24492299	2112810	24492299	6126
			max	171	1920	3	6110748	75624744	6110748	75624744	20653
		TGTA	avg	18	184	0	**1871267**	**21752673**	**1871267**	**21752673**	**5258**
			max	68	915	2	6110643	75624228	6110643	75624228	19865
	Φ_2	TGBA	–	6	165	1	46494	302350	46494	302350	50
		TA	–	9	293	2	**33376**	**289235**	**33376**	**289235**	**50**
		TGTA	–	8	452	1	**33376**	**289235**	**33376**	**289235**	**49**

Table 5.3: Comparison of the three approaches for the **case studies when counterexamples do not exist.**

				Violated properties (a counterexample exists)								
				Automaton			Full product		Emptiness check			
				st.	tr.	T_φ	st.	tr.	st.	tr.	T
PolyORB 3/3/2	RND	TGBA	avg	14	760	1	132 232	380 305	53 901	128 960	291
			max	65	7 004	9	1 295 661	5 045 814	192 692	514 905	1 049
		TA	avg	81	6 223	29	135 237	296 081	56 518	126 300	311
			max	540	59 971	574	1 377 784	2 976 709	243 950	547 459	1 284
		TGTA	avg	68	6 815	33	**131 544**	**289 575**	**52 399**	**116 656**	**289**
			max	368	67 215	821	1 106 728	2 391 169	250 279	561 571	1 351
	WFair	TGBA	avg	5	66	0	96 849	258 704	69 053	177 391	372
			max	22	516	2	432 240	1 584 372	193 264	1 031 898	1 110
		TA	avg	82	1 157	1	201 879	434 747	113 705	250 573	615
			max	262	4 636	5	560 685	1 194 198	254 328	594 751	1 358
		TGTA	avg	32	520	1	**91 642**	**197 811**	**66 987**	**147 418**	**361**
			max	142	3 266	5	393 006	836 499	153 963	372 478	840
MAPK 8	RND	TGBA	avg	13	513	1	27 266 223	405 363 157	1 850 695	12 575 125	2 702
			max	59	4 298	2	91 059 214	1 963 331 216	13 284 206	111 916 150	21 771
		TA	avg	88	4 818	13	22 458 466	298 950 743	965 567	9 245 311	2 392
			max	334	35 401	165	90 049 281	1 300 904 178	5 241 327	61 655 512	14 227
		TGTA	avg	71	4 822	14	**19 806 355**	**264 723 502**	**954 582**	**9 085 179**	**2 243**
			max	283	38 342	196	52 079 530	722 041 705	5 265 831	61 724 760	13 674
	WFair	TGBA	avg	6	81	0	20 904 101	300 416 331	1 962 765	13 896 236	3 129
			max	23	414	2	101 813 441	1 681 905 871	9 463 562	72 266 873	16 928
		TA	avg	88	1 212	1	27 535 941	358 622 897	1 112 324	11 462 361	3 002
			max	245	3 749	5	93 441 682	1 258 076 980	5 203 746	59 642 346	16 992
		TGTA	avg	38	598	1	**16 051 206**	**211 401 085**	**964 009**	**9 865 703**	**2 462**
			max	106	2 437	3	91 711 581	1 302 884 523	4 279 478	53 059 472	11 687

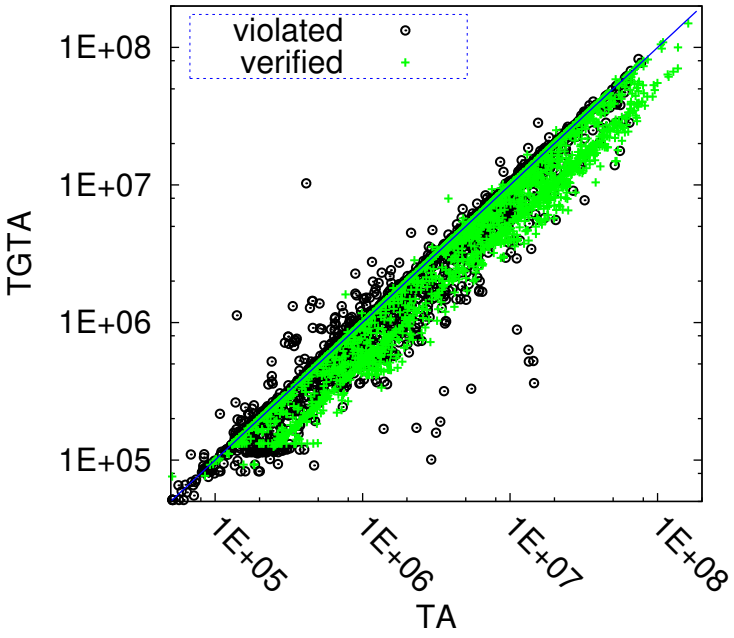
Table 5.4: Comparison of the three approaches for the **case studies when counterexamples exist.**

			Verified properties (no counterexample)							
			Automaton			Full product		Emptiness check		
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	TGBA	avg	4	52	0	3071854	15745483	3071854	15745483	21163
		max	17	357	2	9551018	87453872	9551018	87453872	70107
	TA	avg	50	564	1	3018533	11771410	3436223	13401691	23597
		max	184	3362	4	10750205	42228050	11867218	47116180	78154
	TGTA	avg	25	368	1	**2576578**	**10081563**	**2576578**	**10081563**	**17572**
		max	127	2998	3	6567277	26641077	6567277	26641077	43486
Ring6	TGBA	avg	5	64	0	1121590	7920270	1121590	7920270	1903
		max	25	357	2	3873576	38047692	3873576	38047692	7910
	TA	avg	58	682	1	1118511	6494556	1774991	10328239	3046
		max	183	3353	3	3383372	22481284	6766744	44962568	11037
	TGTA	avg	31	476	1	**878913**	**5232410**	**878913**	**5232410**	**1567**
		max	127	2998	3	2700824	18878992	2700824	18878992	4862
FMS5	TGBA	avg	4	41	0	3030750	23589355	3030750	23589355	4674
		max	12	212	2	11882973	114787553	11882973	114787553	23008
	TA	avg	48	515	1	2259230	17600147	2259230	17600147	4589
		max	171	4256	3	9764223	79707273	9764223	79707273	18521
	TGTA	avg	20	258	1	**1908957**	**15050447**	**1908957**	**15050447**	**3711**
		max	71	1295	3	7100604	58423901	7100604	58423901	13774
Kanban5	TGBA	avg	4	33	1	3098615	29435137	3098615	29435137	4712
		max	21	168	2	13014212	187004177	13014212	187004177	26143
	TA	avg	36	306	1	2083777	19156589	2083777	19156589	4112
		max	140	2049	3	11039112	106534190	11039112	106534190	23453
	TGTA	avg	16	182	1	**1876505**	**17440799**	**1876505**	**17440799**	**3512**
		max	60	1185	2	10017560	98020083	10017560	98020083	20406
Philo10	TGBA	avg	4	35	0	4255979	26793678	4255979	26793678	9064
		max	21	216	2	12974557	112561242	12974557	112561242	39806
	TA	avg	47	501	1	3229690	27631036	3229690	27631036	10439
		max	289	8199	6	10987384	94141317	10987384	94141317	38378
	TGTA	avg	18	216	0	**2278597**	**19552345**	**2278597**	**19552345**	**7184**
		max	85	2393	3	9026746	81612913	9026746	81612913	28914
Robin15	TGBA	avg	5	59	0	1782133	17996764	1782133	17996764	5749
		max	16	337	1	5029881	70344648	5029881	70344648	19263
	TA	avg	54	618	1	1508082	12171559	1802576	14591484	5731
		max	239	3706	4	3944448	32391168	7098368	61216768	22074
	TGTA	avg	28	420	1	**1363362**	**11099030**	**1363362**	**11099030**	**4345**
		max	120	2591	4	3649536	30605312	3649536	30605312	11669

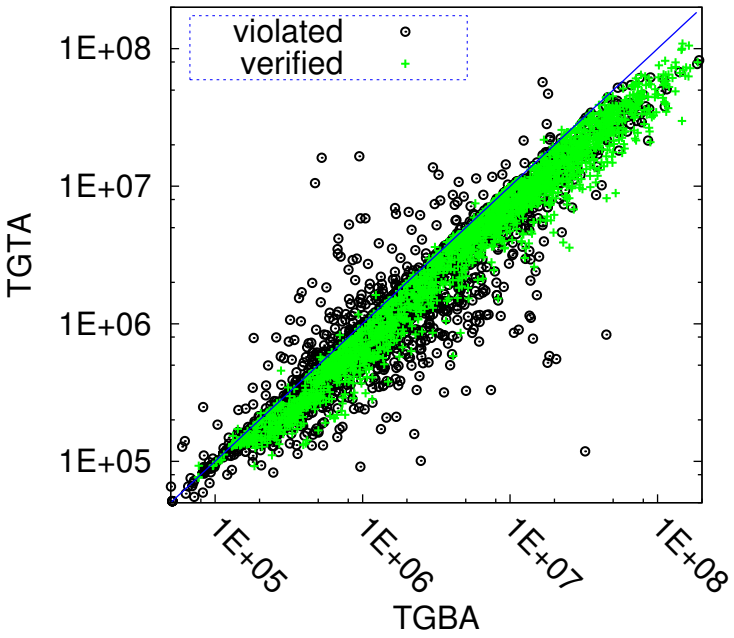
Table 5.5: Comparison of the three approaches on toy examples with **weak-fairness formulæ, when counterexamples do not exist.**

			Violated properties (a counterexample exists)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	TGBA	avg	5	78	0	3 731 947	17 751 951	522 414	1 759 041	3 438
		max	18	351	2	13 508 730	79 589 415	3 696 481	16 733 420	24 769
	TA	avg	95	1 400	2	10 950 499	42 790 086	641 997	2 123 401	4 370
		max	350	5 968	13	30 354 888	120 985 972	7 637 347	29 621 420	51 473
	TGTA	avg	36	598	1	**3 563 419**	**14 000 819**	**506 574**	**1 641 570**	**3 400**
		max	139	2 666	4	11 920 402	47 292 924	3 611 716	14 056 752	24 606
Ring6	TGBA	avg	7	123	1	2 000 787	15 868 857	890 567	5 736 568	1 410
		max	38	1 168	3	4 670 124	53 246 768	2 766 283	26 917 302	6 101
	TA	avg	128	2 009	2	3 456 785	21 069 783	905 265	5 156 327	1 621
		max	388	12 783	14	6 152 232	43 283 696	2 747 578	17 464 827	4 793
	TGTA	avg	55	997	1	**1 640 027**	**10 134 116**	**677 559**	**3 974 017**	**1 183**
		max	220	7 174	8	4 234 800	28 442 432	2 317 315	14 895 284	4 336
FMS5	TGBA	avg	5	68	0	7 978 129	75 701 408	1 372 877	7 544 397	1 607
		max	23	370	2	27 570 312	293 219 919	6 676 584	53 451 994	9 961
	TA	avg	71	871	1	10 007 988	81 736 556	848 557	5 707 348	1 522
		max	245	3 766	5	27 733 464	249 573 778	6 425 987	51 962 113	12 307
	TGTA	avg	33	474	1	**5 682 339**	**46 813 094**	**709 810**	**4 738 120**	**1 223**
		max	131	2 214	3	19 497 180	165 240 528	5 062 934	41 766 785	10 619
Kanban5	TGBA	avg	5	66	0	7 040 236	73 867 374	1 344 870	9 279 129	1 562
		max	17	305	2	25 423 161	388 961 954	7 950 633	80 041 971	11 336
	TA	avg	69	849	1	8 102 548	77 515 606	809 467	6 987 095	1 474
		max	237	3 655	3	32 732 392	329 125 090	5 501 242	53 375 472	10 218
	TGTA	avg	31	459	1	**4 837 392**	**46 966 950**	**718 473**	**6 247 954**	**1 262**
		max	128	2 446	3	21 299 320	218 268 197	4 580 718	46 290 149	8 953
Philo10	TGBA	avg	5	69	0	10 515 366	102 622 810	833 663	3 762 217	1 357
		max	25	392	3	24 496 745	402 197 001	5 017 174	32 273 634	9 802
	TA	avg	81	1 027	1	16 779 360	150 030 607	742 819	5 019 286	1 943
		max	267	4 339	5	43 186 049	394 172 486	6 703 064	55 546 036	20 284
	TGTA	avg	35	525	1	**7 800 504**	**70 806 206**	**442 830**	**2 727 715**	**1 103**
		max	116	3 004	4	20 986 825	203 614 146	3 405 659	28 381 991	9 712
Robin15	TGBA	avg	9	157	1	4 006 015	46 910 526	1 486 839	14 360 747	4 691
		max	44	1 168	2	14 802 883	195 152 925	5 666 009	78 112 291	21 619
	TA	avg	143	2 467	2	5 832 674	48 201 707	1 348 406	10 788 993	4 327
		max	466	12 783	14	18 204 684	151 155 748	6 486 142	53 115 008	18 844
	TGTA	avg	65	1 272	1	**3 428 985**	**28 693 985**	**1 188 348**	**9 648 116**	**3 825**
		max	228	7 174	7	12 249 088	102 016 000	4 618 329	38 837 339	13 884

Table 5.6: Comparison of the three approaches on toy examples with **weak-fairness formulæ, when counterexamples exist.**



(a) TGTA against TA approaches



(b) TGTA against TGBA approaches

Figure 5.8: Performance (number of transitions explored by the emptiness check) of TGTA against TA and TGBA.

5.5.3 Discussion

On verified properties the results are very straightforward to interpret when looking at the number of transitions explored by the emptiness check. TA outperform TGBA except for both Random and weak-fairness properties against Peterson, Ring, Robin and PolyORB. These are typical cases where the TA emptiness check has to perform two passes: this can be observed in the tables 5.1, 5.3 and 5.5 when the number of transitions visited by the emptiness check is on the average twice the number of transitions of the product.

In these three cases, the TGTA approach, with its single-pass emptiness check, is a clear improvement over TA. On the scatter plots of Figure 5.8a, these cases where the TGTA approach is twice faster than TA's, appear as a linear cloud of green crosses below the diagonal (we recall that the axes are displayed using a logarithmic scale).

In the other cases where TA need only one pass on the average (e.g. Kanban, MAPK, Philo), TGTA and TA have similar performance, with a slight advantage for TGTA because the products are smaller, especially for weak-fairness formulas because TGTA represent more concisely this kind of formulas using a large number of generalized acceptance conditions (similar to TGBA).

To summarize, the TGTA approach outperforms TGBA and TA approaches in all cases on verified properties.

On violated properties, we recall that it is difficult to interpret the scatter plots of results because the emptiness check is an on-the-fly algorithm. It stops as soon as it finds a counterexample. Thus, the exploration order of non-deterministic transitions of TGBA, TA and TGTA changes the number of states and transitions to be explored in the product before a counterexample is found.

However, if we analyze more precisely tables 5.2, 5.4 and 5.6, we observe that the TGTA approach produces the smallest products on the average. This allows the TGTA approach to seek a counterexample in a smaller product and therefore have a better chance to find it faster. Thus, we observe that in the majority of cases the emptiness check of TGTA approach explores less states and transitions on the average than TGBA or TA.

In the tables of results, we generally observe for both verified and violated properties that:

- Although the TGTA constructed from properties are usually larger than TGBA (and even larger than BA), the average sizes of the products in TGTA approach are smaller than the average sizes of the products in TGBA approach (and BA approach, see section 3.6.3 page 54). We believe this is due to the elimination of useless stuttering-transitions in TGTA (see Section 5.3.2). specificity.

- In addition, if we compare the automata sizes for TA versus TGTA, we observe that TGTA are smaller than TA in all tables in terms of average numbers of states, and in terms of average numbers of states and transitions for weak-fairness formulas (in Tables 5.2, 5.6, 5.3 and 5.4). We believe this is due to the fact that TGTA represent more concisely the LTL formulas using (multiple) generalized acceptance conditions, especially for weak-fairness formulas (for which the number of acceptance conditions is greater, in our experiments we have $|\mathcal{F}| = 3.51$ on the average for weak-fairness formulas, while $|\mathcal{F}| = 1.32$ on the average for random formulas). This is the consequence of the fact that a TA is built from a BA while a TGTA is built from a TGBA. Thus, TGTA can take advantage from the fact that

TGBA are more concise than BA. TGBA are smaller [49, 36] because they use generalized acceptance conditions. For example, the BA of $\varphi = \mathsf{GF}a \wedge \mathsf{GF}b$ (Figure 2.4a) contains 3 states and it is transformed [1] into a TA that contains 10 states (30 transitions); the TGBA of $\varphi = \mathsf{GF}a \wedge \mathsf{GF}b$ (Figure 2.7a) is only composed by one state and it is therefore transformed [1] into a TGTA that only contains 4 states (16 transitions).

5.5.4 Experimental Results once the TGBA is improved by simulation-reduction

Recently, a new TGBA optimization was added to SPOT including the simulation-reduction [3] of TGBA. Unfortunately, we did not have the time to implement this simulation-reduction for TA and TGTA. However, since the construction of TA and TGTA depend on TGBA (we recall that TA is constructed from BA, which is obtained from TGBA by degeneralization), then the reduction of TGBA can also reduce both TA and TGTA sizes.

The tables and scatter plots presented in the Appendix A show the impact of this optimization on the experimental results presented in the previous Section 5.5.2. This impact is positive for the three approaches. Indeed, if we compare the results of the previous Section 5.5.2 against the tables of Appendix A, we observe that the simulation-reduction of TGBA also reduces the TA and TGTA sizes.

These reductions produce smaller products on average and thus improve the performance of the three approaches. However, this does not change the result of the comparison of the three approaches: for verified formulas, TGTA remains more efficient than the other approaches; for violated formulas, the results are still difficult to interpret.

5.6 Conclusion

In the previous chapters, we have shown that TA outperformed BA and sometimes TGBA for unverified properties (i.e., when a counterexample was found). However, this was not the case when no counterexample was computed since the entire product had to be visited twice to check for each acceptance mode of a TA (Büchi acceptance or livelock-acceptance).

In this chapter, we propose a new type of ω-automaton for stutter-invariant properties, called Transition-based Generalized Testing Automata (TGTA).

TGTA combines advantages observed on both TA and TGBA:

- From TA, it reuses the labeling of transitions with changesets, and the elimination of the useless stuttering-transitions, but without requiring a second pass in the emptiness check of the product.
- From TGBA, it inherits the use of generalized acceptance conditions on transitions.

TGTA have been implemented in Spot easily, because only two new algorithms are required: the conversion of a TGBA into a TGTA, and a new definition of a product between a TGTA and a Kripke structure.

We have run benchmarks to compare TGTA against TA and TGBA (BA). Experiments reported that, in most cases, TGTA produce the smallest products on the average and they outper-

[1] http://spot.lip6.fr/ltl2tgba.html

form TA and TGBA when no counterexample is found in the system, but they are comparable when the property is violated, because in this case the on-the-fly algorithm stops as soon as it finds a counterexample without exploring the entire product.

We conclude that there is nothing to lose by using TGTA to verify stuttering-insensitive properties, since they are always at least as good as TA and TGBA.

We believe that TGTA is better than TA because TGTA does not require a second pass during the emptiness check and because TGTA represent more concisely the LTL formulas using (multiple) generalized acceptance conditions, as observed in our experiments for weak-fairness formulas (for which the number of acceptance conditions is greater than random formulas).

Compared to TGBA, we believe that TGTA is better thanks to the elimination of the useless stuttering-transitions during the TGTA construction (Section 5.3.2). This elimination exploits the fact that the TGTA are specific to stutter-invariant formulas, while the TGBA does not exploit at all this specificity.

After this elimination of useless stuttering-transitions, the obtained TGTA represents all the stuttering-transitions with only self-loops on all states (see the "stuttering-normalization constraint" of the TGTA Definition 30). This advantage of TGTA will be more clearly exploited in the symbolic approach presented in the next chapter. This symbolic model checking approach using TGTA allows us to tackle much larger state-spaces than in explicit model checking.

Part III

Using TGTA to improve Symbolic/Hybrid Model Checking

CHAPTER 6

Symbolic LTL Model Checking using TGTA

Contents

6.1 Introduction

In the previous chapter, we showed how to generalize the Testing Automata (TA) using several acceptance sets, and allowing a single-pass efficient emptiness check. Our experimental comparison showed these Transition-based Generalized Testing Automata (TGTA) to be superior to Büchi Automata in the *explicit approach* for model-checking of stutter-invariant properties. In this explicit approach, the automata and their products were represented as explicit graphs.

Another implementation of this procedure is the *symbolic approach* where the automata and their products are represented by means of decision diagrams (a concise way to represent large

sets or relations) [17]. Encoding generalized Büchi automata is pretty common [75]. With such encoding, we can compute, in one step, the sets of all direct successors (*PostImage*) or predecessors (*PreImage*) of any set of states. Using this technique, there have been a lot of propositions for symbolic emptiness-check algorithms [43, 80, 57]. These symbolic algorithms manipulate standard BFS-based fixed-points on the transition relation of the product which can be optimized using saturation techniques [20, 85].

To the best of our knowledge, these algorithms do not provide efficient optimizations specific to stutter-invariant properties, and Testing Automata have never been used in symbolic model checking. In this chapter, we propose and evaluate a symbolic approach for model checking using TGTA [7], and compare it to the symbolic approach using TGBA. In particular, we show that the computation of fixpoints on the transition relation of the product can be sped up with a dedicated evaluation of stuttering transitions. The implementation uses the saturation technique introduced by [20] that departs from standard BFS-based approaches for the symbolic fixed-point computation. Saturation nicely fits well with TGTA. Indeed, we exploit a separation of the transition relation into two terms, one of which greatly benefits from saturation techniques.

This chapter is organized as follows. Section 6.2 presents the symbolic model-checking approach for TGBA. For generality we define our symbolic structures using predicates over state variables in order to remain independent of the variant of Decision Diagrams used to actually implement the approach. Section 6.3 focuses on the encoding of TGTA in the same framework. We first show how a TGTA can be encoded, then we show how to improve the encoding of the Kripke structure and the product to benefit from saturation in the encoding of stuttering transitions in the TGTA. Finally, Section 6.4 compares the two approaches experimentally with an implementation that uses hierarchical Set Decision Diagrams (SDD) [85] (a particular type of Decision Diagrams on integer variables, on which we can apply user-defined operations). We concentrate on a comparison of TGBA versus TGTA and on the impact of the saturation technique. On our large, BEEM-based benchmark (presented in Section 6.4.3) and huge set of LTL formulas, our symbolic encoding of TGTA appears to be superior to TGBA.

6.2 Symbolic LTL Model Checking

We first present how to perform the automata-theoretic approach to LTL model checking using symbolic encodings of TGBA and Kripke structures. This setup will serve as a baseline to measure our improvements from later sections.

6.2.1 Symbolic Kripke Structure

In symbolic model checking we encode such a Kripke structure with predicates that represent sets of states or transitions [79]. These predicates are then implemented using decision diagrams [17].

Definition 32 (Symbolic Kripke Structure)**.** *A Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ can be encoded by the following predicates where $s, s' \in \mathcal{S}$ and $\ell \in \Sigma$:*

- $P_{\mathcal{S}_0}(s)$ *is true* iff $s \in \mathcal{S}_0$,
- $P_{\mathcal{R}}(s, s')$ *is true* iff $(s, s') \in \mathcal{R}$,

- $P_l(s,\ell)$ *is true* iff $l(s) = \ell$.

In the sequel, we use the notations $S_0(s)$, $R(s,s')$ *and* $L(s,\ell)$ *instead of* $P_{S_0}(s)$, $P_{\mathcal{R}}(s,s')$ *and* $P_l(s,\ell)$. *A* Symbolic Kripke structure *is therefore a triplet of predicates* $K = \langle S_0, R, L\rangle$ *on state variables.*

Variables s and s' used above are typically implemented using decision diagrams to represent either a state or a set of states. In a typical encoding [17], states are represented by conjunctions of Boolean variables. For instance if $\mathcal{S} = \{0,1\}^3$, a state $s = (1,0,1)$ would be encoded as $s_1\bar{s}_2s_3$. Similarly, s_1s_3 would encode the set of states $\{(1,0,1),(1,1,1)\}$. With this encoding, S_0, R and L are propositional formulas which can be implemented with BDDs or other kind of decision diagrams. In our implementation, we used SDDs on integer variables [85].

6.2.2 Symbolic Büchi Automata TGBA

We chose the TGBA to represent the negation of the LTL property to verify, because generalized acceptance is classically used in symbolic model checking [43] and using transition-based acceptance is not a problem [75]. People working with Generalized Büchi automata (GBA) can adjust to our definitions by "pushing" the acceptance of states to their outgoing transitions [36].

Like Kripke structures, a TGBA can be encoded by predicates [79] on state variables.

Definition 33 (Symbolic TGBA)**.** *A TGBA* $\langle Q, I, \delta, \mathcal{F}\rangle$ *is symbolically encoded by a the predicates* $\langle I, \Delta, \{\Delta_f\}_{f\in\mathcal{F}}\rangle$ *where:*

- $I(q)$ *is true* iff $q \in I$,
- $\Delta(q,\ell,q')$ *is true* iff $\big(\exists F \in 2^{\mathcal{F}}, (q,\ell,F,q') \in \delta\big)$,
- *For every* $f \in \mathcal{F}$, *the predicate* Δ_f *is defined by:* $\Delta_f(q,\ell,q')$ *is true* iff $\big(\exists (q,\ell,F,q') \in \delta, f \in F\big)$.

6.2.3 Symbolic Product of a TGBA with a Kripke structure

We now show how to build a synchronous product by composing the symbolic representations of a TGBA with that of a Kripke structure, inspired from Sebastian *et al.* [79].

Definition 34 (Symbolic Product for TGBA)**.** *Given a Symbolic Kripke structure* $K = \langle S_0, R, L\rangle$ *and a Symbolic TGBA* $A = \langle I, \Delta, \{\Delta_f\}_{f\in\mathcal{F}}\rangle$ *sharing a set AP of atomic propositions, the Symbolic Product* $K \otimes A = \langle P_0, T, \{T_f\}_{f\in\mathcal{F}}\rangle$ *is defined by the predicates* P_0, T *and* T_f *encoding respectively the set of initial states, the transition relation and the acceptance transitions of the product:*

- (s,q) *denotes the state variables of the product (*s *for the Kripke structure and* q *for TGBA),*
- $P_0(s,q) = S_0(s) \wedge I(q)$,
- $T((s,q),(s',q')) = \exists \ell \big[R(s,s') \wedge L(s,\ell) \wedge \Delta(q,\ell,q')\big]$, *where* (s',q') *encodes the next state variables,*
- $\forall f \in \mathcal{F}, T_f((s,q),(s',q')) = \exists \ell \big[R(s,s') \wedge L(s,\ell) \wedge \Delta_f(q,\ell,q')\big]$.

The labels ℓ are used to ensure that a transition (q,ℓ,q') of A is synchronized with a state s of K such that $L(s,\ell)$. This way, we ensure that the product recognizes only the executions of K that are also recognized by A. However we do not need to remember how product transitions are labeled to check $K \otimes A$ for emptiness. Therefore, a product can be seen as a TGBA without labels on transitions. This explains why the predicate $T((s,q),(s',q'))$ does not take as an argument a variable (ℓ) to encode the labels of transitions (as in $\Delta(q,\ell,q')$).

In symbolic model checkers, the exploration of the product is based on the following *PostImage* operation [79]. For any set of states encoded by a predicate P, $\mathit{PostImage}(P)\ (s',q') = \exists(s,q)\left[P(s,q) \wedge T((s,q),(s',q'))\right]$ returns a predicate on state variables (s',q') encoding the set of states reachable in one step from the set of states encoded by P.

Because in TGBA the acceptance conditions are based on transitions, we also define $\mathit{PostImage}(P,f)$ to compute the successors of P reached using the transitions labeled with the acceptance condition $f \in \mathcal{F}$: $\mathit{PostImage}(P,f)(s',q') = \exists(s,q)\left[P(s,q) \wedge T_f((s,q),(s',q'))\right]$.

These two operations are at the heart of the symbolic emptiness check presented in the next section.

6.2.4 Symbolic Emptiness Check algorithm

```
1 Input: PostImage, P_0 and F
2 begin
3     P ← Reach(P_0)
4     while P changes do
5         while P changes do
6             P ← PostImage(P)
7         for f in F do
8             P ← Reach(PostImage(P, f))
9     return P = ∅
```

```
1 Reach(P)
2     while P changes do
3         P ← P ∪ PostImage(P)
4     return P
```

Figure 6.1: Forward-variant of OWCTY, a symbolic emptiness check.

One way to check if a product is not empty is to find a reachable Strongly Connected Component that contains transitions from all acceptance sets (we call it an *accepting SCC*). Figure 6.1 shows such an algorithm implemented using symbolic operations. It mimics the algorithm FEASIBLE of Kesten et al. [57] and can be seen as a forward variant of OWCTY (One Way Catch Them Young [43]) that uses *PostImage* computations instead of *PreImage*. Line 3 computes the set P of all reachable states of the product. The main loop on lines 4–8 refines P at each iteration. Lines 5–6 keep only the states of P that can be reached from a cycle in P. Lines 7–8 then remove all cycles that never visit some acceptance set $f \in \mathcal{F}$. Eventually the main loop will reach a fixpoint where P contains all states that are reachable from an accepting SCC. The product is empty *iff* that set is empty.

There are many variants of such symbolic emptiness checks [43, 80, 57]. We selected this variant mainly for its simplicity, as our contributions are mostly independent of the chosen algorithm: essentially, we will improve the cost of computing $Reach(P)$ (used lines 3 and8).

6.3 TGTA-based Symbolic LTL Model Checking

In this section, we show how to encode a TGTA and a product for symbolic model checking using TGTA instead of TGBA.

6.3.1 Symbolic TGTA

A TGTA can be encoded symbolically in a similar way as we encoded a TGBA.

Definition 35 (Symbolic TGTA)**.** *A TGTA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$ is symbolically encoded by the predicates $\langle U_0, \Delta^{\oplus}, \{\Delta^{\oplus}_f\}_{f\in\mathcal{F}} \rangle$ where:*

- *$U_0(q,\ell)$ is true* iff $(q \in I) \wedge (U(q) = \ell)$
- *$\Delta^{\oplus}(q,k,q')$ is true* iff $\big(\exists F \in 2^{\mathcal{F}}, (q,k,F,q') \in \delta\big)$,
- *For every $f \in \mathcal{F}$, the predicate $\Delta^{\oplus}_f$ is defined by:*
  $\Delta^{\oplus}_f(q,k,q')$ is true iff $\big(\exists (q,k,F,q') \in \delta, f \in F\big)$.

The predicates encoding the symbolic TGTA are the same as those encoding the symbolic TGBA, but they are based on changesets (encoded by the variable k) instead of valuations (variable ℓ).

The predicate $U_0(q,\ell)$ encodes the set of initial states and their valuations. The predicate $\Delta^{\oplus}(q,k,q')$ encodes the set of transitions of the TGTA, with the variable k encodes the changeset between q and q'. For each $f \in \mathcal{F}$, a predicate $\Delta^{\oplus}_f(q,k,q')$ encodes the transitions labeled with the acceptance condition f.

6.3.2 Naive Symbolic Product of TGTA with a Kripke structure

The product between a TGTA and a Kripke structure is similar to the TGBA case, except that we have to deal with changesets. The transitions (s,s') of a Kripke structure that must be synchronized with a transition (q,k,q') of a TGTA are all the transitions such that the label of s and s' differs by the changeset k: in the definition below, this is encoded by the terms $L(s,\ell) \wedge L(s',\ell')$, with $(\ell \oplus \ell') = k$.

Definition 36 (Naive Symbolic Product for TGTA)**.** *Given a Symbolic Kripke structure $K = \langle S_0, R, L \rangle$ and a Symbolic TGTA $A^{\oplus} = \langle U_0, \Delta^{\oplus}, \{\Delta^{\oplus}_f\}_{f\in\mathcal{F}} \rangle$ sharing the same set of atomic propositions AP, the Symbolic Product $K \otimes A^{\oplus} = \langle P_0, T, \{T_f\}_{f\in\mathcal{F}} \rangle$ is defined by the following predicates:*

- *The set of initial states is encoded by: $P_0(s,q) = \exists \ell \big[S_0(s) \wedge L(s,\ell) \wedge U_0(q,\ell)\big]$.*
- *A naive encoding of the transition relation of the product is:*
  $T(s,q,s',q') = \exists k \big[R(s,s') \wedge \big(\exists \ell, \ell' [L(s,\ell) \wedge L(s',\ell') \wedge xor(\ell,k,\ell')]\big) \wedge \Delta^{\oplus}(q,k,q')\big]$, with the predicate $xor(\ell,k,\ell')$ is true iff $(\ell \oplus \ell') = k$

- *The definition of T_f is similar to T by replacing $\Delta^{\oplus}$ with $\Delta^{\oplus}_f$.*

The formulas of $PostImage(P)$ and $PostImage(P,f)$ in TGTA approach are the same as in the TGBA approach, with the new expressions of T and T_f defined above for $K \otimes A^{\oplus}$.

This naive definition of T contains the term $\exists \ell, \ell' [L(s,\ell) \wedge L(s',\ell') \wedge xor(\ell,k,\ell')]$ which requires several symbolic operations. In the next section, we show how to re-encode the transition relation of Kripke structures to remove this term.

6.3.3 Adjusting the Symbolic Transition Relation of the Kripke Structure to TGTA

In order to reduce the number of symbolic operations in T and T_f, we introduce a changeset-based encoding of a Kripke structure (only the transition relation changes).

Definition 37 (Changeset-based symbolic Kripke structure)**.** *A Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$, can be encoded by a changeset-based symbolic Kripke structure $K^{\oplus} = \langle S_0, R^{\oplus}, L \rangle$, where:*

- *the predicate $R^{\oplus}(s,k,s')$ is true* iff $((s,s') \in \mathcal{R} \wedge (l(s) \oplus l(s')) = k)$,
- *the predicates S_0 and L have the same definition as for a symbolic Kripke structure K of Definition 32.*

The changeset-based symbolic transition relation $R^{\oplus}(s,k,s')$ of a Kripke structure is similar to the symbolic transition relation $\Delta^{\oplus}(q,k,q')$ of a TGTA. It encodes the transitions $((s,s') \in \mathcal{R}$ with the variable k encodes the changeset between the two valuations $l(s)$ and $l(s')$.

In practice, the (changeset-based or not) symbolic transition relation of a Kripke structure should be constructed directly from the model and atomic propositions of the formula to check. In Section 6.4.2, we discuss how we build such changeset-based Kripke structures in our setup.

Adjusting the symbolic encoding of the Kripke structure to TGTA, allows us to obtain the following natural definition of the symbolic product using TGTA:

Definition 38 (Symbolic Product for TGTA)**.** *Given a changeset-based Symbolic Kripke structure $K^{\oplus} = \langle S_0, R^{\oplus}, L \rangle$ and a Symbolic TGTA $A^{\oplus} = \langle U_0, \Delta^{\oplus}, \{\Delta^{\oplus}_f\}_{f \in \mathcal{F}} \rangle$ sharing the same set of atomic propositions AP, the Symbolic Product $K^{\oplus} \otimes A^{\oplus} = \langle P_0, T, \{T_f\}_{f \in \mathcal{F}} \rangle$ is defined by the following predicates:*

- *The set of initial states is encoded by: $P_0(s,q) = \exists \ell \left[S_0(s) \wedge L(s,\ell) \wedge U_0(q,\ell) \right]$*
- *The transition relation of the product is: $T((s,q),(s',q')) = \exists k \left[R^{\oplus}(s,k,s') \wedge \Delta^{\oplus}\ (q,k,q') \right]$*
- *The definition of T_f is similar to T by replacing $\Delta^{\oplus}$ with $\Delta^{\oplus}_f$.*

The definitions of $PostImage(P)$ and $PostImage(P,f)$ are the same as in the TGBA approach, with the new expressions of T and T_f above.

As for the product in TGBA approach, the product in TGTA approach is a TGBA (or a TGTA) without labels on transitions, and the same emptiness check algorithm (Figure 6.1) can be used for the two products.

6.3.4 Exploiting stuttering transitions to Improve Saturation in the TGTA Approach

Among symbolic approaches for evaluating a fixpoint on a transition relation, the *saturation* algorithm offers gains of one to three orders of magnitude [20] in both time and memory, especially when applied to asynchronous systems [18].

The saturation algorithm does not use a breadth-first exploration of the product, i.e., each iteration in the function `Reach` (Figure 6.1) is not a "global" *PostImage*() computation. Saturation instead recursively repeats "local" fixed-points by recognizing and exploiting transitions locality and identity transformations on state variables [18]. This algorithm considers that the system state consists of n discrete variables encoded by a Decision Diagram, and that the transition relation is expressed as a disjunction of terms called transition clusters. Each cluster typically only reads or writes a limited subset consisting of $n' \leq n$ variables, called the *support* of the cluster. During the least fixpoint computing the reachable states, the saturation technique consists in reordering [51] the evaluation of ("local" fixed-points on) clusters in order to avoid the construction of (useless) intermediate decision diagram nodes.

The algorithm to determine an ordering for saturation is based on the support of each cluster.

We now show how to decompose the transition relation of the product $(K^{\oplus} \otimes A^{\oplus})$ to exhibit clusters having a smaller support, favoring the saturation technique.

We base our decomposition on the fact that in a TGTA, all stuttering transitions are self-loops and every state has a stuttering self-loop (see the stuttering-normalization constraint in Definition 30 of TGTA). Therefore, stuttering transitions in the Kripke structure can be mapped to stuttering transitions in the product regardless of the TGTA state.

Let us separate stuttering and non-stuttering transitions in the transition relation T of the symbolic product $K^{\oplus} \otimes A^{\oplus}$ (Definition 38):

$$T((s,q),(s',q')) = \left(R^{\oplus}(s,\emptyset,s') \wedge \Delta^{\oplus}(q,\emptyset,q')\right) \vee \left(\exists k \left[R^{\oplus}_{*}(s,k,s') \wedge \Delta^{\oplus}_{*}(q,k,q')\right]\right)$$

where $R^{\oplus}_{*}$ and $\Delta^{\oplus}_{*}$ encode respectively the non-stuttering transitions of the model and of the TGTA:

- $\Delta^{\oplus}_{*}(q,k,q')$ is true *iff* $\Delta^{\oplus}(q,k,q') \wedge (k \neq \emptyset)$
- $R^{\oplus}_{*}(s,k,s')$ is true *iff* $R^{\oplus}(s,k,s') \wedge (k \neq \emptyset)$

On the one hand, according to Definition 35, the predicate $\Delta^{\oplus}(q,\emptyset,q')$ is defined by the following equivalence:

$$\forall (q,q') \in Q^2 : \left[\Delta^{\oplus}(q,\emptyset,q') \Longleftrightarrow \left(\exists F \in 2^{\mathcal{F}}, (q,\emptyset,F,q') \in \delta\right)\right] \tag{6.1}$$

On the other hand, according to the stuttering-normalization constraint of Definition 30 of TGTA (i.e., all stuttering transitions are self-loops and every state has a stuttering self-loop):

$$\forall (q,q') \in Q^2 : \left[\left(\exists F \in 2^{\mathcal{F}}, (q,\emptyset,F,q') \in \delta\right) \Longleftrightarrow (q = q')\right] \tag{6.2}$$

Combining the two equations (6.1) and (6.2), we obtained the following third equivalence:

$$\forall (q,q') \in Q^2 : \left[\Delta^{\oplus}(q,\emptyset,q') \Longleftrightarrow (q = q')\right] \tag{6.3}$$

In the other words, the predicate $\Delta^{\oplus}(q,\emptyset,q')$ encodes the set of all self-loops of the TGTA and can be replaced by the identity predicate: $equal(q,q')$, simplifying T as:

$$T((s,q),(s',q')) = \underbrace{\left(R^{\oplus}(s,\emptyset,s') \wedge equal(q,q')\right)}_{T_{\emptyset}((s,q),(s',q'))} \vee \underbrace{\left(\exists k \left[R^{\oplus}_{*}(s,k,s') \wedge \Delta^{\oplus}_{*}(q,k,q')\right]\right)}_{T_{*}((s,q),(s',q'))} \tag{6.4}$$

The transition relation of equation (6.4) is a disjunction of T_*, synchronizing updates of both TGTA and Kripke structure, and $T_\emptyset$, corresponding to the stuttering transitions of the Kripke structure. Since all states in the TGTA have a stuttering self-loop, $T_\emptyset$ does not depend on the TGTA state. In practice, the predicate $equal(q,q')$ is an identity relation for variable q [18] and is simplified away (i.e., the term $T_\emptyset$ can be applied to a decision diagram without consulting or updating the variable q [51]). Hence q is not part of the clusters supports in $T_\emptyset$ (while q is part of the clusters supports in T_*). This gives more freedom to the saturation technique for reordering the application of clusters in $T_\emptyset$.

Note that in the product of TGBA with Kripke structure (Definition 34) there is no $T_\emptyset$ that could be extracted since there is no stuttering hypothesis in general. This severely limits the possibilities of the saturation algorithm in the TGBA approach.

In the symbolic emptiness check presented in Figure 6.1, the function `Reach` corresponds to a least fixpoint performed using saturation. As we shall see experimentally in the next section, the better encoding of $T_\emptyset$ (without q in its support) in the product of TGTA with Kripke structure, greatly favors the saturation technique, leading to gains of roughly one order of magnitude.

The improvement of T proposed in this section is not applicable to T_f because the term $\Delta^{\oplus}_{f}(q,\emptyset,q')$ encodes only a subset of self-loops of the TGTA (not the all self-loops as $\Delta^{\oplus}(q,\emptyset,q')$), hence the expressions of T_f and $Postimage(P,f)$ are the same as the previous section.

Table 6.1 summarizes all our definitions for the predicates encoding the automaton, the Kripke structure, and their product, in the four approaches presented in Sections 6.2 and 6.3.

The results of the TGTA based symbolic approach will be showed in the next section presenting our experimentation.

6.4 Experimental evaluation

We now compare the symbolic approaches presented in this chapter. The symbolic model-checking approach using TGBA, presented in Section 6.2 serves as our baseline. We first describe our implementation and selected benchmarks, prior to discussing the results.

		TGBA approach (our baseline) Section 6.2	**Naive TGTA approach** Sections 6.3.1 and 6.3.2	**First improvement of the TGTA approach** Sections 6.3.1 and 6.3.3	**Second improvement of the TGTA approach** Sections 6.3.1, 6.3.3 and 6.3.4
Automaton	Initial states	$Q_0(q)$	$U_0(q,\ell)$		
	Transitions	$\Delta(q,\ell,q')$	$\Delta^{\oplus}(q,k,q')$		
	Acceptance	$\forall f \in F, \Delta_f(q,\ell,q')$	$\forall f \in F, \Delta_f^{\oplus}(q,k,q')$		
Kripke struct.	Initial states	$S_0(s)$			
	Transitions	$R(s,s')$		$R^{\oplus}(s,k,s')$ (encoding $(s,s') \in R \wedge (L(s) \oplus L(s')) = k$)	
	State labels	$L(s,\ell)$			
Synchr. Product	Initial states $P_0(s,q) =$	$S_0(s) \wedge Q_0(q)$	$\exists \ell[S_0(s) \wedge L(s,\ell) \wedge U_0(q,\ell)]$		
	Transitions $T(s,q,s',q') =$	$\exists \ell[R(s,s') \wedge L(s,\ell) \wedge \Delta(q,\ell,q')]$	$\exists k,\ell,\ell'\big[R(s,s') \wedge \Delta^{\oplus}(q,k,q') \wedge L(s,\ell) \wedge L(s',\ell') \wedge \oplus(\ell,k,\ell')\big]$	$\exists k[R^{\oplus}(s,k,s') \wedge \Delta^{\oplus}(q,k,q')]$	$[R^{\oplus}(s,\emptyset,s') \wedge equal(q,q')] \vee (\exists k[R_*^{\oplus}(s,k,s') \wedge \Delta_*^{\oplus}(q,k,q')])$
	Acceptance $T_f(s,q,s',q') =$	$\exists \ell[R(s,s') \wedge L(s,\ell) \wedge \Delta_f(q,\ell,q')]$	$\exists k,\ell,\ell'\big[R(s,s') \wedge \Delta_f^{\oplus}(q,k,q') \wedge L(s,\ell) \wedge L(s',\ell') \wedge \oplus(\ell,k,\ell')\big]$	$\exists k[R^{\oplus}(s,k,s') \wedge \Delta_f^{\oplus}(q,k,q')]$	

Table 6.1: Predicates encoding the automata, the Kripke structures and their synchronous product in the different symbolic approaches.

6.4.1 Implementation

We have implemented the symbolic approach based on TGTA, Changeset-based symbolic Kripke structure and Saturation in LTL-ITS [1] tool, which already provides an implementation of the symbolic approach based on TGBA and Saturation. This tool is built on top of three libraries [2]: SDD/ITS, Spot, and LTSmin.

Spot is a model-checking library providing several bricks that can be combined to build model checkers [36]. In our implementation, we reused the modules providing a translation from an LTL formula into a TGBA and into a TGTA [6].

SDD/ITS is a library for symbolic representation of state-spaces in the form of Instantiable Transition Systems (ITS): an abstract interface for symbolic Labeled Transition Systems (LTS).

The symbolic encoding of ITS is based on Hierarchical Set Decision Diagrams (SDD) [85]. SDDs allow a compact symbolic representation of states and transition relation.

The algorithms presented in this paper can be implemented using any kind of decision diagram (such as OBDD), but use of the SDD software library allows to easily benefit from the automatic saturation mechanism described in [51].

LTSmin [12] can generate state spaces from various input formalisms (μCRL, DVE, GNA, MAPLE, PROMELA, ...) and store the obtained LTS in a concise symbolic format, called Extended Table Format (ETF). We used LTSmin to convert DVE models into ETF for our experiments. This approach offers good generality for our tool, since it can process any formalism supported by LTSmin tool.

Our symbolic model checker inputs an ETF file and an LTL formula. The LTL formula is converted into TGBA or TGTA which is then encoded using an ITS. The ETF model is also symbolically encoded using an ITS (see Section 6.4.2). The two obtained ITSs are then composed to build a symbolic product, which is also an ITS. Finally, the OWCTY emptiness check is applied to this product.

In all the approaches evaluated in this experimentation, the symbolic products are encoded using the same variables ordering: we used the *top ordering* proposed by Sebastiani et al. [79]. In top ordering, the variables that encode the property automaton (TGBA or TGTA) are at the top of the variable ordering, they precede the variables encoding the model in the decision diagram of the product.

6.4.2 Using ETF to build the transition relation of a changeset-based symbolic Kripke structure

An ETF file[3] produced by LTSmin is a text-based serialization of the symbolic representation of the transition relation of a model whose states consist in n integer variables. Transitions are described in the following tabular form:

```
0/1  0/1  *    *
```

[1] http://ddd.lip6.fr

[2] Respectively http://ddd.lip6.fr, http://spot.lip6.fr, and http://fmt.cs.utwente.nl/tools/ltsmin.

[3] http://fmt.cs.utwente.nl/tools/ltsmin/doc/etf.html

```
1/2  *    0/1  *
...
```

where each column correspond to a variable, and each line describes the effect of a symbolic transition on the corresponding variables. The notation "in/out" means that the variable must have the value "*in*" for the transition to fire, and the value is then updated to "*out*". A "*" means that the variable is not consulted or updated by the transition. Each line may consequently encode a set of explicit transitions that differ only by the values of the starred variables: the support of a transition is the set of unstarred variables.

A changeset-based symbolic Kripke structure, as defined in Section 6.3.3, can be easily obtained from such a description. To obtain a changeset associated to a line in the file, it is enough to compute the difference between values of atomic propositions associated to the *in* variables and the values associated to the *out* variables. Because they do not change, starred variables have no influence on the changeset.

Note that an empty changeset does not necessarily correspond to a line where all variables are starred. Even when *in* and *out* values are different, they may have no influence on the atomic propositions, and the resulting changeset may be empty. For instance if the only atomic proposition considered is $p = (v_1 > 1)$ (where v_1 denotes the first-column variable), then the changeset associated to the first line is $\emptyset$, and the changeset for the second line is $\{p\}$.

ETF example for the Dining philosophers model: In the dining philosophers problem, n philosophers spend their lives just thinking and easting. In the dining room, there are n chairsa around a table with a big plate of spaghetti and only n forks available. Each philosopher starts by thinking and when he gets hungry, he sits down and try to pick up the two forks that are closest to him. A philosopher can eat only if he pick up both forks. Then, when a philosopher finishes eating, he puts down the forks and returns to think.

The table below shows a part of the ETF file describing the Dining philosophers model with $n = 2$: In columns, we have the four variables of the state vector. The first line gives the variables values in the initial state and the rest of the table lists the transitions. Each line describe one ETF transition and each cell of this line indicate the change in the value of the column variable: For example in the first ETF transition, the notation "0/1" in the first column means that the first philosopher *phil*_0 changes from "0:think" to "1:one" (i.e picks up one fork); the "*" in the last column means that the value of the second philosopher *phil*_1 does not change in this transition.

	phil_0	**fork[0]**	**fork[1]**	**phil_1**
Initial state	0	0	0	0
Transitions	0/1	0/1	*	*
	1/2	*	0/1	*
		...		

phil_0 phil_1 values
0:think
1:one
2:eat
3:finish

This compressed format of transitions allowed us to easily compute the transition relation of changeset-based symbolic Kripke structure (i.e. computing the predicate $R^{\oplus}(s,k,s')$ introduced in section. 6.3.3). To illustrate the computing of the predicate $R^{\oplus}(s,k,s')$ (section. 6.3.3) in the Dining philosophers example, we assume that the set of observed atomic propositions $AP = \{p_0, p_1\}$ where $p_0 =$ "$phil\_0 = eat$" and $p_1 =$ "$phil\_1 = eat$". The changeset of the first ETF transition is equal to $\emptyset$ because *phil*_0 does not take the value "2:eat" (i.e p_0 is false) before and after the

transition, p_1 also does not change because the value of *phil*_1 does not change in this transition. Similarly, it is easy to compute the changeset of the second ETF transition equal to $\{p_0\}$.

6.4.3 Benchmark

BEEM model	states $10^3\times$	stut. ratio	BEEM model	states $10^3\times$	stut. ratio
at.5	31 999	96%	lann.6	144 151	44%
bakery.4	157	76%	lann.7	160 025	52%
bakery.7	29 047	80%	lifts.7	5 126	85%
bopdp.3	1 040	86%	lifts.8	12 359	85%
brp2.3	40	73%	mcs.5	60 556	81%
elevator.4	888	63%	peterson.5	131 064	73%
fischer.5	101 028	82%	pgm_protocol.8	3 069	88%
iprotocol.7	59 794	81%	phils.7	71 934	89%
lamport_nonatomic.5	95 118	86%	production_cell.6	14 520	78%
lamport.7	38 717	91%	reader_writer.3	604	81%

Table 6.2: Characteristics of our selected benchmark models. The stuttering-ratio represents the percentage of stuttering transitions in the model. Since the definition of stuttering depends on the atomic propositions of the formula, we give an average over the 200 properties checked against each model.

We evaluated the TGBA and TGTA approaches on the following models and formulas:

- Our models come from the BEEM benchmark [67], a suite of models for explicit model checking, which contains some models that are considered difficult for symbolic model checkers [12]. Table 6.2 summarizes the 20 models we used to evaluate our approaches. These models represent various controllers, communication protocols, mutual exclusion and leader election algorithms.

- BEEM provides a few LTL formulas, but they mostly represent safety properties and can thus be checked without building a product. Therefore, for each model, we randomly generated 200 stutter-invariant LTL formulas: 100 verified formulas (empty product) and 100 violated formulas (non-empty product). We consequently have a total 4000 pairs of (model, formula).

All tests were run on a 64bit Linux system running on an Intel(R) 64-bit Xeon(R) @2.00GHz with 64GB of RAM. Executions that exceeded 60 minutes or 6GB of RAM were aborted and are reported with time and memory above these thresholds in our graphics.

6.4.4 Results

The results of our experimental comparisons are presented by the two scatter plot matrices of Figure 6.2 and Figure 6.3.

Figure 6.2 compares the time-performance of the TGTA-approach against the reference TGBA approach. In order to show the influence of the saturation technique, we also ran the TGBA

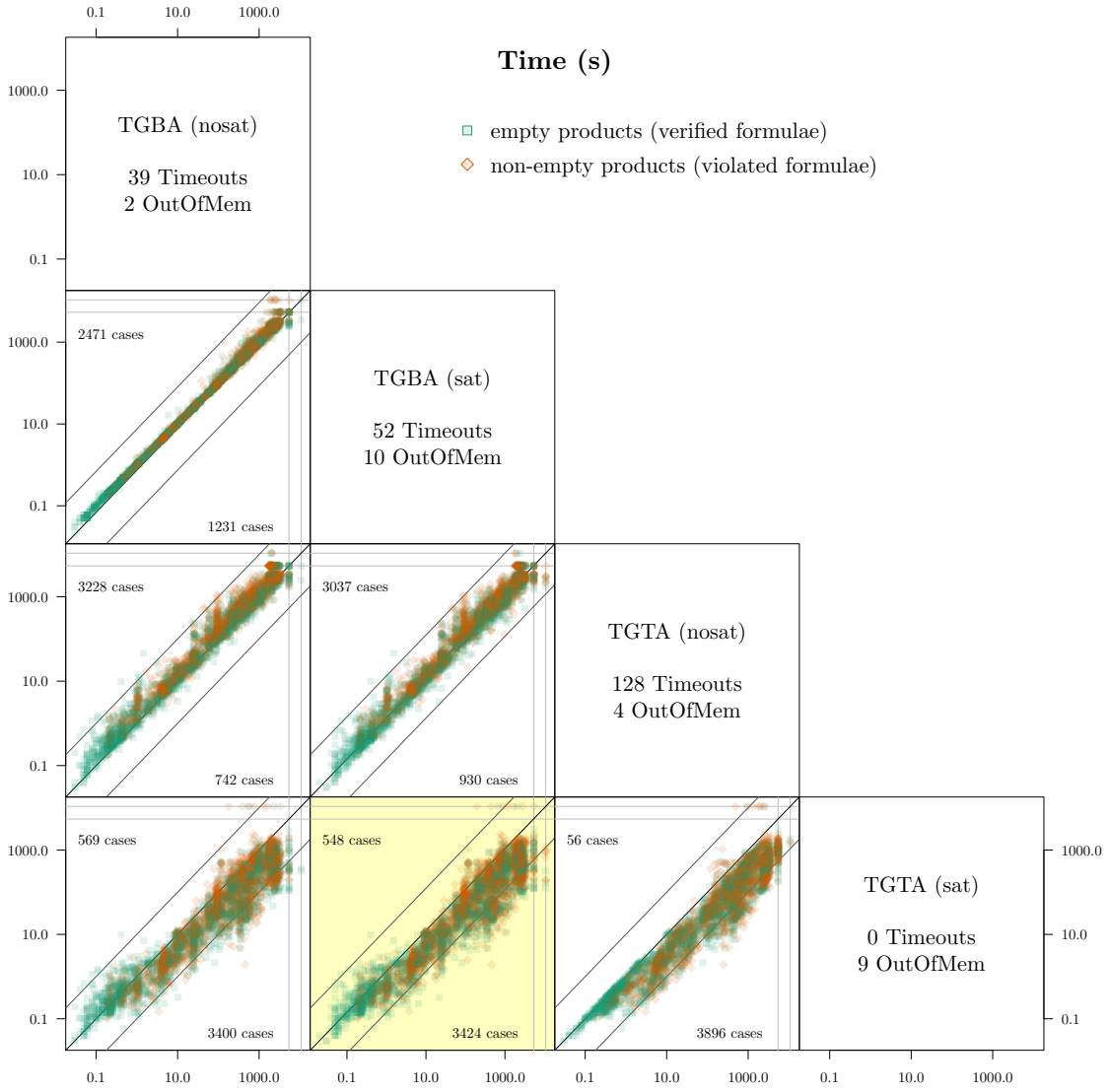


Figure 6.2: Time-comparison of the TGBA and TGTA approaches, with saturation enabled "(sat)" or disabled "(nosat)", on a set of 4000 pairs (*model*,*formula*). Timeouts and Out-of-memory errors are plotted on separate lines on the top or right edges of the scatter plots. Each plot also displays the number of cases that are above or below the main diagonal (including timeouts and out-of-memory errors), i.e., the number of (*model*,*formula*) for which one approach was better than the other. Additional diagonals show the location of $\times 10$ and $/10$ ratios. Points are plotted with transparency to better highlight dense areas, and lessen the importance of outliers.

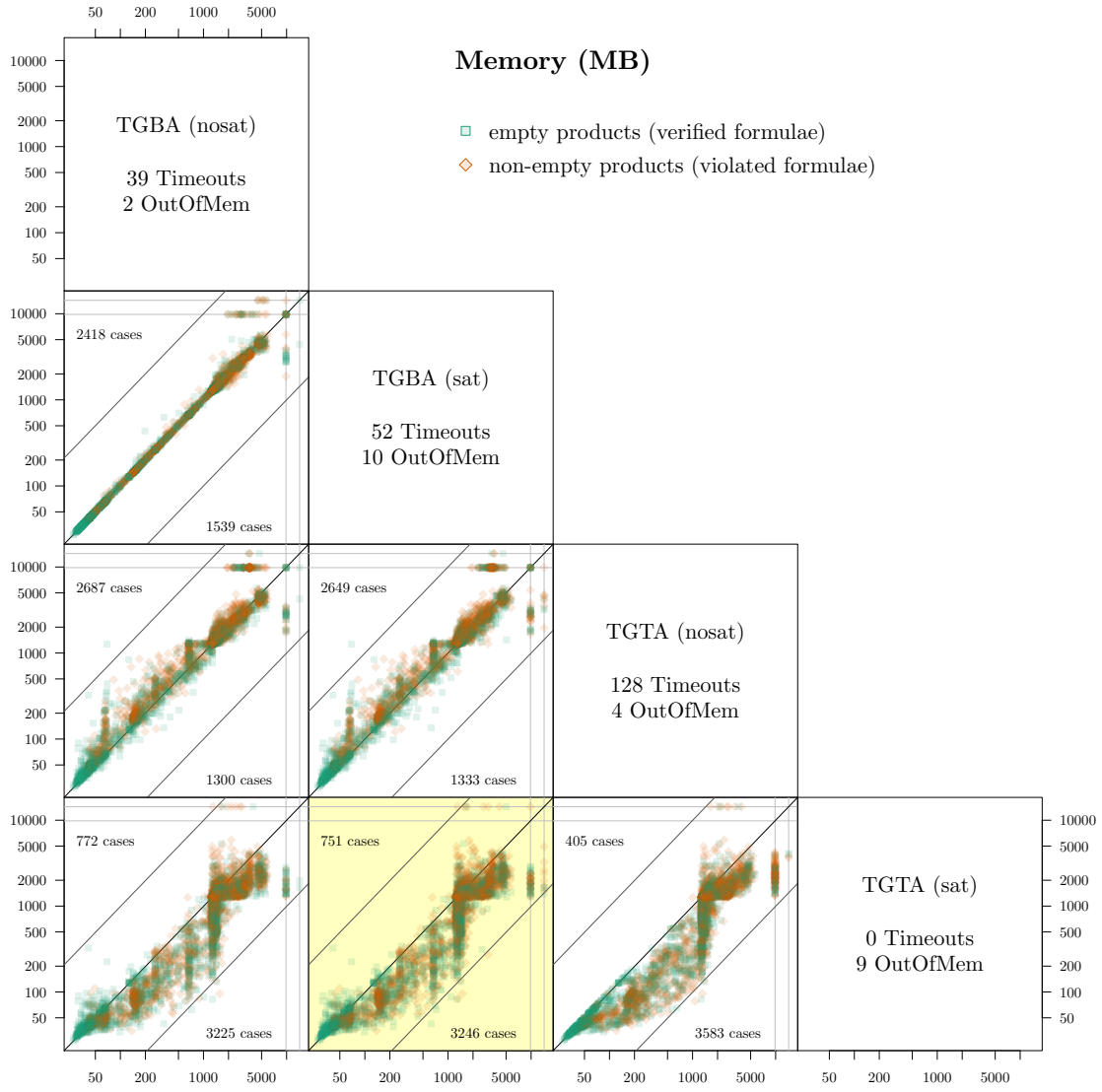
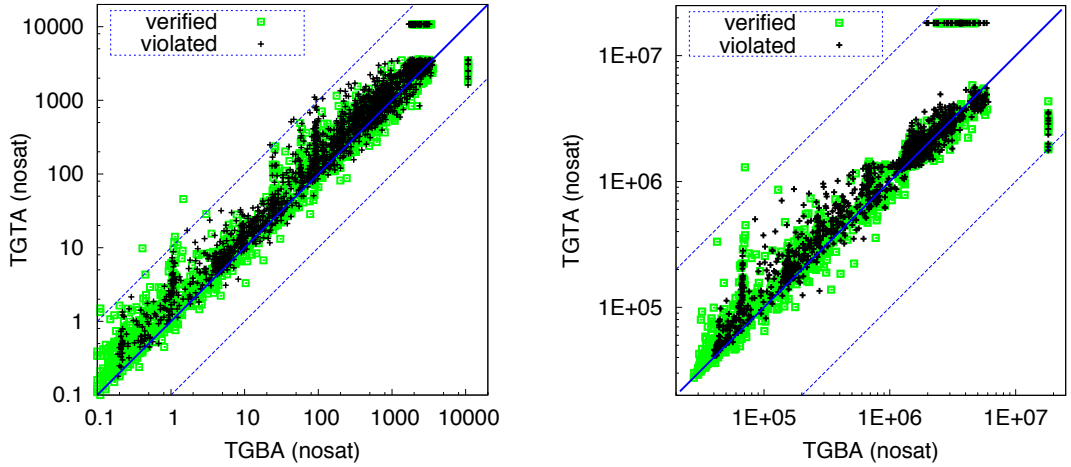
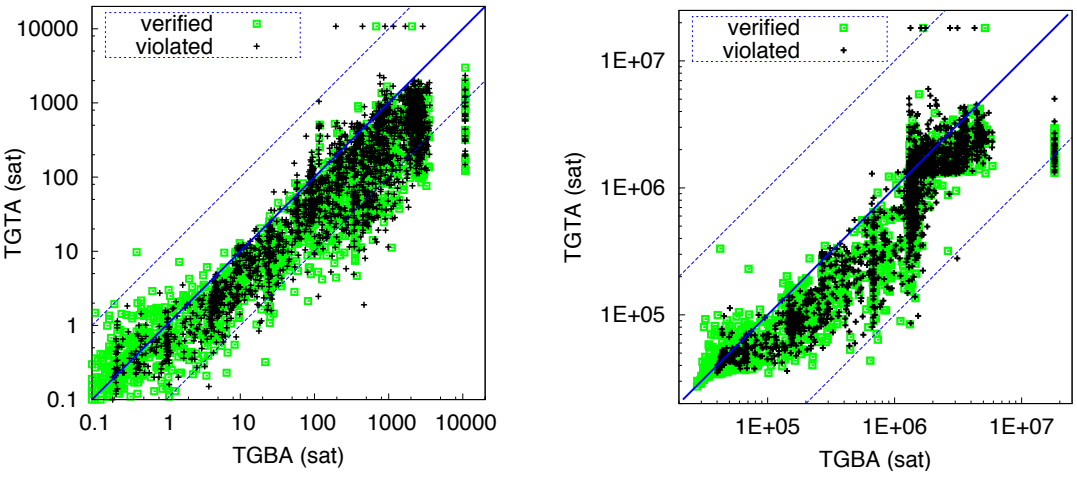


Figure 6.3: Comparison of the memory-consumption of the TGBA and TGTA approaches, with or without saturation, on the same set of problems.



(a) Time (b) Memory

Figure 6.4: Comparison of TGBA and TGTA approaches, without saturation "(nosat)". Timeouts and Out-of-memory errors are plotted on the top or right edges of the scatter plots.



(a) Time (b) Memory

Figure 6.5: Comparison of TGBA and TGTA approaches, with saturation enabled "(sat)". Timeouts and Out-of-memory errors are plotted on the top or right edges of the scatter plots.

		time (s)				memory (MB)			
		TGBA		TGTA		TGBA		TGTA	
	cases	(nosat)	(sat)	(nosat)	(sat)	(nosat)	(sat)	(nosat)	(sat)
at.5	200	391.86	446.56	590.44	182.95	1668	1649	1760	1376
bakery.4	200	14.07	14.74	17.90	7.43	331	341	388	197
bakery.7	192	634.79	632.04	593.24	168.80	1353	1361	1330	1160
bopdp.3	200	18.89	19.36	28.65	9.44	325	332	392	214
brp2.3	200	5.05	5.20	6.31	2.65	147	149	170	90
elevator.4	199	154.02	169.32	204.15	81.40	1142	1144	1165	969
fischer.5	190	300.82	372.75	418.96	187.41	1455	1486	1496	1590
iprotocol.7	145	1790.41	1776.72	1987.39	290.30	2271	2302	2148	1297
lamport_nonatomic.5	200	111.24	147.24	191.97	17.00	917	935	1108	380
lamport.7	196	479.56	570.71	696.13	63.61	1523	1547	1620	838
lann.6	129	504.95	576.05	793.42	148.53	1073	1048	1021	631
lann.7	196	161.05	208.84	379.38	99.07	944	960	1027	901
lifts.7	200	541.09	579.36	643.38	152.58	1302	1318	1349	951
lifts.8	173	1804.29	1858.65	1910.90	283.42	1903	1891	1919	1151
mcs.5	187	1659.96	1889.17	2208.65	460.33	4155	3890	3718	2085
peterson.5	199	196.12	283.39	309.68	55.40	1105	1156	1145	789
pgm_protocol.8	194	935.45	982.70	1118.97	658.58	1419	1423	1437	1336
phils.7	200	6.79	12.33	15.58	1.64	181	200	330	74
production_cell.6	197	663.13	702.71	862.59	375.44	1367	1388	1515	1457
reader_writer.3	200	0.67	0.71	1.12	0.63	52	53	63	48
all models	3797	487.65	530.94	612.79	158.04	1204	1201	1232	866

Table 6.3: Comparison of the performances of the symbolic approaches TGBA versus TGTA , presented model by model. The average run-time and memory consumption is computed over the 3797 cases where all methods terminated normally (without timeout or out-of-memory error).

and TGTA approaches with saturation disabled. In our comparison matrix, the labels "(sat)" and "(nosat)" indicate whether saturation was enabled or not. Each point of the scatter plots represents a measurement for a pair $(model, formula)$. Axes use a logarithmic scale. The colors distinguish violated formulas (non-empty products) from verified formulas (empty products).

In each scatter plot, each point below the diagonal is in favor of the approach displayed on the right, while each point above the diagonal is in favor of the approach displayed in the top of the scatter plot. For instance, the highlighted scatter plot (bottom of Figure 6.2) compares the time-performance between TGBA and TGTA approaches where the saturation is enabled (sat), the x-axis represents the "TGBA (sat)" approach and the y-axis represents the "TGTA (sat)" approach, so 3424 points below the diagonal correspond to cases where the "TGTA (sat)" approach is better, and the 548 points above the diagonal corresponds to points were the "TGBA (sat)" approach is better.

Figure 6.3 gives the memory view of this experiment.

As shown by the highlighted scatter plots in Figure 6.2 and 6.3, the TGTA approach clearly outperforms the traditional TGBA-based scenario by one order of magnitude. This is due to the combination of two factors: saturation and exploration of stuttering.

The saturation technique does not significantly improve the model checking using TGBA (compare "TGBA (sat)" against "TGBA (nosat)" at the top of Figure 6.2 and 6.3). In fact, the saturation technique is limited on the TGBA approach, because in the transition relation of Definition 34 each conjunction must consult the variable q representing the state of the TGBA, therefore q impacts the supports and the reordering of clusters evaluated by the saturation. This situation is different in the case of TGTA approach, where the $T_\emptyset$ term of the transition-relation of the product (equation (6.4)) does not involve the state q of the TGTA: here, saturation strongly improves performances (compare "TGTA (sat)" against "TGTA (nosat)").

Overall the improvement to this symbolic technique was only made possible because the TGTA representation makes it easy to process the stuttering behaviors separately from the rest. These stuttering transitions represent a large part of the models transitions, as shown by the stuttering-ratios of Table 6.2. Using these stuttering-ratios, we can estimate in our Benchmark the importance of the term $T_\emptyset$ compared to T_* in equation (6.4).

Figures 6.4 and 6.5 show more clearly the most important scatter plots of the two matrices. Figure 6.4 shows the scatter plots (for time and memory performance) comparing TGBA and TGTA approaches with saturation disabled (nosat). Figure 6.5 displays the same comparison but with saturation enabled (sat).

Table 6.3 gives an overview of the performance of the TGBA and TGTA approach, model by model. The average run-time and memory consumption is computed over the 3797 cases where all methods terminated normally (without timeout or out-of-memory error).

Table 6.4 shows the best and the worst approach among the four possible combinations, i.e., TGBA and TGTA approaches with saturation disabled (no saturation) or with saturation enabled. The verified formulas are separated from violated formulas.

The Tables 6.3 and 6.4 confirm the above observations deduced from the scatter plots: i.e., the TGTA approach with saturation outperform the TGBA approach (with or without saturation).

		no saturation		saturation	
		TGBA	TGTA	TGBA	TGTA
Verified	Win	297 (14%)	13 (0%)	222 (11%)	1613 (81%)
	Lose	204 (10%)	1359 (68%)	464 (23%)	67 (3%)
	Fail	29 (1%)	43 (2%)	30 (1%)	2 (0%)
Violated	Win	150 (7%)	1 (0%)	97 (4%)	1795 (89%)
	Lose	118 (5%)	1594 (79%)	297 (14%)	26 (1%)
	Fail	12 (0%)	89 (4%)	32 (1%)	7 (0%)

Table 6.4: On all experiments (grouped by verified versus violated formulas), we count the number of cases a specific method has (Win) the best time or (Lose) it has either run out of time/memory or it has the worst time amongst successful methods. The Fail line shows how much of the Lost cases failed (because of timeout or out-of-memory). The sum of a line may exceed 100% if several methods are equally placed.

6.5 Conclusion

Testing automata are a way to improve the explicit model checking approach when verifying stutter-invariant properties, but they had not been used for symbolic model checking.

In the previous chapters, we introduced a generalization of these testing automata, called TGTA, and we evaluated their use for explicit model checking.

In this chapter, we have shown how to use TGTA in a symbolic approach. We compare this new TGTA approach to a more classical symbolic approach (using TGBA).

On our benchmark, using TGTA and saturation technique, we were able to gain one order of magnitude over the TGBA-based approach.

Three versions of this TGTA approach, including a basic version and two improvements were implemented. The first improvement was to use a new encoding of a Kripke structure inspired by the TGTA transition relation, i.e., based on the labeling of transitions by changesets. The second improvement is based on the exploration of stuttering transitions during the emptiness check of the symbolic product.

The latter optimization is based on the property of TGTA that all stuttering transitions are self-loops and every state has a stuttering self-loop. Consequently, the exploration of stuttering transitions in the product is equivalent to explore stuttering transitions in the model (remaining in the same TGTA states).

Using this property, we have shown that fixpoints over the transition relation of a product between a Kripke structure and a TGTA can benefit from the saturation technique, especially because part of their expression is only dependent on the model, and can be evaluated without consulting the transition relation of the property automaton. This allows to the saturation algorithm to ignore the symbolic variables encoding the TGTA in the product, and therefore effectively saturate the symbolic product nodes of the variables encoding the model.

This improvement was possible only because TGTA makes it possible to process stuttering behaviors specifically, in a way that helps the saturation technique.

In the next chapter, we evaluate the use of TGTA in the context of hybrid approaches, which

combines the use of both explicit and symbolic approaches [79, 38]. These hybrid approaches use an On-the-Fly exploration unlike the symbolic approaches presented in this chapter.

CHAPTER 7

Hybrid LTL Model Checking using TGTA

Contents

7.1 Introduction

In the previous chapter, we have shown that the symbolic approach to model checking allows to encode the product automaton in a concise way, but its emptiness check can not be performed on-the-fly as in the explicit approach.

In order to take advantage of the best of the both worlds, *Hybrid approaches* [79, 38] to model checking are proposed as combinations of explicit and symbolic approaches.

In these hybrid approaches, the property automaton is described explicilty because its size is not large in most cases (and can be reduced by means of several explicit optimizations). However, the state-space of the model is typically very large and must be encoded symbolically. An hybrid approach is generally based on an on-the-fly construction of an explicit graph of symbolic nodes, called aggregates. Each aggregate symbolically encodes a set of states of the Kripke structure or of the product. In this work, we focused on three hybrid techniques [37]: SOG, SOP and SLAP. The Symbolic Observation Graph (SOG) approach exploits the fact that only a subset of the atomic propositions of the model are observed by the LTL property to check. A SOG is an abstraction of a Kripke structure where consecutive states are aggregated if they share the same values of the observed atomic propositions. The *Symbolic Observation Product* (SOP) approach tries to allow further aggregation than SOG by exploiting the fact that the number of observed atomic propositions decreases as we progress in the property automaton. A *Self-Loop Aggregation Product* (SLAP) is similar to a SOP, an aggregation graph alternative to the traditional product automaton. In SLAP, the Kripke structure states are aggregated according to the valuations of the self-loops in the property automaton.

As for symbolic model checking, Testing Automata (and their variants) have never been used before for hybrid model checking.

The goal of this chapter is to show how the traditional hybrid approaches based on TGBA can be adapted to obtain TGTA-based approaches.

In this chapter, we present three existing hybrid approaches SOG, SOP and SLAP that are based on TGBA. Then, we define and implement variations of these three approaches using TGTA instead of TGBA. We then experimentally compare the performance of each hybrid approach (SOG, SOP and SLAP) against its TGTA-based variant (SOG-TGTA, SOP-TGTA and SLAP-TGTA).

7.2 Preliminaries

The formalization of hybrid approaches requires the following definitions and notations introduced by Duret-Lutz et al. [38].

We firstly remind the formalization of propositional formula , then we present alternative definitions of TGBA Duret-Lutz et al. [38] and TGTA that use these propositional formulas as labels.

- $\mathbb{B} = \{\bot, \top\}$ represents the Boolean values.
- $\mathbb{B}(AP)$ is the set of all propositional formulas over AP. The formulas of $\mathbb{B}(AP)$ are built inductively from the propositions AP, $\mathbb{B}$, and the logical operators $\wedge$, $\vee$, and $\neg$.
- Using this definition of $\mathbb{B}(AP)$, $AP' \subseteq AP$ implies that $\mathbb{B}(AP') \subseteq \mathbb{B}(AP)$.
- $FV(\phi)$ (Free Variables) Duret-Lutz et al. [38] is the set of atomic propositions observed in the formula ϕ, e.g., $FV(ab \vee a\bar{b}) = \{a\}$ because $\phi = (ab \vee a\bar{b})$ can be simplified into $\phi = a$ (we say that b is a silent (or a "don't care") atomic proposition in ϕ).

- The notation $\ell \overset{AP'}{=} \ell'$ is equivalent to $\ell_{|AP'} = \ell'_{|AP'}$, where $\ell_{|AP'}$ denotes the restriction of the valuation ℓ to the subset of atomic propositions AP'. In other words, $\ell \overset{AP'}{=} \ell'$ means that the valuations ℓ and ℓ' match on the atomic propositions of AP'.

Duret-Lutz et al. [38] use an alternative definition of TGBA more suited than Definition 16 to define and implement the hybrid approaches SOP and SLAP. In these TGBA, each transition is labeled with a propositional formula ϕ over AP (the resulting transition relation is of the form $\delta \subseteq Q \times \mathbb{B}(AP) \times 2^{\mathcal{F}} \times Q$). In the following, we will see that this labeling of transitions with propositional formula simplifies the formalization of the different hybrid approaches presented in this work. For instance, in SOP and SOP-TGTA approaches, it allows to simplify the definition (and the implementation) of the concept of observed alphabet from a state of an automaton (TGBA or TGTA).

7.2.1 TGBA labeled with propositional formulas

In a TGBA labeled with propositional formulas, each transition is labeled with a propositional formula $\phi \in \mathbb{B}(AP)$ instead of a valuation $\ell \in 2^{AP}$. This formulas ϕ represents the maximal set of valuations $\{\ell_0, \ell_1, \ldots, \ell_n\} \subseteq 2^{AP}$ such that $\forall i \leq n$, $\ell_i \models \phi$. In other words, the set of valuations $\{\ell_0, \ell_1, \ldots, \ell_n\}$ is the set of all models of ϕ (it can be represented by $\phi = \bigvee_{i \leq n} \ell_i$)). For example, in Figure 7.1a, the formula $\phi = b$ labeling the transition $q_0 \xrightarrow{b} q_1$ represents the set of valuations $\{\bar{a}b, ab\}$.

In order to obtain this form of TGBA, we merge into a single transition all transitions between each pair of states (q, q'), when these transitions are labelled with the same acceptance conditions F. Formally, for a pair of states $(q, q') \in Q^2$ and a set of acceptance conditions $F \in \mathcal{F}$, the set of transitions $(q, \ell_0, F, q'), \ldots, (q, \ell_n, F, q') \in \delta$ are merged into a single transition (q, ϕ, F, q') where $\phi = \bigvee_{i \leq n} \ell_i$.

Definition 39 (TGBA (labeled with propositional formulas))**.** *A Transition-based Generalized Büchi Automata (TGBA) over the alphabet $\Sigma = 2^{AP}$ is a tuple $\mathcal{G} = \langle Q, I, \delta, \mathcal{F} \rangle$ where*

- *Q is a finite set of states,*
- *I is a finite set of initial states,*
- *$\mathcal{F} \neq \emptyset$ is a finite and non-empty set of acceptance conditions,*
- *$\delta \subseteq Q \times \mathbb{B}(AP) \times 2^{\mathcal{F}} \times Q$ is a transition relation, where each element $(q, \phi, F, q') \in \delta$ represents a transition from state q to state q' labeled by a propositional formula $\phi \in \mathbb{B}(AP)$, and a set of acceptance conditions $F \in 2^{\mathcal{F}}$. In the following, an element $(q, \phi, F, q') \in \delta$ will be denoted $q \xrightarrow{\phi, F} q'$*

An infinite word $\sigma = \ell_0 \ell_1 \ell_2 \ldots \in \Sigma^{\omega}$ is accepted by $\mathcal{G}$ if there exists an infinite sequence of transitions $\pi = (q_0, \phi_0, F_0, q_1)(q_1, \phi_1, F_1, q_2) \cdots (q_i, \phi_i, F_i, q_{i+1}) \cdots \in \delta^{\omega}$ (π is called a run of $\mathcal{G}$) where:

- *$q_0 \in I$, and $\forall i \in \mathbb{N}$, $\ell_i \models \phi_i$ (i.e., the infinite word σ is recognized by the run π)*

- $\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$ *(i.e., the run* π *is accepting* iff *it visits each acceptance condition infinitely often).*

The language accepted by $\mathcal{G}$ *is the set* $\mathscr{L}(\mathcal{G}) \subseteq \Sigma^{\omega}$ *of infinite words it accepts.*

Figure 7.1a shows the TGBA of the LTL formula $a \mathsf{U} b$. The transitions are labeled with the propositional formulas $a\bar{b}$, b and $\top$ (which encodes all the valuations over $AP = \{a, b\}$).

Using this variant of TGBA, we obtain a new definition of the synchronous product between a TGBA $\mathcal{G}$ and a Kripke structure $\mathcal{K}$. We remind that this synchronous product is also a TGBA, which only accepts the words accepted by both $\mathcal{G}$ and $\mathcal{K}$ (Formally, $\mathscr{L}(\mathcal{K} \otimes \mathcal{G}) = \mathscr{L}(\mathcal{K}) \cap \mathscr{L}(\mathcal{G})$).

Definition 40 (Synchronous product of a TGBA with a Kripke structure)**.** *For a TGBA* $\mathcal{G} = \langle Q, I, \delta, \mathcal{F} \rangle$ *over the alphabet* $\Sigma = 2^{AP}$ *and a Kripke structure* $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$*, the product* $\mathcal{G} \otimes \mathcal{K}$ *is the TGBA* $\langle Q_{\otimes}, I_{\otimes}, \delta_{\otimes}, \mathcal{F} \rangle$ *over the alphabet* $\Sigma = 2^{AP}$ *where*

- $Q_{\otimes} = Q \times \mathcal{S}$,
- $I_{\otimes} = I \times \mathcal{S}_0$,
- $\delta_{\otimes} \subseteq Q_{\otimes} \times \mathbb{B}(AP) \times 2^{\mathcal{F}} \times Q_{\otimes}$ *where*

$$\delta_{\otimes} = \left\{ (q_1, s_1) \xrightarrow{l(s_1), F} (q_2, s_2) \;\middle|\; \begin{array}{l} (s_1, s_2) \in \mathcal{R} \text{ and} \\ \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi, F} q_2 \in \delta \text{ and } l(s_1) \models \phi \end{array} \right\}$$

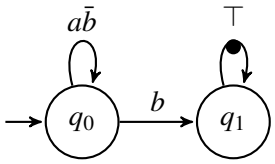
Figure 7.1c (taken from [37]) is an illustration of Definition 40. It shows an example of a synchronous product $\mathcal{G} \otimes \mathcal{K}$ between a TGBA $\mathcal{G}$ of $a \mathsf{U} b$ (Figure 7.1a) and an example of Kripke structure $\mathcal{K}$ over $AP' = \{a, b, c\}$ shown in Figure 7.1b. The initial state of the product is (q_0, s_0). Then, the successors $\{s_1, s_4\}$ of s_0 in $\mathcal{K}$ are synchronized with the state q_0 of $\mathcal{G}$, because the TGBA self-loop $q_0 \xrightarrow{a\bar{b}} q_0$ is labeled by the formula $\phi = a\bar{b}$ and $l(s_0) = a\bar{b}c \models \phi$. From state (q_0, s_4), the product move to state (q_1, s_5) through the TGBA transition $q_0 \xrightarrow{b} q_1$ because $l(s_4) = ab\bar{c} \models b$. From the product state (q_1, s_5), the TGBA state q_1 only requires to verify $\top$ (i.e, any valuation) to explore the self-loop labeled with the acceptance condition $\bullet$. Therefore, any cycle of $\mathcal{K}$ starting in s_5 corresponds to an accepting cycle in the product.

7.2.2 TGTA labeled with propositional formulas

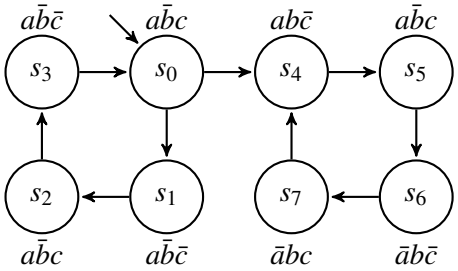
Similar to TGBA, we present in this section a definition of a TGTA labeled with propositional formulas, and for the same reasons as for TGBA, we will see that this alternative definition of TGTA is more suited than Definition 30 to define and implement the TGTA-based hybrid approaches presented later in this chapter.

The difference between TGBA and TGTA is mainly in the interpretation of the transition relation $\delta \subseteq Q \times \mathbb{B}(AP) \times 2^{\mathcal{F}} \times Q$. Indeed, in TGTA labeled with propositional formulas, for each transition $(q, \phi, F, q') \in \delta$, the formula ϕ encodes a set of changesets $\{k_0, k_1, \ldots, k_n\} \subseteq 2^{AP}$, where $\forall i \leq n$, each changeset $k_i \models \phi$.

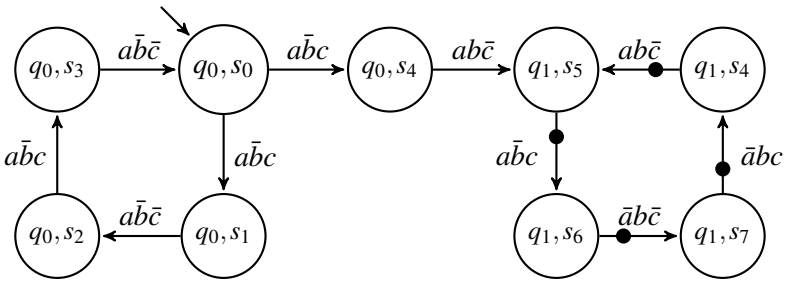
As for TGBA, in order to obtain the TGTA labeled with propositional formulas, we merge into a single transition all transitions between each pair of states (q, q'), when these transitions are



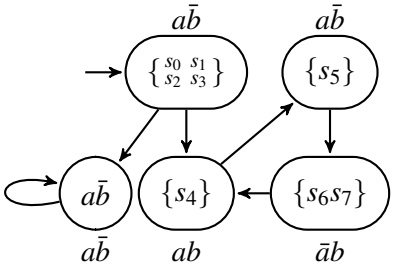
(a) TGBA $\mathcal{G}$ for $a \cup b$



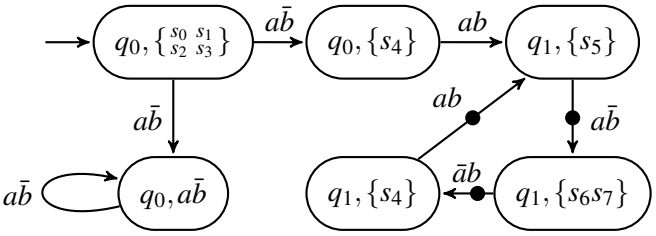
(b) Kripke structure $\mathcal{K}$



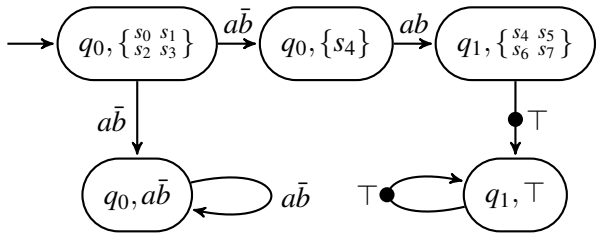
(c) TGBA of product $\mathcal{G} \otimes \mathcal{K}$



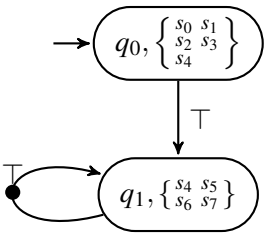
(d) Aggregated Kripke structure SOG $\widehat{\mathcal{K}}_{\{a,b\}}$



(e) TGBA of the product $\mathcal{G} \otimes \widehat{\mathcal{K}}_{\{a,b\}}$

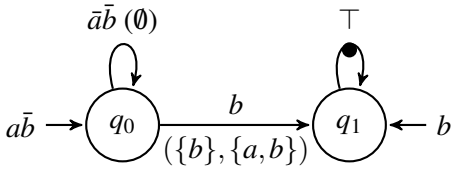


(f) TGBA of the SOP $\mathcal{G} \widehat{\otimes} \mathcal{K}$

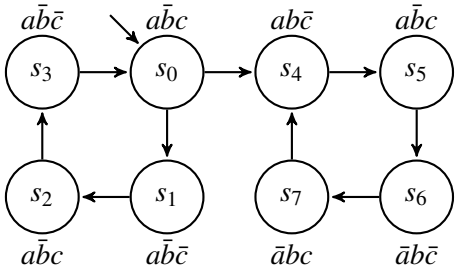


(g) TGBA of the SLAP $\mathcal{G} \boxtimes \mathcal{K}$

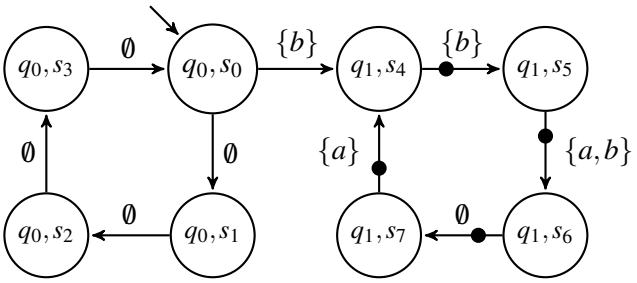
Figure 7.1: Examples [37] Using TGBA



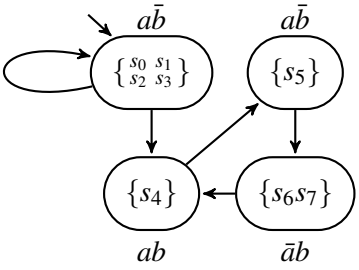
(a) TGTA $\mathcal{T}$ for $a \cup b$. The formula $\bar{a}\bar{b}$ labeling the self-loop on q_0 represents the empty changeset $\emptyset$. The formula b labeling the transition $q_0 \xrightarrow{b} q_1$ represents the set of changesets $\{\{b\},\{a,b\}\}$. The label $\top$ of the self-loop on q_1 encodes the set of all changesets $\{\emptyset,\{a\},\{b\},\{a,b\}\}$.



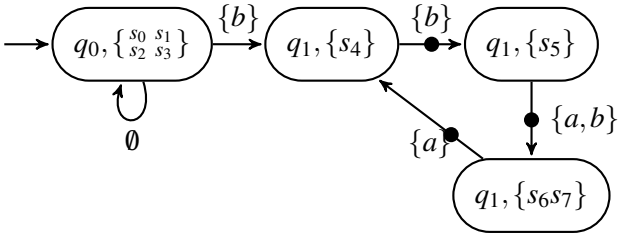
(b) Kripke structure $\mathcal{K}$



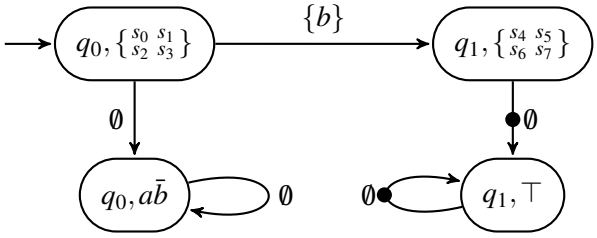
(c) $\emptyset$-TGTA of product $\mathcal{T} \otimes \mathcal{K}$



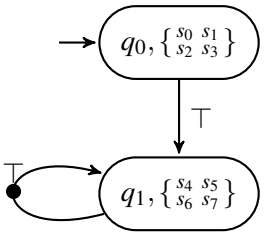
(d) Symbolic Observation Graph for TGTA: a SOG-TGTA $\widehat{\mathcal{K}}'_{\{a,b\}}$



(e) $\emptyset$-TGTA of the product $\mathcal{T} \otimes \widehat{\mathcal{K}}'_{\{a,b\}}$



(f) $\emptyset$-TGTA of the SOP $\mathcal{T} \widehat{\otimes} \mathcal{K}$



(g) $\emptyset$-TGTA of the SLAP $\mathcal{T} \boxtimes \mathcal{K}$

Figure 7.2: Examples Using TGTA

labelled with the same acceptance conditions F. Formally, , for a pair of states $(q,q') \in Q^2$ and a set of acceptance conditions $F \in \mathcal{F}$, the set of transitions $(q,k_0,F,q'),\ldots,(q,k_n,F,q') \in \delta$ are merged into a single transition (q,ϕ,F,q') where the set of changesets $\{k_0,\ldots,k_n\}$ is the set of all models of ϕ. In the following, we will use (q,ϕ,F,q') and $(q,\{k_0,\ldots,k_n\},F,q')$ interchangeably as transitions of δ. For example, in Figure 7.2a, the formula $\phi = b$ labeling the transition $q_0 \xrightarrow{b} q_1$ represents the set of changesets $\{\{b\},\{a,b\}\}$ (meaning that the value of b changes between q_0 and q_1 and we "do not care" about a).

The following definition formalizes this form of TGTA and how it changes the way an infinite word is accepted.

Definition 41 (TGTA (labeled with propositional formulas))**.** *A Transition-based Generalized Testing Automata (TGTA) over the alphabet $\Sigma = 2^{AP}$ is a tuple $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$ where*

- *Q is a finite set of states,*
- *I is a finite set of initial states,*
- *$U : I \to \mathbb{B}(AP)$ is a function mapping each initial state to a propositional formula $\phi \in \mathbb{B}(AP)$,*
- *$\mathcal{F} \neq \emptyset$ is a finite and non-empty set of acceptance conditions,*
- *$\delta \subseteq Q \times \mathbb{B}(AP) \times 2^{\mathcal{F}} \times Q$ is a transition relation, where each element $(q,\phi,F,q') \in \delta$ represents a transition from state q to state q' labeled by a propositional formula $\phi \in \mathbb{B}(AP)$, and a set of acceptance conditions $F \in 2^{\mathcal{F}}$.*
- *δ has to satisfy the following* stuttering-normalization constraint*:*
  $\forall (q,q') \in Q^2 : \big(\exists (\phi,F) \in \mathbb{B}(AP) \times 2^{\mathcal{F}}, \emptyset \models \phi \wedge (q,\phi,F,q') \in \delta\big) \iff (q = q')$

An infinite word $\sigma = \ell_0\ell_1\ell_2\ldots \in \Sigma^\omega$ is accepted by $\mathcal{T}$ iff *there exists an infinite sequence of transitions $\pi = (q_0,\phi_0,F_0,q_1)(q_1,\phi_1,F_1,q_2)\cdots(q_i,\phi_i,F_i,q_{i+1})\cdots \in \delta^\omega$ (π is called a run of $\mathcal{T}$) where:*

- *$q_0 \in I$, and $\ell_0 \models U(q_0)$*
- *$\forall i \in \mathbb{N}, (\ell_i \oplus \ell_{i+1}) \models \phi_i$ (i.e., the infinite word σ is recognized by the run π)*
- *$\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$ (i.e., the run π is accepting).*

The language accepted by $\mathcal{T}$ is the set $\mathscr{L}(\mathcal{T}) \subseteq \Sigma^\omega$ of infinite words it accepts.

Figure 7.2a shows the TGTA of the LTL formula $a \mathbin{U} b$. The valuations of the initial states are $U(q_0) = a\bar{b}$ and $U(q_1) = b = \{ab, \bar{a}b\}$. The transitions are labeled with the propositional formulas. The formula $\bar{a}\bar{b}$ labeling the stuttering self-loop on q_0 is the empty changeset $\emptyset$. The formula b labeling the transition $q_0 \xrightarrow{b} q_1$ is obtained by merging the two changesets of the transitions $q_0 \xrightarrow{\{b\}} q_1$ and $q_0 \xrightarrow{\{a,b\}} q_1$ (because $\{\{b\},\{a,b\}\} \models b$). On the self-loop $q_1 \xrightarrow{\top,\{\bullet\}} q_1$, the formula $\top$ is obtained by merging the set of all changesets over $AP = \{a,b\}$, and the acceptance condition is indicated by the black dot ($\mathcal{F} = \{\bullet\}$).

Using this alternative definition of TGTA, we obtain the following definition of the synchronous product between a TGTA and a Kripke structure. We remind that this synchronous product is an $\emptyset$-TGTA (Definition 29).

Definition 42 (Synchronous product of a TGTA with a Kripke structure)**.** *For a TGTA* $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$ *over the alphabet* $\Sigma = 2^{AP}$ *and a Kripke structure* $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$*, the product* $\mathcal{T} \otimes \mathcal{K}$ *is the* $\emptyset$*-TGTA* $\langle Q_\otimes, I_\otimes, U_\otimes, \delta_\otimes, \mathcal{F} \rangle$ *over the alphabet* $\Sigma = 2^{AP}$ *where*

- $Q_\otimes = Q \times \mathcal{S}$,
- $I_\otimes = \{(q,s) \in I \times \mathcal{S}_0 \mid l(s) \models U(q)\}$
- $\forall (q,s) \in I_\otimes, U_\otimes((q,s)) = l(s)$,
- $\delta_\otimes \subseteq Q_\otimes \times \mathbb{B}(AP) \times 2^{\mathcal{F}} \times Q_\otimes$ *where*

$$\delta_\otimes = \left\{ (q_1,s_1) \xrightarrow{(l(s_1) \oplus l(s_2))_{|AP}, F} (q_2,s_2) \;\middle|\; \begin{array}{l} (s_1,s_2) \in \mathcal{R} \text{ and} \\ \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi, F} q_2 \in \delta \text{ and} \\ (l(s_1) \oplus l(s_2)) \models \phi \end{array} \right\}$$

Figure 7.2c is an illustration of Definition 42. It shows an example of a synchronous product $\mathcal{T} \otimes \mathcal{K}$ between the TGTA $\mathcal{T}$ of $a \mathsf{U} b$ of Figure 7.2a and the Kripke structure $\mathcal{K}$ of Figure 7.1b. The initial state of the product is (q_0,s_0) because $l(s_0) = a\bar{b}c \models U(q_0) = a\bar{b}$. Then, (q_0,s_0) have two successors: the first successor is (q_0,s_1) because $\mathcal{K}$ has a transition $s_0 \to s_1$ with $l(s_0) \oplus l(s_1) = \emptyset \models \bar{a}\bar{b}$ and $\mathcal{T}$ have a stuttering self-loop $q_0 \xrightarrow{\bar{a}\bar{b}} q_0$; the second successor is (q_1,s_4) because in $\mathcal{K}$ we have $s_0 \to s_4$ with $l(s_0) \oplus l(s_4) = \{b,c\} \models b$ and the TGTA have a transition $q_0 \xrightarrow{b} q_1$ labeled with $\phi = b$. From the product state (q_1,s_4), the TGTA can explore any changeset through the self-loop labeled with $\top$ and the acceptance condition $\bullet$. Therefore, the TGTA state q_1 can be synchronized with any reachable state from s_4 in $\mathcal{K}$ and the cycle $(q_1,s_4) \to (q_1,s_5) \to (q_1,s_6) \to (q_1,s_7) \to (q_1,s_4)$ is an accepting cycle in the product. In the obtained product $\mathcal{T} \otimes \mathcal{K}$, each transition is labeled with the changeset $((l(s_1) \oplus l(s_2))_{|AP})$ between the states of $\mathcal{K}$. These changesets are computed according to the set of atomic propositions $AP = \{a,b\}$ observed by $\mathcal{T}$. The product transitions also bear the acceptance conditions coming from the TGTA $\mathcal{T}$.

7.3 Symbolic Observation Graph (SOG)

In this section, we propose an adaptation of the SOG hybrid approach for use with TGTA instead of TGBA. We start by presenting a variant of SOG [58, 37] used in a TGBA-based approach [37], then we present another variant of SOG proposed in Klai and Poitrenaud [58], this variant is called SOG-TGTA in this work because it is more suited to be used in a TGTA-based approach. In Section 7.6.3, we will show the results of an experimental comparison between the original approach based on SOG and TGBA and our approach based on TGTA and SOG-TGTA.

A SOG is a transformation of a Kripke structure allowing to aggregate states according to the set AP of atomic propostions observed in the property automaton. This transformation only preserves stutter-invariant properties. The constructed SOG is an explicit graph where each node is a symbolic set of states. Theses states are aggregated because they share the same values for the atomic propositions of AP (they may have different values for the other atomic propositions of the model that are not in AP).

In hybrid approaches, symbolic data structure are used to represent sets of states of the Kripke structure. The following symbolic operations are introduced in [38] to manipulate this symbolic aggregate of states.

Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ a Kripke structure, encoded by a Symbolic Kripke structure $K = \langle S_0, R, L \rangle$ (Definition 32). For a set of states $a \subseteq \mathcal{S}$ and a propositional formula $\phi \in \mathbb{B}(AP)$, the symbolic operations $\text{SuccF}(a,\phi)$ and $\text{ReachF}(a,\phi)$ are defined as follows:

- $\text{SuccF}(a,\phi) = \{s' \in \mathcal{S} \mid \exists s \in a, R(s,s') \wedge \exists \ell, [\ell \models \phi \wedge L(s',\ell)]\}$, i.e., the set of the **Succ**essors of states of aggregate a, **F**iltered to keep only those satisfying ϕ.
- $\text{ReachF}(a,\phi)$ computes the least subset of $\mathcal{S}$ satisfying:
  - $a \subseteq \text{ReachF}(a,\phi)$,
  - $\text{SuccF}(\text{ReachF}(a,\phi),\phi) \subseteq \text{ReachF}(a,\phi)$.

  $\text{ReachF}(a,\phi)$ is implemented using symbolic least fixed-points on Decision Diagrams.

Definition 43 (Homogeneous aggregate [37])**.** *Let $a \in 2^{\mathcal{S}} \setminus \{\emptyset\}$ be a subset of states of $\mathcal{K}$. We say that a is* a homogeneous aggregate *w.r.t. (**w**ith **r**espect **t**o) a given set of atomic propositions AP* iff $\forall s,s' \in a, l(s) \overset{AP}{=} l(s')$. *In other words, all the states of the aggregate a have the same valuation for all the atomic propositions in AP.*

For a homogeneous aggregate a w.r.t. AP, we write $l_{AP}(a) = l(s)_{|AP}$ for any state $s \in a$ (i.e., the valuation of a is the valuation of any one of its states).

For any $AP' \subseteq AP$, a homogeneous aggregate a w.r.t. AP is also homogeneous w.r.t. AP'.

7.3.1 SOG

Definition 44 (Symbolic Observation Graph [58, 37])**.** *Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ be a Kripke structure over the set of atomic propositions $AP_{\mathcal{K}}$. A* symbolic observation graph *over $AP \subseteq AP_{\mathcal{K}}$ is defined as $\widehat{\mathcal{K}}_{AP} = \langle \mathcal{S}', \mathcal{S}'_0, \mathcal{R}', l' \rangle$ a Kripke structure over AP satisfying :*

1. $\mathcal{S}' = \Gamma' \cup 2^{AP}$ *with* $\Gamma' = \left\{ a \in 2^{\mathcal{S}} \setminus \{\emptyset\} \;\middle|\; \begin{array}{l} a \text{ is homogeneous w.r.t. } AP \\ a = \text{ReachF}(a, l_{AP}(a)) \end{array} \right\}$

   Elements of Γ' are called aggregates *and elements of 2^{AP} are* divergent states.

2. $\forall a \in \mathcal{S}',\ l'(a) = \begin{cases} l_{AP}(a) & \text{if } a \in \Gamma' \\ a & \text{if } a \in 2^{AP} \end{cases}$

3. $\mathcal{R}' = \{a \to a' \in \Gamma' \times \Gamma' \mid a' = \text{ReachF}(\text{SuccF}(a, l'(a')) \setminus a, l'(a'))\}$
   $\cup \{a \to \ell \in \Gamma' \times 2^{AP} \mid a \text{ contains a cycle and } \ell = l'(a)\}$
   $\cup \{\ell \to \ell \mid \ell \in 2^{AP}\}$

4. $\mathcal{S}'_0 = \{a_{s_0} = \text{ReachF}(\{s_0\}, l(s_0)_{|AP}) \mid s_0 \in \mathcal{S}_0\}$.

The above Definition details how to build a SOG $\widehat{\mathcal{K}}_{AP}$. The set $\mathcal{S}'_0$ of initial states of $\widehat{\mathcal{K}}_{AP}$ is composed by the set of homogenous aggregates a_{s_0} satisfying $a_{s_0} = \text{ReachF}(\{s_0\}, l(s_0)_{|AP})$, i.e., for each initial state s_0 of $\mathcal{K}$, a_{s_0} is the set of reachable states s' from s_0 in $\mathcal{K}$ such that $l(s') \overset{AP}{=} l(s_0)$. The set $\mathcal{S}'$ of states of $\widehat{\mathcal{K}}_{AP}$ is composed of two kinds of nodes:

1. homogenous aggregates a satisfying $a = \text{ReachF}(a, \lambda_{AP}(a))$
2. divergent states $\ell \in 2^{AP}$ labeled with subsets of atomic propositions of AP.

For the transition relation of $\widehat{\mathcal{K}}_{AP}$, $\mathcal{R}'$ is composed of three kinds of edges:

1. case a and a' are two aggregates of Γ': $a \to a' \in \mathcal{R}'$ iff $l_{AP}(a) \neq l_{AP}(a')$ and a' contains every state $s' \in \Gamma$ satisfying $l(s')_{|AP} = l'(a')$ and $s \to s' \in \mathcal{R}$ with $s \in (a \cup a')$.
2. case a is an aggregates of Γ' and ℓ is a divergent state: $a \to \ell$ iff a contains a cycle and $l'(a) = \ell$.
3. each divergent state $\ell \in 2^{AP}$ has a self-loop $\ell \to \ell$.

Figure 7.1d shows the SOG $\widehat{\mathcal{K}}_{\{a,b\}}$ built from the Kripke structure $\mathcal{K}$ of Figure 7.1b according to $AP = \{a,b\}$ (ignoring the atomic proposition c of $\mathcal{K}$).

The initial state of $\widehat{\mathcal{K}}_{\{a,b\}}$ is an aggregate $\{s_0, s_1, s_2, s_3\}$ because they agree on the value of atomic propositions observed in $AP = \{a,b\}$: $l(s_0)_{|\{a,b\}} = l(s_1)_{|\{a,b\}} = l(s_2)_{|\{a,b\}} = l(s_3)_{|\{a,b\}} = a\bar{b}$. This initial aggregate contains a cycle so one of its successors is a divergent state labeled by $a\bar{b}$.

The constructed SOG $\widehat{\mathcal{K}}_{\{a,b\}}$ is also a Kripke structure, that allows to check any stutter-invariant property over the alphabet $2^{\{a,b\}}$. As example, Figure 7.1e represents $\mathcal{G} \otimes \widehat{\mathcal{K}}_{\{a,b\}}$, the synchronous product between the TGBA $\mathcal{G}$ of $a \mathsf{U} b$ and the SOG $\widehat{\mathcal{K}}_{\{a,b\}}$.

Theorem 2 ([58]). *Given a Kripke Structure $\mathcal{K}$ defined on $AP_{\mathcal{K}}$, then the SOG $\widehat{\mathcal{K}}_{AP}$ of $\mathcal{K}$ built over $AP \subseteq AP_{\mathcal{K}}$ preserves any stuttering-invariant property $\mathcal{A}$ on AP. In other words: $\mathscr{L}(\mathcal{A} \otimes \mathcal{K}) \neq \emptyset \iff \mathscr{L}(\mathcal{A} \otimes \widehat{\mathcal{K}}_{AP}) \neq \emptyset$.*

7.3.2 SOG for TGTA (SOG-TGTA)

Klai and Poitrenaud [58] proposed another variant of SOG that does not use divergent states. In a TGTA-based approach, we chose to use this variant of SOG, and we call it SOG-TGTA in this work. Instead of using divergent states, this SOG-TGTA has a self-loop on each aggregate that contains a cycle [58]. In addition, in TGTA all stuttering transitions are self-loops. Therefore, the synchronization between this stuttering self-loops and the self-loops of SOG-TGTA only produces self-loops in the product automaton, and therefore does not generate new states in this product.

Definition 45 (Symbolic Observation Graph [37] (SOG-TGTA)). *Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ be a Kripke structure over $AP_{\mathcal{K}}$. A* SOG-TGTA *over $AP \subseteq AP_{\mathcal{K}}$ of $\mathcal{K}$ is defined as $\widehat{\mathcal{K}}'_{AP} = \langle \mathcal{S}', \mathcal{S}'_0, \mathcal{R}', l' \rangle$ a Kripke structure over AP satisfying :*

1. $\mathcal{S}' = \left\{ a \in 2^{\mathcal{S}} \setminus \{\emptyset\} \;\middle|\; \begin{array}{l} a \text{ is homogeneous w.r.t. } AP \\ a = \text{ReachF}(a, l_{AP}(a)) \end{array} \right\}$
2. $\forall a \in \mathcal{S}',\ l'(a) = l_{AP}(a)$
3. $\mathcal{R}' = \{a \to a' \in \mathcal{S}' \times \mathcal{S}' \mid a' = \text{ReachF}(\text{SuccF}(a, l'(a')) \setminus a, l'(a'))\}$
   $\cup \{a \to a \in \Gamma' \times \Gamma' \mid a \text{ contains a cycle}\}$
4. $\mathcal{S}'_0 = \{\text{ReachF}(\{s_0\}, l(s_0)_{|AP}) \mid s_0 \in \mathcal{S}_0\}$.

The only difference between the above Definition 45 and the Definition 44 of SOG is the fact that SOG-TGTA does not use the divergent states labeled with the elements of 2^{AP}. These divergent states are replaced in SOG-TGTA by adding a self-loop on each aggregate that contains a cycle (see point 3 of Definition 45). The obtained SOG-TGTA contains only one kind of nodes: homogenous aggregates.

Theorem 3 ([58]). *Given a Kripke Structure $\mathcal{K}$ defined on $AP_{\mathcal{K}}$, then the SOG-TGTA $\widehat{\mathcal{K}}'_{AP}$ of $\mathcal{K}$ built over $AP \subseteq AP_{\mathcal{K}}$ preserves any stuttering-invariant property $\mathcal{A}$ on AP. In other words: $\mathscr{L}(\mathcal{A} \otimes \mathcal{K}) \neq \emptyset \iff \mathscr{L}(\mathcal{A} \otimes \widehat{\mathcal{K}}'_{AP}) \neq \emptyset$.*

Figure 7.2d shows an example of a SOG-TGTA $\mathcal{K}'_{\{a,b\}}$ and Figure 7.2e presents the product $\mathcal{T} \otimes \widehat{\mathcal{K}}'_{\{a,b\}}$ of $\mathcal{K}'_{\{a,b\}}$ with the TGTA $\mathcal{T}$ of $a \cup b$. $\widehat{\mathcal{K}}'_{\{a,b\}}$ is similar to the SOG $\widehat{\mathcal{K}}_{\{a,b\}}$ of Figure 7.1d, but without the divergent state labeled with $a\bar{b}$. In addition, the initial aggregate of $\mathcal{K}'_{\{a,b\}}$ has a self-loop because it contains a cycle.

We can notice that $\mathcal{T} \otimes \widehat{\mathcal{K}}'_{\{a,b\}}$ is smaller than the product $\mathcal{G} \otimes \widehat{\mathcal{K}}_{\{a,b\}}$ (Figure 7.1e) using the TGBA $\mathcal{G}$ (recognizing the same formula $a \cup b$ as the TGTA $\mathcal{T}$). We will present in Section 7.6 an experimental comparison that will confirm this observation.

7.4 Symbolic Observation Product (SOP)

The *Symbolic Observation Product* (SOP) [37] is a dynamic extension of SOG that exploits the fact that the number of observed atomic propositions decreases as we progress in the property automaton. The goal of this extension is to allow further aggregation than SOG. However, contrary to SOG, which is an abstraction of a Kripke structure, SOP is an hybrid synchronous product between a Kripke structure and the TGBA of a stutter-invariant property. In this section, we start by giving the definition and an illustrative example of SOP. Then, we present SOP-TGTA, an adaptation of SOP that uses TGTA instead of TGBA to represent the property automaton.

In Section 7.6.4, we will present the results of an experimental comparison between the original SOP approach based on TGBA and our SOP-TGTA approach based on TGTA.

Given a TGBA $\mathcal{G} = \langle Q, I, \delta, \mathcal{F} \rangle$ or a TGTA $\mathcal{G} = \langle Q, I, U, \delta, \mathcal{F} \rangle$, the alphabet $\mathrm{FV}(q)$ of a state $q \in Q$ is defined as the union of the atomic propositions which can be observed from q. Formally, $\mathrm{FV}(q) = \bigcup_{q_1 \xrightarrow{\phi,F} q_2 \in \delta^\star(q)} \mathrm{FV}(\phi)$ with $\delta^\star(q)$ is the set of transitions reachable from a state q. From this definition of $\mathrm{FV}(q)$, we can easily deduce that for any $q_1 \xrightarrow{\phi,F} q_2 \in \delta$, we have $\mathrm{FV}(q_1) \supseteq \mathrm{FV}(q_2)$. In other words, the set of observed atomic propositions decreases as we progress through the successive states of the property automaton (TGBA or TGTA).

7.4.1 SOP

SOP [38] is an hybrid product that is constructed over a dynamic alphabet (FV) which decreases as the constrcution of the product progresses. This allows to obtain larger aggregates and therefore fewer number of states than in SOG approach.

Definition 46 (SOP of a TGBA and a Kripke structure)**.** *Given a stutter-invariant TGBA* $\mathcal{G} = \langle Q, I, \delta, \mathcal{F} \rangle$ *over AP and a Kripke structure* $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$*, the* Symbolic Observation Product *of* $\mathcal{G}$ *and* $\mathcal{K}$ *is the TGBA denoted* $\mathcal{G} \widehat{\otimes} \mathcal{K} = \langle Q_{\widehat{\otimes}}, I_{\widehat{\otimes}}, \delta_{\widehat{\otimes}}, \mathcal{F} \rangle$ *where:*

1. $Q_{\widehat{\otimes}} = Q' \cup \mathcal{D}'$ *where states constructed from aggregates are in* Q' *and divergent states are in* $\mathcal{D}'$*:*

$$Q' = \left\{ (q,a) \in Q \times (2^{\Gamma} \setminus \{\emptyset\}) \;\middle|\; \begin{array}{l} a \text{ is homogeneous w.r.t. } \mathrm{FV}(q) \\ a = \mathrm{ReachF}(a, l_{\mathrm{FV}(q)}(a)) \end{array} \right\}$$

$$\mathcal{D}' = \{(q,\ell) \mid q \in Q \text{ and } \ell \in 2^{\mathrm{FV}(q)}\}$$

2. $$\delta_{\widehat{\otimes}} = \left\{ (q_1,a_1) \xrightarrow{\ell,F} (q_2,a_2) \;\middle|\; \begin{array}{l} (q_1,a_1) \in Q', (q_2,a_2) \in Q', \ell = l_{\mathrm{FV}(q_1)}(a_1), \\ \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi,F} q_2 \in \delta, \text{ and } \ell \models \phi, \\ \exists \ell' \in 2^{\mathrm{FV}(q_2)} \text{ s.t. } a_2 = \mathrm{ReachF}(\mathrm{SuccF}(a_1,\ell') \setminus a_1, \ell') \end{array} \right\}$$

$$\cup \left\{ (q_1,a) \xrightarrow{\ell_1,F} (q_2,\ell_2) \;\middle|\; \begin{array}{l} (q_1,a) \in Q', (q_2,\ell_2) \in \mathcal{D}', \ell_1 = l_{\mathrm{FV}(q_1)}(a), \\ a \text{ contains a cycle}, \ell_2 = \ell_{1|\mathrm{FV}(q_2)}, \\ \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi,F} q_2 \in \delta, \text{ and } \ell_1 \models \phi \end{array} \right\}$$

$$\cup \left\{ (q_1,\ell_1) \xrightarrow{\ell_1,F} (q_2,\ell_2) \;\middle|\; \begin{array}{l} (q_1,\ell_1) \in \mathcal{D}', (q_2,\ell_2) \in \mathcal{D}', \ell_2 = \ell_{1|\mathrm{FV}(q_2)}, \\ \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi,F} q_2 \in \delta, \text{ and } \ell_1 \models \phi \end{array} \right\}$$

3. $I_{\widehat{\otimes}} = \{(q_0, \mathrm{ReachF}(\{s_0\}, l(s_0)_{|\mathrm{FV}(q_0)})) \mid (q_0,s_0) \in I \times \mathcal{S}_0\}$

We have $\mathscr{L}(\mathcal{G} \otimes \mathcal{K}) \neq \emptyset \iff \mathscr{L}(\mathcal{G} \widehat{\otimes} \mathcal{K}) \neq \emptyset$ *by construction.*

As in SOG, the set of states $Q_{\widehat{\otimes}}$ of a SOP is composed of two kinds of states in Q' and $\mathcal{D}'$. The states of Q' are pairs of the form (q,a), where q is a state of TGBA and a is an aggregate of states from the Kripke structure. a is similar to an aggregate in a SOG but computed according to the alphabet $\mathrm{FV}(q)$ instead of all AP. The states of $\mathcal{D}'$ are the divergent states of the SOP.

The SOP transition relation $\delta_{\widehat{\otimes}}$ is composed of three parts. The first part contains the transitions of the form $(q_1,a_1) \to (q_2,a_2)$, where q_1, q_2 are two successive states of the TGBA, and a_1, a_2 are two aggregates of states from the Kripke structure. The aggregate a_2 contains the successors of states of a_1 and its computation is similar to SOG, except that a_2 is homogeneous w.r.t. the set $\mathrm{FV}(q_2)$ instead of all AP. The second and third parts of the SOP transition relation $\delta_{\widehat{\otimes}}$ are also similar to the SOG transition relation parts for cycle detection. Figure 7.1f shows an example of a SOP $\mathcal{G} \widehat{\otimes} \mathcal{K}$ computed from the Kripke structure $\mathcal{K}$ and the TGBA $\mathcal{G}$ of $a \mathsf{U} b$. The difference between the SOP and the product $\mathcal{G} \otimes \widehat{\mathcal{K}}_{\{a,b\}}$ using the SOG (Figure 7.1e), is mainly when the TGBA $\mathcal{G}$ reaches the state q_1. Indeed, from this state, the alphabet $\mathrm{FV}(q_1)$ becomes empty. This allows the SOP to aggregate the states $\{s_4, s_5, s_6, s_7\}$. In addition, these states form a cycle in $\mathcal{K}$, and therefore implies to add a divergent state $(q_1, \top)$ in the SOP.

7.4.2 SOP Using TGTA (SOP-TGTA)

In this section, we propose a TGTA-based SOP, called SOP-TGTA. The main difference between SOP and SOP-TGTA is related to the changesets labeling the TGTA and their synchronization with the states of the Kripke structure.

Definition 47 (SOP of a TGTA and a Kripke structure)**.** *Given a TGTA* $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$ *over AP and a Kripke structure* $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$*, the* SOP-TGTA *of* $\mathcal{T}$ *and* $\mathcal{K}$ *is the TGTA denoted* $\mathcal{T} \widehat{\otimes} \mathcal{K} = \langle Q_{\widehat{\otimes}}, I_{\widehat{\otimes}}, U_{\widehat{\otimes}}, \delta_{\widehat{\otimes}}, \mathcal{F} \rangle$ *where:*

1. $Q_{\widehat{\otimes}} = \mathcal{Q}' \cup \mathcal{D}'$ *where states of the automaton are synchronized with aggregates in* $\mathcal{Q}'$ *and with divergent states in* $\mathcal{D}'$*:*

$$\mathcal{Q}' = \left\{ (q,a) \in Q \times (2^{\mathcal{S}} \setminus \{\emptyset\}) \;\middle|\; \begin{array}{l} a \text{ is homogeneous w.r.t. } \mathrm{FV}(q) \\ a = \mathrm{ReachF}(a, l_{\mathrm{FV}(q)}(a)) \end{array} \right\}$$

$$\mathcal{D}' = \{(q,\ell) \mid q \in Q \text{ and } \ell \in 2^{\mathrm{FV}(q)}\}$$

2. $$\delta_{\widehat{\otimes}} = \left\{ (q_1,a_1) \xrightarrow{(\ell \oplus \ell'),F} (q_2,a_2) \;\middle|\; \begin{array}{l} (q_1,a_1) \in \mathcal{Q}', (q_2,a_2) \in \mathcal{Q}', \ell = l_{\mathrm{FV}(q_1)}(a_1), \\ \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi,F} q_2 \in \delta, \\ \exists \ell' \in 2^{\mathrm{FV}(q_1)} \text{ s.t. } (\ell \oplus \ell') \models \phi, \text{ and} \\ a_2 = \mathrm{ReachF}(\mathrm{SuccF}(a_1, \ell'_{|\mathrm{FV}(q_2)}) \setminus a_1, \ell'_{|\mathrm{FV}(q_2)}) \end{array} \right\}$$
$$\cup \left\{ (q_1,a) \xrightarrow{\emptyset,F} (q_2,\ell_2) \;\middle|\; \begin{array}{l} (q_1,a) \in \mathcal{Q}', (q_2,\ell_2) \in \mathcal{D}', \ell_1 = l_{\mathrm{FV}(q_1)}(a), \\ a \text{ contains a cycle}, \ell_2 = \ell_{1|\mathrm{FV}(q_2)}, \\ \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi,F} q_2 \in \delta, \text{ and } \emptyset \models \phi \end{array} \right\}$$
$$\cup \left\{ (q_1,\ell_1) \xrightarrow{\emptyset,F} (q_2,\ell_2) \;\middle|\; \begin{array}{l} (q_1,\ell_1) \in \mathcal{D}', (q_2,\ell_2) \in \mathcal{D}', \ell_2 = \ell_{1|\mathrm{FV}(q_2)}, \\ \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi,F} q_2 \in \delta, \text{ and } \emptyset \models \phi \end{array} \right\}$$

3. $I_{\widehat{\otimes}} = \{(q_0, \mathrm{ReachF}(\{s_0\}, l(s_0)_{|\mathrm{FV}(q_0)})) \mid (q_0,s_0) \in I \times \mathcal{S}_0 \text{ and } l(s_0) \models U(q_0)\}$,

4. $\forall (q_0,a_0) \in I_{\widehat{\otimes}}, U_{\widehat{\otimes}}((q_0,a_0)) = l_{\mathrm{FV}(q_0)}(a_0)$.

We have $\mathscr{L}(\mathcal{T} \otimes \mathcal{K}) \neq \emptyset \iff \mathscr{L}(\mathcal{T} \widehat{\otimes} \mathcal{K}) \neq \emptyset$ *by construction.*

The set of states $Q_{\widehat{\otimes}}$ of a SOP-TGTA is the same as in SOP. But, the transition relation is a little different because it is based on changesets. As in SOP, the transition relation $\delta_{\widehat{\otimes}}$ of SOP-TGTA is composed of three rules. The first rule defines the transitions between aggregates, the second rule is about transitions between aggregates and divergent states, and the third rule is for transitions between divergent states.

The transitions between aggregates are of the form $(q_1,a_1) \to (q_2,a_2)$, where q_1,q_2 are two successive states of the TGTA, and a_1,a_2 are two aggregates of states from the Kripke structure. Each aggregate a_2 contains the successors of states of a_1, filtered to keep only those satisfying a valuation $\ell'_{|\mathrm{FV}(q_2)}$, where ℓ' satisfies $(\ell \oplus \ell') \models \phi$ with $\ell = l_{\mathrm{FV}(q_1)}(a_1)$ and $q_1 \xrightarrow{\phi,F} q_2$. In other words, ℓ' is obtained from $\ell = l_{\mathrm{FV}(q_1)}(a_1)$ by applying one changeset from the set of changesets encoded by ϕ. In addition, each aggregate a_2 is homogeneous w.r.t. the set of atomic prpositions of $\mathrm{FV}(q_2)$. In the second and third parts of $\delta_{\widehat{\otimes}}$ of SOP-TGTA, all the transitions between aggregates and divergent states and all the self-loops on divergent states are labeled with an empty changeset $\emptyset$. Figure 7.2f shows an example of a SOP-TGTA $\mathcal{T} \widehat{\otimes} \mathcal{K}$ computed from the Kripke structure $\mathcal{K}$ and the TGTA $\mathcal{T}$ of $a \mathsf{U} b$. The first difference between the SOP-TGTA $\mathcal{T} \widehat{\otimes} \mathcal{K}$ and the product $\mathcal{T} \otimes \widehat{\mathcal{K}}'_{\{a,b\}}$ using SOG-TGTA (Figure 7.2e), is the divergent state labeled with $(q_0, a\bar{b})$. This divergent state is added to the SOP-TGTA because the aggregate $\{s_0,s_1,s_2,s_3\}$ of the initial state

contains a cycle. The second difference between $\mathcal{T} \widehat{\otimes} \mathcal{K}$ and $\mathcal{T} \otimes \widehat{\mathcal{K}}'_{\{a,b\}}$ comes from the fact that the alphabet $\mathrm{FV}(q_1)$ becomes empty when the TGTA $\mathcal{T}$ reaches the state q_1. This allow the SOP-TGTA to aggregate the states $\{s_4, s_5, s_6, s_7\}$ of $\mathcal{K}$. In addition, this aggregate contains a cycle, and therefore allows to add in $\mathcal{T} \widehat{\otimes} \mathcal{K}$ a divergent state labeled with $(q_1, \top)$.

In our example of verification of the LTL property $a \mathsf{U} b$ on the Kripke structure $\mathcal{K}$, we observe that the SOP-TGTA $\mathcal{T} \widehat{\otimes} \mathcal{K}$ of Figure 7.2f is smaller than the original SOP $\mathcal{G} \widehat{\otimes} \mathcal{K}$ of Figure 7.1f.

7.5 Self-Loop Aggregation Product (SLAP)

SLAP [38] is an hybrid synchronous product, in which the aggregation of Kripke structure states is based on the self-loops of the property automaton. This section presents the original SLAP based on TGBA and SLAP-TGTA, a variant of SLAP based on TGTA. In Section 7.6.5, we will show the results of an experiment comparing the performance of SLAP and SLAP-TGTA.

Definition 48. *Given a TGBA $\mathcal{G} = \langle Q, I, \delta, \mathcal{F} \rangle$ or a TGTA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$, for a state $q \in Q$,* $\mathrm{SF}(q)$ *encodes the **S**elf-loop **F**ormulas labeling edges $q \to q$. Formally,*

$$\mathrm{SF}(q) = \bigvee_{q \xrightarrow{\phi, F} q \in \delta} \phi$$

.

Similar to the symbolic operations SuccF and ReachF defined in Section 7.3, for SLAP approach, we define two other operations $\mathrm{FReach}(a, \phi)$ and $\mathrm{Succ}^{\oplus}(a, \phi)$ with $a \subseteq \mathcal{S}$ is a set of states of $\mathcal{K}$ and $\phi \in \mathbb{B}(AP)$ is a propositional formula, such that:

- $\mathrm{FSucc}(a, \phi) = \{s' \in \mathcal{S} \mid \exists s \in a, R(s, s') \wedge \exists \ell, [\ell \models \phi \wedge L(s, \ell)]\}$, i.e, **F**ilter a to only keep states satisfying ϕ, then produce their **Succ**essors. The difference between SuccF and FSucc is whether the filter is applied on the source or destination states.

- $\mathrm{FReach}(a, \phi)$ computes the least subset of $\mathcal{S}$ satisfying:
  - $a \subseteq \mathrm{FReach}(a, \phi)$,
  - $\mathrm{FSucc}(\mathrm{FReach}(a, \phi), \phi) \subseteq \mathrm{FReach}(a, \phi)$.

7.5.1 SLAP

A SLAP is an hybrid product between a Kripke structure $\mathcal{K}$ and a TGBA $\mathcal{G}$. The states of the SLAP are pairs of the form (q, a) composed of a state q of $\mathcal{G}$ and an aggregate a containing successive states of $\mathcal{K}$ aggregated as long as they model $\mathrm{SF}(q)$. These aggregates are computed as symbolic least fixed-points using the operations FSucc and FReach defined in Section 7.2.

Definition 49 (SLAP of a TGBA and a Kripke structure)**.** *Given a TGBA $\mathcal{G} = \langle Q, I, \delta, \mathcal{F} \rangle$ over AP and a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$, the* Self-Loop Aggregation Product *of $\mathcal{G}$ and $\mathcal{K}$ is the TGBA denoted $\mathcal{G} \boxtimes \mathcal{K} = \langle Q_{\boxtimes}, I_{\boxtimes}, \delta_{\boxtimes}, \mathcal{F} \rangle$ where:*

- $Q_{\boxtimes} = Q \times (2^{\mathcal{S}} \setminus \{\emptyset\})$

- $\delta_{\boxtimes} = \left\{ (q_1,a_1) \xrightarrow{\top,F} (q_2,a_2) \;\middle|\; \begin{array}{r} \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi,F} q_2 \in \delta, \\ q_1 = q_2 \Rightarrow F \neq \emptyset, \text{ and} \\ a_2 = \text{FReach}(\text{FSucc}(a_1,\phi), \text{SF}(q_2)) \end{array} \right\}$
- $I_{\boxtimes} = \{(q^0, \text{FReach}(\{s_0\}, \text{SF}(q^0))) \mid (q_0,s_0) \in I \times \mathcal{S}_0\}$

We have $\mathscr{L}(\mathcal{G} \otimes \mathcal{K}) \neq \emptyset \iff \mathscr{L}(\mathcal{G} \boxtimes \mathcal{K}) \neq \emptyset$ *by construction.*

All transitions of the constructed SLAP are labeled with the formula $\top$, because these labels are irrelevant when checking language emptiness of SLAP.

Figure 7.1g shows an example of SLAP $\mathcal{G} \boxtimes \mathcal{K}$ obtained from the Kripke structure $\mathcal{K}$ and the TGBA $\mathcal{G}$ of $a \mathsf{U} b$. The initial state of $\mathcal{G} \boxtimes \mathcal{K}$ is the pair $\left(q_0, \left\{ \begin{smallmatrix} s_0 & s_1 \\ s_2 & s_3 \\ s_4 \end{smallmatrix} \right\}\right)$, where q_0 is the initial state of $\mathcal{G}$ and the aggregate $a_1 = \left\{ \begin{smallmatrix} s_0 & s_1 \\ s_2 & s_3 \\ s_4 \end{smallmatrix} \right\}$ is obtained by iteratively aggregating the successors of the states that satisfy $\text{SF}(q_0) = a\bar{b}$. Then, from the initial state (q_0,a_1), we explore the transition $q^0 \xrightarrow{b} q_1$ of TGBA and we obtain only one successor (q_1,a_2) with the aggregate $a_2 = \text{FReach}(\text{FSucc}(a_1,b), \text{SF}(q_1))$ computed as follows:

- The set $\text{FSucc}(a_1,b)$ contains only the successors of $\{s_4\}$ because only the state s_4 in a_1 satisfies b, thus $\text{FSucc}(a_1,b) = \{s_5\}$;
- $\text{SF}(q_1) = \top$ because $q_1 \xrightarrow{\top} q_1$
- We deduce that $a_2 = \text{FReach}(\{s_5\}, \top)$. a_2 contains all the reachable states from s_5 (satisfying $\top$), thus $a_2 = \{s_4, s_5, s_6, s_7\}$.

Finally, we explore the TGBA transition $q_1 \xrightarrow{\top,\bullet} q_1$ and we obtain an accepting self-loop on (q_1,a_2) because the successor aggregate of a_2 is a_2 (i.e, $\text{FReach}(\text{FSucc}(a_2,\top), \text{SF}(q_1)) = a_2$).

7.5.2 SLAP Using TGTA (SLAP-TGTA)

The SLAP-TGTA is a variant of SLAP based on TGTA instead of TGBA. In SLAP-TGTA, the states of the Kripke structure are aggregated according to the changesets labelling the TGTA transitions. In particular, each $\text{SF}(q)$ represents the set of changesets encoded by the **S**elf-loop **F**ormulas labeling edges $q \to q$ of the TGTA. Therefore, in SLAP-TGTA the successive states of the Kripke structure are aggregated as long as they change according to the changesets encoded by $\text{SF}(q)$. These aggregates are computed as least fixed-points based on changesets using the symbolic operations $\text{Succ}^{\oplus}$ and $\text{Reach}^{\oplus}$ defined as follows:

Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ a Kripke structure, encoded by a changeset-based symbolic Kripke structure $K^{\oplus} = \langle S_0, R^{\oplus}, L \rangle$ (Definition 37 in page 118). For a set of states $a \subseteq \mathcal{S}$ and a propositional formula $\phi \in \mathbb{B}(AP)$, we define the following symbolic operations:

- $\text{Succ}^{\oplus}(a,\phi) = \{s' \in \mathcal{S} \mid \exists s \in a, \exists k, [k \models \phi \wedge R^{\oplus}(s,k,s')]\}$, i.e, the set of the **Succ**essors states of a Filtered to keep only those satisfying $k \models \phi$ where $k = l(s) \oplus l(s')$ is a changeset between $l(s)$ and $l(s')$.
- $\text{Reach}^{\oplus}(a,\phi)$ computes the least subset of $\mathcal{S}$ satisfying:
  - $a \subseteq \text{Reach}^{\oplus}(a,\phi)$,
  - $\text{Succ}^{\oplus}(\text{Reach}^{\oplus}(a,\phi),\phi) \subseteq \text{Reach}^{\oplus}(a,\phi)$.

Definition 50 (SLAP of a TGTA and a Kripke structure)**.** *Given a TGTA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$ over AP and a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$, the* SLAP-TGTA *of $\mathcal{T}$ and $\mathcal{K}$ is the TGTA denoted $\mathcal{T} \boxtimes \mathcal{K} = \langle Q_\boxtimes, I_\boxtimes, \delta_\boxtimes, \mathcal{F} \rangle$ where:*

- $Q_\boxtimes = Q \times (2^{\mathcal{S}} \setminus \{\emptyset\})$
- $\delta_\boxtimes = \left\{ (q_1, a_1) \xrightarrow{\top, F} (q_2, a_2) \;\middle|\; \begin{array}{l} \exists \phi \in \mathbb{B}(AP) \text{ s.t. } q_1 \xrightarrow{\phi, F} q_2 \in \delta, \\ \qquad q_1 = q_2 \Rightarrow F \neq \emptyset, \text{ and} \\ \qquad a_2 = \text{Reach}^{\oplus}(\text{Succ}^{\oplus}(a_1, \phi), \text{SF}(q_2)) \end{array} \right\}$
- $q^0_\boxtimes = \{(q^0, \text{Reach}^{\oplus}(\{s_0\}, \text{SF}(q^0))) \mid (q_0, s_0) \in I \times \mathcal{S}_0 \text{ and } l(s_0) \models U(q_0)\}$

We have $\mathcal{L}(\mathcal{T} \otimes \mathcal{K}) \neq \emptyset \iff \mathcal{L}(\mathcal{T} \boxtimes \mathcal{K}) \neq \emptyset$ by construction.

For the same reason as in SLAP, the SLAP-TGTA transitions are only labeled with $\top$. The reachable states of a SLAP-TGTA are of the form (q, a) where q is a state of the TGTA and a is an aggregate of states of the Kripke structure such that: For each state $s \in a$, if s' is a successor of s in the Kripke structure with $l(s) \oplus l(s') \models \text{SF}(q)$, then $s' \in a$.

Figure 7.2g presents $\mathcal{T} \boxtimes \mathcal{K}$, an example of SLAP-TGTA computed from the Kripke structure $\mathcal{K}$ and the TGTA $\mathcal{T}$ of $a \mathsf{U} b$. Because $l(s_0) = a\bar{b}c \models U(q_0) = a\bar{b}$, the initial state of $\mathcal{T} \boxtimes \mathcal{K}$ is the pair (q_0, a_1), where a_1 is computed from s_0 by iteratively aggregates successors that change according to a changeset belonging to (the set of changesets encoded by) $\text{SF}(q_0)$. Formally, $a_1 = \text{Reach}^{\oplus}(\{s_0\}, \text{SF}(q^0)) = \{ \begin{smallmatrix} s_0 & s_1 \\ s_2 & s_3 \end{smallmatrix} \}$ (because $l(s_0) \oplus l(s_1) = l(s_1) \oplus l(s_2) = l(s_2) \oplus l(s_3) = \emptyset \models \text{SF}(q_0) = \bar{a}\bar{b}$). Then, in order to compute the successors of (q_0, a_1), we explore the transition $q^0 \xrightarrow{b} q_1$ of TGTA. We obtain only one successor (q_1, a_2) with the aggregate $a_2 = \text{Reach}^{\oplus}(\text{Succ}^{\oplus}(a_1, b), \text{SF}(q_1))$ is computed as follows:

- $\text{Succ}^{\oplus}(a_1, b) = \{s_4\}$ because s_4 is a successor of $s_0 \in a_1$ and it is the unique successor of states of a_1 that satisfies $l(s_0) \oplus l(s_4) \models b$ ($l(s_0) \oplus l(s_4) = \{b, c\}$),
- $\text{SF}(q_1) = \top$ because $q_1 \xrightarrow{\top} q_1$
- Thus, $a_2 = \text{Reach}^{\oplus}(\{s_4\}, \top) = \{s_4, s_5, s_6, s_7\}$ because a_2 contains all the reachable states from s_4 through any changesets.

Finally, we compute the successors of (q_1, a_2) by exploring the TGTA transition $q_1 \xrightarrow{\top, \bullet} q_1$. This TGTA accepting self-loop also generates an accepting self-loop on state (q_1, a_2) of the SLAP-TGTA. Indeed, the unique successor of (q_1, a_2) is itself because:
$\text{Reach}^{\oplus}(\text{Succ}^{\oplus}(a_2, \top), \top) = a_2$ (the states of a_2 are in a cycle).

7.6 Experimental Comparison of Hybrid Approaches using TGBA vs. TGTA

In the previous sections, we presented three hybrid approaches based on TGBA (SOG, SOP and SLAP) and their variants based on TGTA (SOG-TGTA, SOP-TGTA and SLAP-TGTA). This section presents an experimental evaluation conducted to compare each hybrid approach with its variant based on TGTA.

This experimentation is based on BEEM benchmark [67]. It reuses the same benchmark inputs, formulas and models used to evaluate the fully symbolic approaches in Section 6.4.3 page 124.

7.6.1 Implementation

We have implemented SOG-TGTA, SOP-TGTA and SLAP-TGTA in the same tool LTL-ITS [1] that already contains SOG, SOP, SLAP and the fully symbolic approaches (using TGBA/TGTA). LTL-ITS tool is built on top of libraries [2]: SDD/ITS, Spot, and LTSmin. These three libraries were already presented in the previous chapter, Section 6.4.1, page 122.

The DVE variant of LTSmin [12] is used to produce an ETF file representing the transition relation of each BEEM model. These ETF files are loaded by the SDD/ITS [85] library to encode the symbolic transition relations ($R(s,s')$) of the Kripke structures, used to implement the symbolic operations ReachF and FReach. For SLAP-TGTA, it is a changeset-based symbolic transition relations $R^{\oplus}(s,s')$ (Definition 37) that are built from the ETF files. It is used to implement the symbolic operation Reach$^{\oplus}$ for the SLAP-TGTA aggregates computation.

The Spot library [36] is used to translate the LTL properties into TGBA or TGTA. It is also used to perform the emptiness check of explicit graphs, such as the synchronous products TGBA/SOG and TGTA/SOG-TGTA. In addition, the hybrid synchronous products SOP, SLAP, SOP-TGTA and SLAP-TGTA are also handled by the emptiness check of Spot. Indeed, they are explicit graphs in which each node stores a set of states encoded as a Decision Diagram. These sets of states are computed using least fixed-points (ReachF, FReach or Reach$^{\oplus}$).

A SOG-TGTA is implemented in the same way as a SOG, as a concrete class of the Kripke structure interface provided by Spot. During the the emptiness check of the products TGBA/SOG and TGTA/SOG-TGTA, the SOG and the SOG-TGTA nodes are constructed on-the-fly using the implementation of the symbolic operation ReachF.

Similar to SOP and SLAP, we have implemented SOP-TGTA and SLAP-TGTA as concrete classes of the synchronous product interface of Spot. During the the emptiness check, the nodes of these four hybrid products are built on-the-fly from the states of the property automaton (TGBA or TGTA) and using the symbolic operations: ReachF for SOP and SOP-TGTA, FReach for SLAP and Reach$^{\oplus}$ for SLAP-TGTA.

7.6.2 Results

The results of our experimentations are presented as scatter plots using logarithmic scale. Each scatter plot compares an hybrid approach against its variant based on TGTA. Each point represents the comparison of the performance of the model checking for a model and formula pair. Any process that exceeded 60 minutes of runtime or 6GB of RAM was aborted (thus, the answer of the model checker was not reported for some cases). In our scatter plots, these aborted experiments are plotted as being three times higher than the maximum of the other values. Thus, these points appear separately (by the wide white band) from the other experiments that succeed.

[1] http://ddd.lip6.fr

[2] Respectively http://ddd.lip6.fr, http://spot.lip6.fr, and http://fmt.cs.utwente.nl/tools/ltsmin.

7.6.3 SOG versus SOG-TGTA

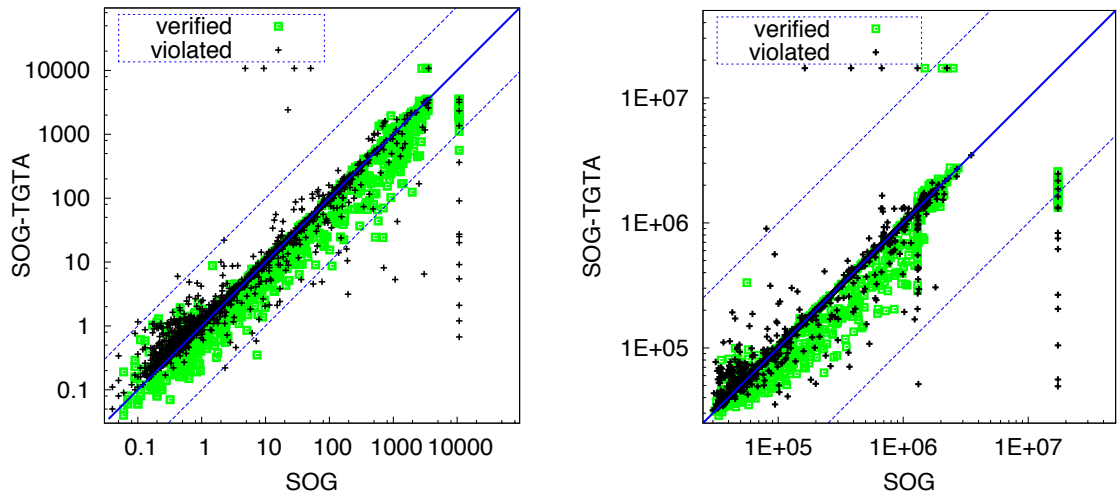


Figure 7.3: Performance comparison of SOG against SOG-TGTA. Left: time (in seconds); Right: memory (in kilobytes).

		SOG	SOG-TGTA
Verified Formulas	Win	315	826
	Fail	39	4
Violated Formulas	Win	1441	609
	Fail	14	5

Table 7.1: On all successful experiments, we count the number of cases a specific method has (Win) the best time. The Fail line shows the cases were an approach failed to solve an experiment solved by the other approach.

The scatter plots of Figure 7.3 compare the performance of two hybrid approaches: the first is based on TGBA and SOG (called just SOG approach in the following); the second is based on TGTA and SOG-TGTA (called SOG-TGTA approach). Each point of the left and right scatter plots compares respectively the time and memory used in the model checking of each pair (formula, model). The x-axis represents the performance of SOG approach and the y-axis shows the performance of SOG-TGTA approach, so the points below the diagonal correspond to cases where the SOG-TGTA approach is better. Symmetrically, the points above the diagonal corresponds to points were the SOG approach is better. The points represented by green squares correspond to verified formulas (empty products), and the black crosses correspond to violated formulas (non-empty products).

In the two scatter plots, we observe that for verified formulas (green points), the SOG-TGTA approach outperforms the SOG approach (in time and memory). This result is similar to the comparison between the explicit approaches based on TGBA versus TGTA, presented in Chapter 5 (Section 5.5.2 page 99). This similarity is justified by the fact that these approaches are based on

traditional explicit synchronous products.

For violated formulas, the SOG approach outperforms the SOG-TGTA approach in the cases where the execution time is less than one second. On the contrary, for hard cases, there are more cases that failed using SOG than using SOG-TGTA (compare the "aligned" black points at top and on right of scatter plots, or for more details see Table 7.1).

In total in these scatter plots, SOG failed in 53 cases solved by SOG-TGTA, while SOG-TGTA approach failed for only 9 cases solved by SOG. In addition, 875 cases are not solved by the two approaches within the time and memory limits. In the experiments that have not failed, SOG-TGTA was at least a ten times faster than SOG in 20 cases, and twice times faster in 287 cases. A contrario, SOG was at least a ten times faster than SOG-TGTA in 4 cases, and twice times faster in 144 cases.

Table 7.1 presents the best and the failed approach for each experiment where at least one approach succeeded. This table does not take into account cases where the two approaches failed.

7.6.4 SOP versus SOP-TGTA

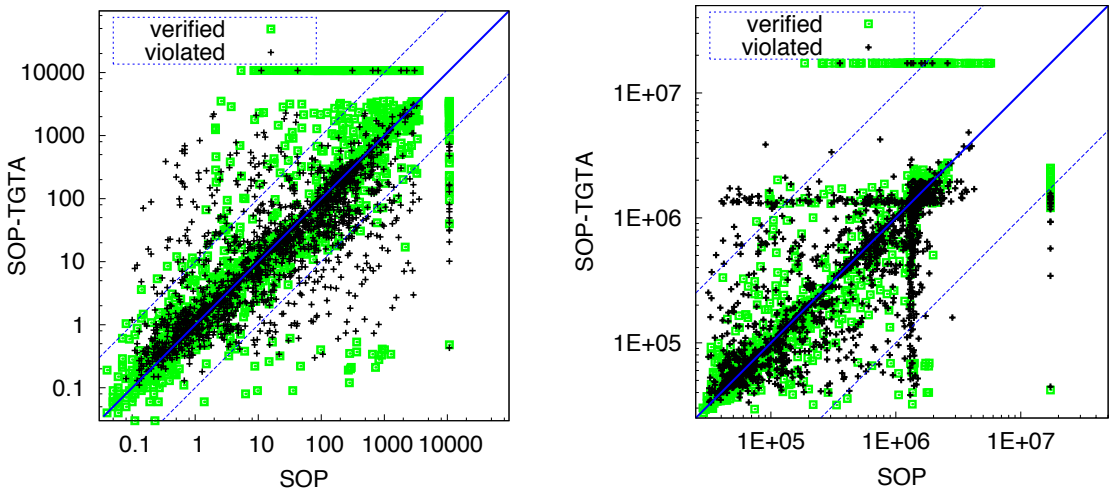


Figure 7.4: Performance comparison of SOP against SOP-TGTA. Left: time (in seconds); Right: memory (in kilobytes).

		SOP	SOP-TGTA
Verified Formulas	Win	915 (61%)	593 (39%)
	Fail	44 (2%)	327 (21%)
Violated Formulas	Win	957 (47%)	1061 (52%)
	Fail	18 (0%)	8 (0%)

Table 7.2: On all successful experiments, we count the number of cases a specific method has (Win) the best time. The Fail line shows the cases were each approach failed to compute the result of an experiment solved by the other approach.

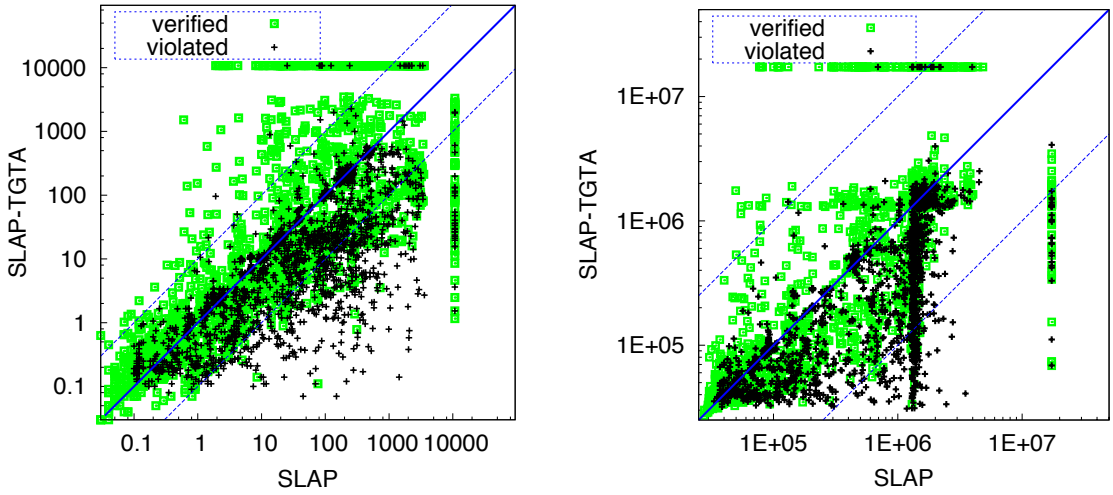


Figure 7.5: Performance comparison of SLAP against SLAP-TGTA. Left: time (in seconds); Right: memory (in kilobytes).

The scatter plots of Figure 7.4 compares the performance of SOP against SOP-TGTA. Each point compare the time (left scatter plot) and memory (right scatter plot) used to perform the model checking of each pair (formula, model) of our benchmark. The points below the diagonal correspond to cases where the SOP-TGTA approach is better, and for the other points the SOP approach is better.

On the one hand, the results of the scatter plots are very difficult to interpret, because there are many points where SOP is better and many others with SOP-TGTA which is the best (SOP-TGTA was at least a hundred times faster than SOP in 36 cases, ten times faster in 160 cases, and twice times faster in 579 cases. A contrario, SOP was at least one hundred times faster than SOP-TGTA in 38 cases, ten times faster in 169 cases, and twice times faster in 511 cases).

On the other hand, if we look at the scatter plots in more detail, we can observe that there are more cases of failure for SOP-TGTA (represented by the linear cloud at the top of the scatter plots) than the SOP approach (linear cloud on the right of the scatter plots). Indeed, SOP-TGTA failed for 335 experiments where SOP reached the result, while SOP only failed for 62 cases solved by SOP-TGTA. Table 7.2 gives more details for these failed cases by distinguishing verified and violated formulas (without taking into account the cases where the two approaches failed). This table also shows the number of cases in which each approach is better.

7.6.5 SLAP versus SLAP-TGTA

The scatter plots of Figure 7.5 compares the performance of SLAP against SLAP-TGTA. The left scatter plot compares the time performance and the right concerns the memory consumption. The points below the diagonal correspond to the cases where SLAP-TGTA is better.

The interpretation of the scatter plots results depends on the colors of the points. For black crosses that correpond to violated formulas, SLAP-TGTA is more efficient than SLAP in most cases. For the green squares representing the verified formulas, the scatter plots are difficult to

		SLAP	SLAP-TGTA
Verified Formulas	Win	981 (50%)	964 (50%)
	Fail	57 (2%)	261 (13%)
Violated Formulas	Win	644 (31%)	1374 (68%)
	Fail	28 (1%)	16 (0%)

Table 7.3: On all successful experiments, we count the number of cases a specific method has (Win) the best time. The Fail line shows the cases were each approach failed to compute the result of an experiment solved by the other approach.

interpret, there are many cases on both sides of the diagonal. In addition, according to Table 7.3, there are roughly the same number of cases where each approach is better than the other. However, Table 7.3 shows that SLAP is clearly better for failed cases. In total, SLAP-TGTA failed in 277 experiments solved by SLAP, while SLAP failed for only 85 cases solved by SLAP-TGTA. In the other cases where the two approaches were successful, we observe a relative advantage for SLAP-TGTA. Indeed, on the one hand, SLAP-TGTA was at least a hundred times faster than SLAP in 85 cases, ten times faster in 646 cases, and twice times faster in 1605 cases. On the other hand, SLAP was at least one hundred times faster than SLAP-TGTA in 12 cases, ten times faster in 95 cases, and twice times faster in 434 cases only.

We believe that the two approaches SLAP and SLAP-TGTA are complementary and very different because of the fact that SLAP-TGTA aggregates are based on changesets and therefore are very different from SLAP aggregates that are based on valuations. Thus, these two approaches can be considered complementary and can be launched in parallel in order to retrieve the result of the fastest approach.

7.7 Conclusion

In this chapter, we proposed three hybrid approaches variants using TGTA: SOG-TGTA, SOP-TGTA and SLAP-TGTA.

SOG-TGTA is a variant of SOG without divergent states. They are replaced in SOG-TGTA by adding a self-loop on each aggregate that contains a cycle. Adding these self-loop is better than adding divergent states for SOG-TGTA because in TGTA all stuttering transitions are self-loops, and therefore adding self-loops in SOG-TGTA does not generate new states in the product between TGTA and SOG-TGTA.

SOP-TGTA is an adaptation of SOP to TGTA. The difference between SOP and SOP-TGTA appears when computing the transitions of the hybrid product. Indeed, the synchronisation of the transitions between the Kripke structure aggregates and the TGTA transitions is based on changesets.

SLAP-TGTA is also a adaptation of SLAP to use TGTA instead of TGBA. The two variants (SLPA and SLAP-TGTA) are based on the aggregation of the states of a Kripke structure according to the self-loops of the formula automaton (i.e., TGBA for SLAP and TGTA for SLAP-TGTA). However, the obtained aggregates are very different between the two variants. In a SLAP-TGTA,

the states of an aggregate change according to the changesets of the TGTA self-loops (instead of satisfying the valuations labeling the TGBA self-loops in the case of SLAP).

We implemented and experimentally compared the performance of each TGTA based hybrid approach (SOG-TGTA, SOP-TGTA and SLAP-TGTA) against its reference variant (SOG, SOP and SLAP). The obtained results show that SOG-TGTA is better than SOG to check verified properties but it is worse for the violated properties. For SOP versus SOP-TGTA, the results are harder to interpret. The results seem balanced with a slight advantage for SOP, especially because we have more cases solved by SOP and failed for SOP-TGTA than otherwise. Finally, for SLAP versus SLAP-TGTA, although we can observe an advantage for SLAP-TGTA for violated formulas, the results remain difficult to interpret. There are many cases where SLAP-TGTA is better and vice versa in many cases SLAP is better. In addition, for verified properties, there are many cases that failed for SLAP-TGTA but successful for SLAP. This large difference in performance between SLAP and SLAP-TGTA can be explained by the fact that the aggregates computed using TGBA and TGTA are very different (in SLAP-TGTA, the aggregates are based on changesets instead of valuations as in SLAP).

More generally, for the three hybrid approachs, we believe that the two variants (using TGBA versus TGTA) are complementary, and the best solution is to run the two variants in parallel, then take the result of the faster one.

Our work presented in this chapter about hybrid approaches using TGTA is not finished. In particular, we must look for an optimization that exploits the fact that TGTA is specific to stutter-invariant properties, as we did in the previous chapter about symbolic approaches, in which we have proposed an optimization based on the “stuttering-normalization constraint” of TGTA.

CHAPTER 8

Conclusion and Perspectives

8.1 Context

The automata-theoretic approach [89, 90] is traditionally used for the model checking of LTL properties. In this approach, a Kripke structure $\mathcal{K}_M$ is used to represent the state-space of the model M. The property to check is expressed as an LTL formula φ, then its negation is converted into an ω-automaton $A_{\neg\varphi}$. The third operation is the synchronization between $\mathcal{K}_M$ and $A_{\neg\varphi}$. This constructs a product automaton $\mathcal{K}_M \otimes A_{\neg\varphi}$ whose language, $\mathcal{L}(\mathcal{K}_M) \cap \mathcal{L}(A_{\neg\varphi})$, is the set of executions of M invalidating φ. The last operation is the emptiness check algorithm that explores the product to tell whether it accepts or not an infinite word, i.e., a counterexample. The model M verifies φ iff $\mathcal{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$.

Problem. The performance of the emptiness check suffers from the well known state-space explosion problem [87]. The product automaton is often too large to be emptiness checked in a reasonable run time and memory (the size of the product can reach $(|A_{\neg\varphi}| \times |\mathcal{K}_M|)$ with $|\mathcal{K}_M|$ often very large).

8.2 Existing Work

In order to improve the performance of the LTL model checking, many works have attempted to build optimized property automaton $A_{\neg\varphi}$, either by improving the LTL translation, or by proposing several reductions for the automaton produced by this translation such as bisimulation/simulation reductions. In most of this works, $A_{\neg\varphi}$ is represented using the traditional variant of Büchi Automata with state-based accepting (BA). However, according to [27, 49], moving the accepting conditions from the states to the transitions reduces the size of $A_{\neg\varphi}$. For instance, *Transition-based Generalized Büchi Automata* (TGBA) is another variant of Büchi Automata that represent the LTL properties more concisely [49, 36] than BA, because TGBA use *generalized* (i.e., multiple) Büchi acceptance conditions *on transitions* rather than on states.

Testing Automata (TA). Hansen et al. [52] propose an alternative to Büchi automata, called Testing Automata (TA), that only represent stutter-invariant properties. According to Geldenhuys and Hansen [46], thanks to their high degree of determinism, the TA allow during the model checking to obtain a smaller product than BA. However, TA have two different ways to accept infinite words (livelock or Büchi), an unfortunate consequence is that the emptiness-check algorithm required must perform two passes on the whole product in the worst case.

Evaluation of the Testing Automata Approach. We experimentally evaluated the performance of the model checking approach using TA against two variants of Büchi automata BA and TGBA. In the benchmark results, we distinguished violated formulas (i.e., when a counterexample is found) from verified formulas (i.e., exhibit no counterexample).

For verified formulas, we found that the product reduction achieved by the TA approach was not enough to compensate for the two-pass emptiness check this approach requires. It is therefore better to use the TGBA approach, which is more efficient than TA and BA to prove that a stutter-invariant formula is verified.

For violated formulas, the TA approach usually processes less transitions in the product than the BA approach to find a counterexample. This is especially true on random formulas. With weak-fairness formulas, TGBA are advantaged by their generalized acceptance conditions and are able to beat the TA on the average in half of examples.

8.3 Contributions

The general objective of this work is to fight against the state-space explosion problem, by reducing the size of the product automaton and/or by decreasing the amount of time and memory used in the emptiness check of this product.

To achieve this goal, we focus on improving the performance of the model checking for stutter-invariant LTL properties. We firstly extend the work of Geldenhuys and Hansen [46] about Testing Automata (TA), by proposing new types of ω-autamata optimized for stutter-invariant LTL properties. These new automata represent all the stuttering-transitions using only self-loops, and (unlike TA) only require a single-pass emptiness check algorithm. Secondly, using our main new type of automata, called *Transition-based Generalized Testing Automata (TGTA)*, we propose contributions to improve the performance of three different approaches, i.e., explicit, symbolic and hybrid model checking approaches, where hybrid means combining explicit and symbolic approaches.

The contributions of this thesis could be summarized as follows:

Single-pass Testing Automata (STA). After improving the emptiness check of TA approach to avoid the second pass in particular cases, we propose STA (*Single-pass Testing Automata*), a transformation of TA in a normal form that never requires such a second pass.

An experimental evaluation shows that these improvements compete well on our benchmarks. Especially, STA remain good for violated properties, and also beat TA and TGBA (and BA) in most cases when properties are verified. Unfortunately, we observe in certain cases that the STA increases the size of the product automaton, because the transformation from TA into STA adds an artificial livelock-accepting state in order to remove the second pass.

Transition-based Generalized Testing Automata (TGTA): A Single-pass and Generalized New Automata. We propose another type of ω-automata for stutter-invariant properties, better than STA, called TGTA (*Transition-based Generalized Testing Automata*). TGTA combine advantages observed on both TA and TGBA, without the second pass of TA, and without adding an artificial state as in STA.

TGTA inherits from TA the elimination of useless stuttering-transitions, but without introducing a second mode of acceptance (i.e, livelock-accepting states). From TGBA, it inherits the use of generalized acceptance conditions on transitions.

We have run benchmarks to evaluate the performance of the TGTA approach against the TA and TGBA (BA) approaches. The experiments report that, in most cases, TGTA lead to smaller products on the average, and the TGTA approach outperforms the TA and TGBA approaches when the property is verified. However, when the property is violated, the results are difficult to interpret because in this case, the on-the-fly algorithm can stop as soon as it finds a counterexample before exploring the entire product. Changing the order in which non-deterministic transitions of the property automaton are iterated is enough to change the number of states and transitions to be explored before a counterexample is found.

Beyond this ordering luckiness in the case of violated formula, we believe that TGTA is better than TA firstly because TGTA does not require a second pass during the emptiness check . Secondly, in our experiments, we observed that TGTA represent more concisely the LTL formulas using (multiple) generalized acceptance conditions, especially for weak-fairness formulas, for which the number of acceptance conditions is greater than other random formulas.

We also believe that TGTA is better than TGBA because the TGTA construction exploits the fact that it is specific to stutter-invariant properties to remove the useless stuttering-transitions, while the TGBA does not exploit at all this specificity. The constructed TGTA represents all the stuttering-transitions using only self-loops, which can reduce the multiplication of stuttering steps in the product. This advantage of TGTA is better exploited in the following symbolic approach.

Using TGTA to improve Symbolic Model Checking. We also use TGTA to improve the symbolic approach.

After showing how to encode a symbolic TGTA, we introduce a new symbolic Kripke structure labeled with changesets on transitions as in TGTA. This changeset-based symbolic Kripke structure simplifies the symbolic transition relation of the product.

The main improvement proposed in this approach is based on the combination of TGTA with saturation technique proposed by Ciardo et al. [20]. We show that the performance of the saturation algorithm greatly benefits from the property of TGTA that all stuttering transitions are self-loops and every state has a stuttering self-loop. In other words, the exploration of stuttering transitions in the product is equivalent to only explore stuttering transitions in the model (remaining in the same TGTA states). This property allowed us to improve the stuttering part in the transition relation of the product, this part is only dependent on the model, and can be evaluated without consulting the transition relation of the TGTA. This allows the saturation algorithm to ignore the symbolic variables encoding the TGTA in this stuttering part of product, and therefore efficiently computes (i.e. saturates) the product nodes corresponding to the variables encoding the model.

We experimentally compared this TGTA-based approach to a symbolic approach based on TGBA and saturation. On our benchmark, using TGTA, we were able to gain one order of magnitude over the TGBA-based approach, for both verified and violated properties.

Using TGTA in three Hybrid Model Checking Approaches. In addition of the explicit and symbolic approaches, we evaluate the use of TGTA in the context of hybrid [79, 38] model check-

ing that combines the use of both explicit and symbolic techniques.

We focus on three hybrid techniques proposed in [37]: the *Symbolic Observation Graph* (SOG), the *Symbolic Observation Product* (SOP) and the *Self-Loop Aggregation Product* (SLAP), and for each of them we propose a variant using TGTA: SOG-TGTA, SOP-TGTA and SLAP-TGTA.

Implementation and experimental evaluation of each TGTA-based variant against the corresponding original approach show that the obtained results depend on whether the formula is verified or violated:

- SOG vs. SOG-TGTA: For verified properties, SOG-TGTA outperforms SOG. For violated properties, the SOG approach outperforms the SOG-TGTA approach in the cases where the execution time is low (less than one second). On the contrary, for hard cases, there are more cases that failed using SOG than using SOG-TGTA.
- SOP vs. SOP-TGTA (the results are harder to interpret): SOP and SOP-TGTA are comparable with a slight advantage for SOP, because we have more cases solved by SOP and failed for SOP-TGTA than otherwise.
- SLAP vs. SLAP-TGTA: For violated properties, SLAP-TGTA outperforms SLAP. For verified properties, the results are more difficult to interpret. There are many cases where SLAP-TGTA is better and vice versa in many cases SLAP is better. We believe that SLAP and SLAP-TGTA are complementary, and the best solution is to run the two variants in parallel, then take the result of the faster one.

8.4 Perspectives

8.4.1 Improving TGTA-based approaches

Several optimizations can be added to TGTA and TA, such as the simulation-reduction [3], which it is currently only implemented for TGBA in Spot. In addition, the optimizations presented in Section 3.4.3 (page 46) for TA can be easily adapted to TGTA.

Another important optimization is to build on-the-fly the TA and the TGTA during the construction of the synchronous product. Especially when the number of atomic propositions (AP) is very large, because this may lead to build a TA or a TGTA with a large number of unnecessary initial states, that are not synchronized with the initial state(s) of the Kripke structure, see for example the product $\mathcal{K} \otimes \mathcal{T}$ presented in Section 5.6 (page 97). In this example, a TGTA $\mathcal{T}$ contains 3 initial states and only one of them is synchronized with the initial state the Kripke structure $\mathcal{K}$. In addition, the optimizations of Section 3.4.3 (already mentioned above) can be easily integrated in a on-the-fly construction of a TA or a TGTA.

As a future work, an idea would be to provide a direct conversion of LTL$\setminus$X to TGTA, without the intermediate TGBA step. We believe a tableau construction such as the one of Couvreur [27] could be easily adapted to produce TGTA. We can also translate an LTL$\setminus$X formula into an intermediate $\emptyset$-TGTA then we apply the stuttering reduction of Property 9 (page 93) to transform the obtained $\emptyset$-TGTA into TGTA.

Another idea is to investigate the use of $\emptyset$-TGTA to improve the model checking of LTL properties (stutter-invariant or not). Indeed, unlike TGTA, the $\emptyset$-TGTA can represent any LTL formula. Furthermore, the procedure proposed in Chapter 5 to build an $\emptyset$-TGTA from a TGBA does not exploit the restriction to stutter-invariant properties and therefore can be used to build an $\emptyset$-TGTA for any LTL formula.

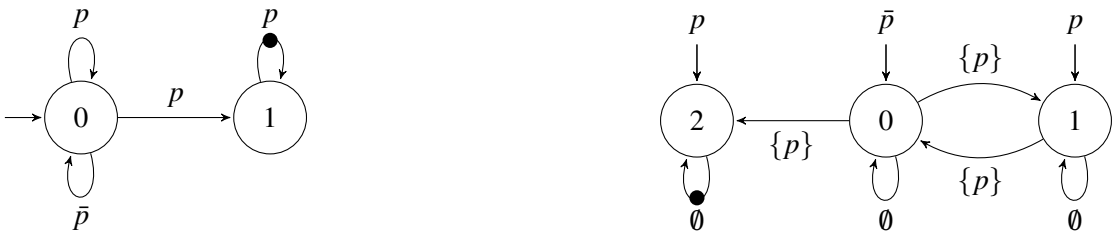
Finally, our work presented in the last chapter about hybrid approaches using TGTA is not finished. In particular, we must look for an optimization that exploits the fact that TGTA is specific to stutter-invariant properties, as we did in the chapter about the symbolic approach, in which we have proposed an optimization based on the stuttering self-loops of TGTA.

8.4.2 Finding sub-classes of LTL formulas for which TGTA is always efficient

An interesting study for TGTA would be to look for a subclass of LTL formulas for which the product of their TGTA with any model will always be smaller than the product with a TGBA or a BA.

We have already started to look for this type of formulas by analyzing the results of our experiments presented in this work. We believe that the subclass of formulas of the form $\varphi = \mathsf{FG}\,p$ is a good candidate. In the following, for $\varphi = \mathsf{FG}\,p$ we illustrate the advantage of TGTA compared to TGBA, in the case of a product with the stuttering parts of a Kripke structure.

Figure 8.1 shows the TGBA and the TGTA for the LTL property $\varphi = \mathsf{FG}\,p$.



(a) A TGBA $\mathcal{A}$ for $\mathsf{FG}\,p$

(b) A TGTA $\mathcal{T}$ for $\mathsf{FG}\,p$

Figure 8.1: A TGBA (left) and a TGTA (right) for the LTL property $\varphi = \mathsf{FG}\,p$.

(a) Kripke Structure $\mathcal{K}_1$ that stutters with p true

(b) Kripke Structure $\mathcal{K}_2$ that stutters with p false

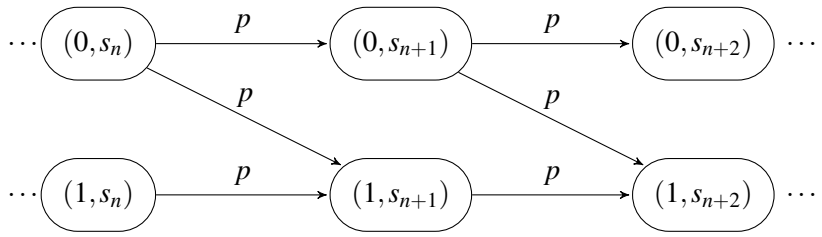
Figure 8.2: The two possible cases of stuttering parts of a Kripke structure.

In order to compare the products, we analyze the two possible cases of stuttering parts of a Kripke structure, represented in Figure 8.2. The first case of Figure 8.4a is a subpart of Kripke structure that stutters with p true. The products of this subpart with the TGBA $\mathcal{A}$ and the TGTA

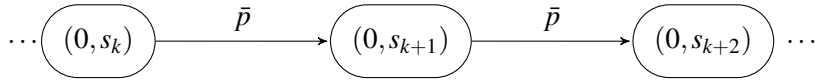
$\mathcal{T}$ are respectively represented in Figure 8.3 and Figure 8.4. In this case, we can observe that the product using TGTA is smaller than the one using TGBA: the (diagonal) transitions from $(0, s_n)$ to $(1, s_{n+1})$ and from $(0, s_{n+2})$ to $(0, s_{n+2})$ in the product using TGBA does not exist in the product using TGTA.

In the second case where the Kripke structure stutters with p false (see Figure 8.4b), the products using TGBA and TGTA are the same.

So overall, for the subclass of formulas of the form $\varphi = \mathsf{FG}\, p$, the stuttering parts of a Kripke structure lead to a smaller parts of product using TGTA than TGBA.

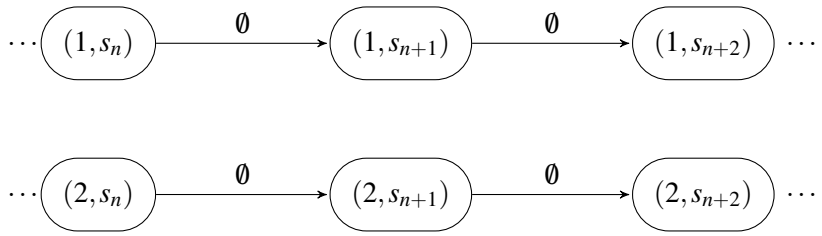


(a) Product of TGBA $\mathcal{A}$ and Kripke Structure $\mathcal{K}_1$ that stutters with p true

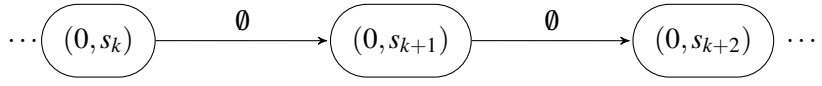


(b) Product of TGBA $\mathcal{A}$ and Kripke Structure $\mathcal{K}_2$ that stutters with p false

Figure 8.3: The two products of the TGBA $\mathcal{A}$ with the two possible cases of stuttering parts of a Kripke structure.



(a) Product of TGTA $\mathcal{T}$ and Kripke Structure $\mathcal{K}_1$ that stutters with p true



(b) Product of TGTA $\mathcal{T}$ and Kripke Structure $\mathcal{K}_2$ that stutters with p false

Figure 8.4: The two products of TGTA $\mathcal{T}$ with the two possible cases of stuttering parts of a Kripke structure.

8.4.3 Combining TGTA with Partial Order Reductions

Finally, another future work is to combine the TGTA with other techniques that propose state-space optimizations specific to stutter-invariant properties, such as the partial order reduction implemented in Spin tool [55].

Several partial order reduction techniques have been proposed, as the *stubborn sets* of Valmari [86], the *persistent sets* of Godefroid [50] and the *ample sets* of Peled [69]. The basic idea of these reductions is to prune the state-space $\mathcal{K}_M$ by identifying equivalent interleaving sequences that only differ by the order of concurrent transitions.

TGTA and partial order reduction are complementary. Indeed, while the TGTA-based approaches focus on optimizing the property automata, the partial order techniques try to reduce the state-space of the model.

Bibliography

[1] André Arnold. *Finite transition systems. Semantics of communicating systems*. Prentice-Hall, 1994. (Cited on page 13.)

[2] Tomáš Babiak, Mojmír Křetínský, Vojtěch Řehák, and Jan Strejček. LTL to Büchi automata translation: Fast and more deterministic. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, volume 7214 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2012. (Cited on page 30.)

[3] Tomáš Babiak, Thomas Badie, Alexandre Duret-Lutz, Mojmír Křetínský, and Jan Strejček. Compositional approach to suspension and other improvements to LTL translation. In *Proceedings of the 20th International SPIN Symposium on Model Checking of Software (SPIN'13)*, volume 7976 of *Lecture Notes in Computer Science*, pages 81–98. Springer, July 2013. (Cited on pages 28, 30, 40, 108, 158 and 173.)

[4] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9. (Cited on page 21.)

[5] Ala Eddine Ben Salem, Alexandre Duret-Lutz, and Fabrice Kordon. Generalized Büchi automata versus testing automata for model checking. In *Procedings of the 2nd workshop on Scalable and Usable Model Checking for Petri Nets and other models of Concurrency (SUMo'11)*, volume 726, pages 65–79, Newcastle, UK, June 2011. CEUR. (Cited on page 40.)

[6] Ala Eddine Ben Salem, Alexandre Duret-Lutz, and Fabrice Kordon. Model Checking using Generalized Testing Automata. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC VI)*, 7400:94–122, 2012. (Cited on pages 47, 88 and 122.)

[7] Ala Eddine Ben Salem, Alexandre Duret-Lutz, Fabrice Kordon, and Yann Thierry-Mieg. Symbolic Model Checking of stutter invariant properties Using Generalized Testing Automata. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 440–454, Grenoble, France, April 2014. Springer. (Cited on page 114.)

[8] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001. URL http://www.springer.com/computer/swe/book/978-3-540-41523-7. (Cited on page 1.)

[9] Béatrice Berard, Laure Millet, Maria Potop-Butucaru, Yann Thierry-Mieg, and Sébastien Tixeuil. Formal verification of Mobile Robot Protocols. Technical report, Laboratoire d'Informatique de Paris 6 - LIP6 , Laboratory of Information, Network and Communication Sciences - LINCS , Institut Universitaire de France - IUF, May 2013. URL http://hal.archives-ouvertes.fr/hal-00834061. (Cited on page 13.)

[10] František Blahoudek, Alexandre Duret-Lutz, Mojmír Křetínský, and Jan Strejček. Is there a best Büchi automaton for explicit model checking? In *Proceedings of the 21th International SPIN Symposium on Model Checking of Software (SPIN'14)*, pages ?–? ACM, July 2014. To appear. (Cited on pages 30 and 40.)

[11] Roderick Bloem. *Search Techniques and Automata for Symbolic Model Checking*. PhD thesis, University of Colorado, 2001. (Cited on pages 28 and 37.)

[12] S. C. C. Blom, J. C. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, Edinburgh*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359, Berlin, July 2010. Springer Verlag. (Cited on pages 122, 124 and 149.)

[13] Stefan Blom and Simona Orzan. Distributed branching bisimulation reduction of state spaces. *Electronic Notes in Theoretical Computer Science*, 89(1):99–113, 2003. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/S1571-0661(05)80099-4. URL http://www.sciencedirect.com/science/article/pii/S1571066105800994. {PDMC} 2003, Parallel and Distributed Model Checking (Satellite Workshop of {CAV} '03). (Cited on page 26.)

[14] Ahmed Bouajjani, Jean-Claude Fernandez, Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992. (Cited on pages 26 and 27.)

[15] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983. (Cited on page 13.)

[16] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, Berkley, 1960*, pages 1–11. Standford University Press, 1962. (Cited on pages 4 and 19.)

[17] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press. (Cited on pages 3, 7, 12, 114 and 115.)

[18] Gianfranco Ciardo and Andy Jinqing Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Proceedings of the 13 IFIP WG 10.5 international conference on Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 146–161. Springer-Verlag, 2005. (Cited on pages 119 and 120.)

[19] Gianfranco Ciardo, Gerarld Lüttgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In Mogens Nielsen and Dan Simpson, editors, *Proceedings of the 21st International Conference on Application and Theory of Petri Nets (ICATPN'00)*, volume 1825 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, 2000. (Cited on page 54.)

[20] Gianfranco Ciardo, Robert M. Marmorstein, and Radu Siminiceanu. Saturation unbound. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 379–393. Springer-Verlag, 2003. (Cited on pages 7, 114, 119 and 157.)

[21] Gianfranco Ciardo, Gerald Lüttgen, and Andrew S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31(1):63–100, 2007. (Cited on page 54.)

[22] Jacek Cichoń, Adam Czubak, and Andrzej Jasiński. Minimal Büchi automata for certain classes of LTL formulas. In *Proceedings of the Fourth International Conference on Dependability of Computer Systems (DEPCOS'09)*, pages 17–24. IEEE Computer Society, 2009. (Cited on page 24.)

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986. ISSN 0164-0925. doi: 10.1145/5397.5399. URL http://doi.acm.org/10.1145/5397.5399. (Cited on page 16.)

[24] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future. *ACM Computing Surveys*, 28(4):626–643, 1996. (Cited on page 1.)

[25] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000. (Cited on page 1.)

[26] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithm for the verification of temporal properties. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of the 2nd international workshop on Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer-Verlag, 1991. (Cited on pages 3 and 12.)

[27] Jean-Michel Couvreur. On-the-fly verification of temporal logic. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, Toulouse, France, September 1999. Springer-Verlag. ISBN 3-540-66587-0. (Cited on pages 24, 30, 33, 37, 155 and 158.)

[28] Jean-Michel Couvreur. Un point de vue symbolique sur la logique temporelle linéaire. In Pierre Leroux, editor, *Actes du Colloque LaCIM 2000*, volume 27 of *Publications du LaCIM*, pages 131–140, Montréal, August 2000. Université du Québec à Montréal. (Cited on page 30.)

[29] Jean-Michel Couvreur, Alexandre Duret-Lutz, and Denis Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In Patrice Godefroid, editor, *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, volume 3639 of *Lecture Notes in Computer Science*, pages 143–158. Springer, August 2005. (Cited on pages 30, 33 and 37.)

[30] Christian Dax, Jochen Eisinger, and Felix Klaedtke. Mechanizing the powerset construction for restricted classes of ω-automata. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *Lecture Notes in Computer Science*. Springer, October 2007. (Cited on page 25.)

[31] Michel Diaz. *Réseaux de Petri, Modèles fondamentaux*. Traité IC2, série Informatique et systèmes d'information. Hermes Science, June 2001. (Cited on page 13.)

[32] Edsger Wybe Dijkstra. EWD 376: Finding the maximum strong components in a directed graph. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF, May 1973. (Cited on pages 36 and 48.)

[33] Edsger Wybe Dijkstra. Finding the maximal strong components in a directed graph. In *A Discipline of Programming*, chapter 25, pages 192–200. Prentice-Hall, 1976. (Cited on pages 36 and 48.)

[34] Alexandre Duret-Lutz. LTL translation improvements in Spot. In *Proceedings of the 5th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'11)*, Electronic Workshops in Computing, Tunis, Tunisia, September 2011. British Computer Society. URL http://ewic.bcs.org/category/15853. (Cited on page 54.)

[35] Alexandre Duret-Lutz. LTL translation improvements in Spot 1.0. *International Journal on Critical Computer-Based Systems*, 5(1/2):31–54, March 2014. (Cited on page 39.)

[36] Alexandre Duret-Lutz and Denis Poitrenaud. SPOT: an extensible model checking library using transition-based generalized Büchi automata. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 76–83. IEEE Computer Society Press, October 2004. (Cited on pages 5, 7, 39, 52, 53, 88, 108, 115, 122, 149 and 155.)

[37] Alexandre Duret-Lutz, Kais Klai, Denis Poitrenaud, and Yann Thierry-Mieg. Combining explicit and symbolic approaches for better on-the-fly LTL model checking. Technical Report 1106.5700, arXiv, June 2011. URL http://arxiv.org/abs/1106.5700. Extended version of our ATVA'11 paper, presenting two new techniques instead of one. (Cited on pages 7, 134, 136, 137, 140, 141, 142, 143 and 158.)

[38] Alexandre Duret-Lutz, Kais Klai, Denis Poitrenaud, and Yann Thierry-Mieg. Self-loop aggregation product — a new hybrid approach to on-the-fly LTL model checking. In *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11)*, volume 6996 of *Lecture Notes in Computer Science*, pages 336–350, Taipei, Taiwan, October 2011. Springer. (Cited on pages 7, 131, 133, 134, 135, 141, 143, 146 and 157.)

[39] Kousha Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer*

Aided Verification (CAV'99), volume 1633 of *Lecture Notes in Computer Science*, pages 236–248. Springer-Verlag, 1999. (Cited on pages 3, 4, 18 and 40.)

[40] Kousha Etessami and Gerard J. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory (Concur'00)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167, Pennsylvania, USA, 2000. Springer-Verlag. (Cited on pages 26, 28, 37 and 46.)

[41] Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Proceedings of the 28th international colloquium on Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 694–707, Crete, Greece, July 2001. Springer-Verlag. (Cited on page 28.)

[42] Berndt Farwer. *Automata logics, and infinite games*, volume 2500 of *Lecture Notes in Computer Science*, chapter ω-automata, pages 3–21. Springer-Verlag, 2002. ISBN 3-540-00388-6. URL http://dl.acm.org/citation.cfm?id=938135.938137. (Cited on pages 2 and 19.)

[43] Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, and Zijiang Yang. Is there a best symbolic cycle-detection algorithm? In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 420–434. Springer-Verlag, 2001. ISBN 3-540-41865-2. (Cited on pages 114, 115, 116 and 117.)

[44] Carsten Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In Oscar H. Ibarra and Zhe Dang, editors, *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA'03)*, volume 2759 of *Lecture Notes in Computer Science*, pages 35–48, Santa Barbara, California, July 2003. Springer-Verlag. ISBN 3-540-40561-5. (Cited on pages 28 and 46.)

[45] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, 2001. Springer-Verlag. (Cited on page 30.)

[46] Jaco Geldenhuys and Henri Hansen. Larger automata and less work for LTL model checking. In *Proceedings of the 13th International SPIN Workshop (SPIN'06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2006. (Cited on pages 5, 6, 33, 40, 44, 45, 47, 48, 52, 54, 61, 64, 70, 75, 85, 155 and 156.)

[47] Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2004. ISBN 3-540-21299-X. (Cited on page 33.)

[48] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th Workshop on Protocol Specification Testing and Verification (PSTV'95)*, pages 3–18, Warsaw, Poland, 1996. Chapman & Hall. URL http://citeseer.nj.nec.com/gerth95simple.html. (Cited on pages 3, 12, 20 and 30.)

[49] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In D.A. Peled and M.Y. Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326, Houston, Texas, November 2002. (Cited on pages 5, 7, 23, 24, 30, 37, 88, 108 and 155.)

[50] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996. ISBN 3540607617. (Cited on pages 4, 37 and 161.)

[51] Alexandre Hamez, Yann Thierry-Mieg, and Fabrice Kordon. Hierarchical set decision diagrams and automatic saturation. In *Proceedings of the 29th international conference on Applications and Theory of Petri Nets*, PETRI NETS '08, pages 211–230. Springer-Verlag, 2008. ISBN 978-3-540-68745-0. doi: 10.1007/978-3-540-68746-7_16. (Cited on pages 119, 120 and 122.)

[52] Henri Hansen, Wojciech Penczek, and Antti Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. In Rance Cleaveland and Hubert Garavel, editors, *Proceedings of the 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (FMICS'02)*, volume 66(2) of *Electronic Notes in Theoretical Computer Science*, Málaga, Spain, July 2002. Elsevier. (Cited on pages 5, 40, 53 and 155.)

[53] Monika Heiner, David Gilbert, and Robin Donaldson. Petri nets for systems and synthetic biology. In *Proceedings of the 8th International School on Formal Methods for the Design of Computer (SFM'08)*, volume 5016 of *Lecture Notes in Computer Science*, pages 215–264. Springer, 2008. (Cited on page 53.)

[54] Gerard J. Holzmann. Software model checking. NATO Summer School, pages 309–355, Marktoberdorf, Germany, August 2000. IOS Press Computer and System Sciences. (Cited on page 1.)

[55] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-321-22862-6. (Cited on pages 4, 13, 16, 37 and 161.)

[56] Jérôme Hugues, Yann Thierry-Mieg, Fabrice Kordon, Laurent Pautet, Soheib Barrir, and Thomas Vergnaud. On the formal verification of middleware behavioral properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, volume 133 of *Electronic Notes in Theoretical Computer Science*, pages 139–157. Elsevier Science Publishers, September 2004. (Cited on page 53.)

[57] Yonit Kesten, Amir Pnueli, and Li on Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedins of the 5th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1998. (Cited on pages 114, 116 and 117.)

[58] Kais Klai and Denis Poitrenaud. MC-SOG: An LTL model checker based on symbolic observation graphs. In *Proceedings of the 29th International Conference on Application and Theory of Petri Nets (ICATPN'08)*, volume 5062 of *Lecture Notes in Computer Science*, pages 288–306, Xi'an, China, June 2008. Springer. (Cited on pages 140, 141, 142 and 143.)

[59] Fabrice Kordon, Alban Linard, Didier Buchs, Maximilien Colange, Sami Evangelista, Kai Lampka, Niels Lohmann, Emmanuel Paviot-Adet, Yann Thierry-Mieg, and Harro Wimmel. Report on the model checking contest at petri nets 2011. *T. Petri Nets and Other Models of Concurrency*, 6:169–196, 2012. (Cited on page 54.)

[60] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983. (Cited on page 4.)

[61] Christof Löding. Efficient minimization of deterministic weak ω-automata. *Information Processing Letters*, 79(3):105–109, 2001. (Cited on page 46.)

[62] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992. ISBN 0-387-97664-7. (Cited on page 16.)

[63] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, October 1966. (Cited on page 25.)

[64] MoVe/LRDE. The Spot home page: `http://spot.lip6.fr`, 2014. (Cited on pages 4, 39, 40 and 52.)

[65] David E. Muller. Infinite sequences and finite machines. In *Proceedings of the 1963 Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, SWCT '63, pages 3–16, Washington, DC, USA, 1963. IEEE Computer Society. doi: 10.1109/SWCT.1963.8. URL `http://dx.doi.org/10.1109/SWCT.1963.8`. (Cited on page 25.)

[66] Denis Oddoux. *Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéaires*. PhD thesis, Universitée Paris 7, Paris, France, December 2003. (Cited on page 28.)

[67] Radek Pelánek. BEEM: benchmarks for explicit model checkers. In *Proceedings of the 14th international SPIN conference on Model checking software*, Lecture Notes in Computer Science, pages 263–267. Springer, 2007. (Cited on pages 124 and 149.)

[68] Radek Pelánek. Properties of state spaces and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(5):443–454, 2008. (Cited on page 61.)

[69] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer-Verlag, 1994. (Cited on pages 4, 37 and 161.)

[70] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, September 1995. (Cited on pages 4 and 18.)

[71] Irfan Pyarali, Marina Spivak, Ron Cytron, and Douglas Clark Schmidt. Evaluating and optimizing thread pool strategies for RT-CORBA. In *Proceeding of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems (LCTES'00)*, pages 214–222. ACM, 2000. (Cited on page 53.)

[72] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959. ISSN 0018-8646. doi: 10.1147/rd.32.0114. URL http://dx.doi.org/10.1147/rd.32.0114. (Cited on page 25.)

[73] Etienne Renault, Alexandre Duret-Lutz, Fabrice Kordon, and Denis Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'13)*, volume 8312 of *Lecture Notes in Computer Science*, pages 668–682. Springer, December 2013. (Cited on page 33.)

[74] Kristin Y. Rozier and Moshe Y. Vardi. LTL satisfiability checking. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'07)*, volume 4595 of *Lecture Notes in Computer Science*, pages 149–167. Springer, 2007. (Cited on page 54.)

[75] Kristin Y. Rozier and Moshe Y. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In *Proceedings of the 17th international conference on Formal methods (FM'11)*, pages 417–431. Springer, 2011. (Cited on pages 114 and 115.)

[76] S. Safra. On the complexity of omega -automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, SFCS '88, pages 319–327, Washington, DC, USA, 1988. IEEE Computer Society. ISBN 0-8186-0877-3. doi: 10.1109/SFCS.1988.21948. URL http://dx.doi.org/10.1109/SFCS.1988.21948. (Cited on page 25.)

[77] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In Nicolas Halbwachs and Lenore Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, Edinburgh, Scotland, April 2005. Springer. (Cited on page 33.)

[78] Roberto Sebastiani and Stefano Tonetta. "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *Lecture Notes in Computer*

Science, pages 126–140, L'Aquila, Italy, October 2003. Springer-Verlag. (Cited on pages 5, 30, 37 and 40.)

[79] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: on hybrid approches for LTL symbolic model checking. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 350–363, Edinburgh, Scotland, UK, July 2005. Springer. (Cited on pages 7, 114, 115, 116, 122, 131, 133 and 157.)

[80] F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic SCC hull algorithms. In *Proc. of FMCAD'02*, volume 2517 of *LNCS*, pages 88–105. Springer, 2002. (Cited on pages 114 and 117.)

[81] Fabio Somenzi and Roderick Bloem. Efficient Büchi automata for LTL formulæ. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 247–263, Chicago, Illinois, USA, 2000. Springer-Verlag. (Cited on pages 28 and 37.)

[82] Robert Tarjan. Depth-first search and linear graph algorithms. In *Conference records of the 12th Annual IEEE Symposium on Switching and Automata Theory*, pages 114–121. IEEE, October 1971. Later republished as [83]. (Cited on page 48.)

[83] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. (Cited on pages 48 and 171.)

[84] Heikki Tauriainen. *Automata and Linear Temporal Logic: Translation with Transition-based Acceptance*. PhD thesis, Helsinki University of Technology, Espoo, Finland, September 2006. (Cited on pages 30 and 33.)

[85] Yann Thierry-Mieg, Denis Poitrenaud, Alexandre Hamez, and Fabrice Kordon. Hierarchical set decision diagrams and regular models. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2009. ISBN 978-3-642-00767-5. doi: 10.1007/978-3-642-00768-2_1. (Cited on pages 7, 114, 115, 122 and 149.)

[86] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets (ICATPN'91)*, volume 618 of *Lecture Notes in Computer Science*, pages 491–515, London, UK, 1991. Springer-Verlag. (Cited on pages 4, 37 and 161.)

[87] Antti Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets 1: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998. (Cited on pages 2, 12 and 155.)

[88] Antti Valmari. Bisimilarity minimization in O($m \log n$) time. In *Proceedings of the 30th International Conference on the Applications and Theory of Petri Nets (ICATPN'09)*, volume 5606 of *Lecture Notes in Computer Science*, pages 123–142. Springer, 2009. ISBN 978-3-642-02423-8. URL http://dx.doi.org/10.1007/978-3-642-02424-5_9. (Cited on pages 26 and 27.)

[89] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Proceedings of the 8th Banff Higher Order Workshop (Banff'94)*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266, Banff, Alberta, Canada, 1996. Springer-Verlag. ISBN 3-540-60915-6. (Cited on pages 2, 12 and 155.)

[90] Moshe Y. Vardi. Automata-theoretic model checking revisited. In *Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'07)*, volume 4349 of *Lecture Notes in Computer Science*, Nice, France, January 2007. Springer. Invited paper. (Cited on pages 2 and 155.)

[91] Ralf Wimmer, Marc Herbstritt, Holger Hermanns, Kelley Strampp, and Bernd Becker. Sigref: a symbolic bisimulation tool box. In *Proceedings of the 4th international conference on Automated Technology for Verification and Analysis*, ATVA'06, pages 477–492, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-47237-1, 978-3-540-47237-7. doi: 10.1007/11901914_35. URL http://dx.doi.org/10.1007/11901914_35. (Cited on pages 26 and 27.)

[92] Pierre Wolper. Constructing automata from temporal logic formulas: A tutorial. In E. Brinksma, H. Hermanns, and J.-P. Katoen, editors, *Proceedings of the FMPA 2000 summer school*, volume 2090 of *Lecture Notes in Computer Science*, pages 261–277, Nijmegen, the Netherlands, July 2000. Springer-Verlag. (Cited on page 28.)

[93] Pierre Wolper, Moshe Y. Vardi, and Aravinda Prasad Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, pages 185–194. IEEE Computer Society Press, 1983. (Cited on page 28.)

APPENDIX A

Experimental Comparison of Explicit approaches using TGBA, TA and TGTA, with TGBA improved using simulation-reduction

Recently, several TGBA optimizations was added to SPOT, especially to apply the simulation-reduction [3] to TGBA. The following tables and scatter plots show the impact of this optimization on the results of our experimental comparison (between TGBA, TA and TGTA) presented in Chapter 5.

Table A.1 and Table A.2 shows how for TGBA, TA and TGTA approaches deal with toy models and random formulas. We omit data for BA since they are always outperformed by TGBA. Table A.5 and Table A.6 show toy models against weak-fairness formulas.

Table A.3 and Table A.4 show the results of the two cases studies against random, weak-fairness, and dedicated formulas issued from the case studies (see Section 3.6.2).

These tables separate cases where formulas are verified from cases where they are violated. In the former (Tables A.1, A.5 and A.3), no counterexample are found and the full state-space had to be explored; in the latter (Tables A.2, A.6 and 5.4) the on-the-fly exploration of the state-space stopped as soon as the existence of a counterexample could be computed.

The column "T_φ" shows the time (in $\frac{1}{100^e}$ of seconds) spent constructing the property automata $A_{\neg\varphi_i}$ from the formulas, this time includes the cost of the simulation-reduction of TGBA. This cost impacts all the other approaches because TGTA, BA and TA (through a BA) are constructed from TGBA.

Figure A.1 and Figure A.2 compare the number of visited transitions when running the emptiness check; plotting TGTA against respectively TGBA and TA.

			Verified properties (no counterexample)							
			Automaton			Full product		Emptiness check		
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	TGBA	avg	9	327	21	2 134 328	8 691 523	2 134 328	8 691 523	13 038
		max	48	2 250	1 937	8 634 463	58 408 155	8 634 463	58 408 155	53 292
	TA	avg	69	3 444	29	2 083 706	8 114 637	3 578 356	13 947 438	22 495
		max	531	31 459	1 960	7 435 026	29 160 195	14 870 052	58 320 390	89 846
	TGTA	avg	57	3 880	30	**1 963 093**	**7 650 369**	**1 963 093**	**7 650 369**	**12 294**
		max	386	42 415	1 971	6 635 199	26 179 377	6 635 199	26 179 377	41 328
Ring6	TGBA	avg	9	273	2	1 043 179	6 279 946	1 043 179	6 279 946	1 473
		max	49	3 550	22	2 849 784	21 495 868	2 849 784	21 495 868	4 633
	TA	avg	54	2 127	8	838 315	4 871 788	1 648 030	9 579 646	2 690
		max	531	24 608	154	2 209 660	12 784 696	4 419 320	25 569 392	6 815
	TGTA	avg	44	2 315	8	**823 663**	**4 921 456**	**823 663**	**4 921 456**	**1 361**
		max	386	31 457	180	2 084 839	12 563 232	2 084 839	12 563 232	3 357
FMS5	TGBA	avg	7	188	4	1 655 712	10 797 107	1 655 712	10 797 107	2 196
		max	22	1 651	286	8 839 019	73 557 865	8 839 019	73 557 865	15 448
	TA	avg	41	1 360	8	1 182 686	9 148 797	1 278 218	9 880 397	2 444
		max	282	11 609	295	5 286 393	41 375 144	6 367 350	50 679 714	11 616
	TGTA	avg	33	1 469	8	**1 138 010**	**8 829 003**	**1 138 010**	**8 829 003**	**2 085**
		max	255	13 384	301	5 286 393	41 375 144	5 286 393	41 375 144	9 606
Kanban5	TGBA	avg	6	121	1	2 585 110	20 573 569	2 585 110	20 573 569	3 344
		max	48	1 994	11	17 706 640	149 153 508	17 706 640	149 153 508	22 039
	TA	avg	38	1 308	3	1 895 866	17 696 106	1 895 866	17 696 106	3 562
		max	264	18 440	80	10 558 520	102 370 471	10 558 520	102 370 471	19 446
	TGTA	avg	31	1 398	3	**1 802 404**	**16 858 757**	**1 802 404**	**16 858 757**	**3 244**
		max	220	22 092	89	9 345 280	91 147 301	9 345 280	91 147 301	17 228
Philo10	TGBA	avg	11	500	5	3 970 577	20 450 114	3 970 577	20 450 114	6 638
		max	73	5 397	233	17 947 837	85 727 092	17 947 837	85 727 092	29 036
	TA	avg	61	3 397	20	2 012 055	16 929 172	2 012 055	16 929 172	6 067
		max	404	51 034	475	7 557 069	65 133 806	7 557 069	65 133 806	22 797
	TGTA	avg	48	3 767	23	**1 797 978**	**15 101 356**	**1 797 978**	**15 101 356**	**5 338**
		max	294	48 954	444	6 660 936	54 496 416	6 660 936	54 496 416	19 902
Robin15	TGBA	avg	9	357	4	1 847 692	16 517 326	1 847 692	16 517 326	5 022
		max	50	3 284	61	7 745 523	77 343 545	7 745 523	77 343 545	23 212
	TA	avg	65	3 881	13	1 468 280	11 959 714	2 829 126	23 057 049	8 151
		max	462	34 568	137	6 544 384	52 510 720	13 088 768	105 021 440	35 548
	TGTA	avg	57	4 518	14	**1 492 636**	**12 399 229**	**1 492 636**	**12 399 229**	**4 343**
		max	369	40 112	133	6 556 672	54 543 361	6 556 672	54 543 361	18 743

Table A.1: Comparison of the three approaches on toy examples with **random formulæ, when counterexamples do not exist.**

			Violated properties (a counterexample exists)								
			Automaton			Full product		Emptiness check			
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	TGBA	avg	11	470	4	6 875 386	29 707 327	556 882	2 059 495	3 308
		max	47	2 630	132	35 221 089	164 860 104	3 266 077	13 122 565	19 494
	TA	avg	77	4 616	15	7 311 696	28 594 154	531 367	1 847 946	3 303
		max	688	62 540	151	33 610 951	130 319 612	3 034 231	10 910 727	18 973
	TGTA	avg	58	4 529	16	**6 724 275**	**26 406 241**	**519 338**	**1 814 387**	**3 171**
		max	377	39 300	156	32 323 597	126 519 495	3 241 584	11 733 133	19 719
Ring6	TGBA	avg	12	443	28	2 366 645	15 175 961	423 272	2 286 638	536
		max	51	2 636	2038	7 710 378	56 195 862	2 543 904	14 240 224	3 351
	TA	avg	96	5 312	37	2 181 104	13 018 794	316 826	1 785 138	546
		max	435	36 434	2069	7 791 264	47 132 088	1 813 390	10 805 447	3 150
	TGTA	avg	76	5 389	38	**1 962 430**	**12 050 193**	**304 280**	**1 732 216**	**476**
		max	419	39 766	2063	6 516 704	40 390 528	1 709 454	10 552 980	2 790
FMS5	TGBA	avg	10	321	5	7 030 618	54 559 769	648 095	2 887 241	649
		max	84	3 034	236	28 022 123	252 921 593	9 048 245	63 249 152	13 067
	TA	avg	68	2 903	11	6 336 121	51 452 459	523 842	3 433 001	894
		max	396	20 095	281	24 916 116	214 008 832	7 336 262	61 123 574	14 439
	TGTA	avg	55	2 963	11	**5 726 926**	**47 023 062**	**389 889**	**2 490 868**	**633**
		max	337	20 595	301	23 351 734	205 586 222	6 594 559	54 402 238	12 704
Kanban5	TGBA	avg	8	169	3	7 389 200	65 265 886	398 291	2 413 326	403
		max	25	1 254	73	20 380 864	249 816 728	4 154 893	40 027 194	6 514
	TA	avg	50	1 655	6	6 099 926	58 043 214	279 149	2 293 392	467
		max	271	15 866	81	20 138 608	208 914 286	2 877 959	27 297 928	5 585
	TGTA	avg	41	1 754	7	**5 664 045**	**54 798 536**	**255 545**	**2 095 285**	**410**
		max	228	18 303	80	15 816 808	170 535 764	2 877 959	27 297 927	5 294
Philo10	TGBA	avg	9	389	2	14 648 744	127 225 739	746 849	3 135 235	1 108
		max	52	3 832	20	67 346 113	944 250 350	9 032 250	43 945 324	14 928
	TA	avg	71	3 877	11	12 807 603	115 165 982	514 894	3 437 571	1 324
		max	272	34 438	103	53 294 292	530 498 041	7 342 016	64 470 778	22 435
	TGTA	avg	58	4 118	12	**11 611 005**	**105 837 118**	**434 875**	**2 948 356**	**1 098**
		max	268	35 598	112	55 198 542	558 520 250	3 622 345	30 490 183	11 529
Robin15	TGBA	avg	14	543	16	4 072 114	37 723 236	514 761	5 099 096	1 506
		max	50	3 832	529	12 193 720	166 810 859	10 485 865	153 206 175	41 271
	TA	avg	122	7 708	28	3 606 864	29 540 049	473 503	3 870 325	1 446
		max	799	55 662	548	8 984 576	81 599 488	7 108 784	66 618 546	22 451
	TGTA	avg	98	8 057	29	**3 390 389**	**28 308 511**	**489 600**	**4 058 013**	**1 406**
		max	485	64 827	561	9 922 560	91 222 016	8 046 768	76 224 690	24 848

Table A.2: Comparison of the three approaches on toy examples with **random formulæ, when counterexamples exist.**

				Verified properties (no counterexample)								
				Automaton			Full product		Emptiness check			
				st.	tr.	T_φ	st.	tr.	st.	tr.	T	
PolyORB 3/3/2	RND	TGBA	avg	10	429	20	85 195	207 750	85 195	207 750	425	
			max	50	3 284	1 098	238 118	1 127 918	238 118	1 127 918	1 248	
		TA	avg	67	3 787	29	78 382	168 410	128 674	276 738	655	
			max	435	34 568	1 137	202 874	436 381	405 748	872 762	2 017	
		TGTA	avg	55	4 045	31	**81 656**	**176 671**	**81 656**	**176 671**	**413**	
			max	323	30 547	1 160	221 779	485 262	221 779	485 262	1 090	
	WFair	TGBA	avg	3	42	1	73 879	168 772	73 879	168 772	386	
			max	18	292	5	181 518	619 068	181 518	619 068	968	
		TA	avg	67	850	1	95 006	204 347	95 623	205 667	508	
			max	353	5 953	11	287 185	630 231	287 185	630 231	1 499	
		TGTA	avg	29	420	1	**70 158**	**150 547**	**70 158**	**150 547**	**370**	
			max	99	1 700	4	132 396	286 698	132 396	286 698	698	
	Φ_1	TGBA	–	4	208	2	221 916	496 626	221 916	496 626	1 070	
		TA	–	68	5 677	3	**219 288**	**478 950**	438 576	957 900	2 088	
		TGTA	–	67	7 556	5	221 952	488 136	**221 952**	**488 136**	**1 052**	
MAPK 8	RND	TGBA	avg	9	246	2	2 598 503	22 108 346	2 598 503	22 108 346	4 833	
			max	64	3 551	16	11 272 599	120 406 938	11 272 599	120 406 938	22 707	
		TA	avg	42	1 651	8	1 608 896	18 824 425	1 608 896	18 824 425	4 583	
			max	336	18 971	157	8 036 021	100 718 455	8 036 021	100 718 455	23 124	
		TGTA	avg	35	1 876	9	**1 519 337**	**17 816 286**	**1 519 337**	**17 816 286**	**4 227**	
			max	317	25 595	211	8 036 021	100 718 455	8 036 021	100 718 455	21 791	
	WFair	TGBA	avg	4	31	1	3 461 837	28 114 156	3 461 837	28 114 156	5 572	
			max	21	160	3	10 871 182	104 579 823	10 871 182	104 579 823	20 539	
		TA	avg	42	386	1	1 961 055	22 640 584	1 961 055	22 640 584	5 129	
			max	167	1 902	4	6 110 748	75 624 744	6 110 748	75 624 744	17 114	
		TGTA	avg	19	203	1	**1 777 295**	**20 615 811**	**1 777 295**	**20 615 811**	**4 550**	
			max	80	1 052	3	9 523 917	118 607 814	9 523 917	118 607 814	26 652	
	Φ_2	TGBA	–	5	152	0	46 493	325 442	46 493	325 442	**51**	
		TA	–	7	210	2	**33 376**	**291 602**	**33 376**	**291 602**	58	
		TGTA	–	6	343	3	**33 376**	**291 602**	**33 376**	**291 602**	55	

Table A.3: Comparison of the three approaches for the **case studies when counterexamples do not exist.**

				Violated properties (a counterexample exists)								
				Automaton			Full product		Emptiness check			
				st.	tr.	T_φ	st.	tr.	st.	tr.	T	
PolyORB 3/3/2	RND	TGBA	avg	10	503	6	113 778	281 259	57 622	148 161	289	
			max	56	4 470	224	772 777	1 787 484	279 266	851 459	1 406	
		TA	avg	60	4 350	24	118 209	256 484	56 406	126 308	289	
			max	472	46 228	368	1 005 017	2 159 320	208 726	487 016	1 037	
		TGTA	avg	50	5 072	27	**114 499**	**249 711**	**47 468**	**105 349**	**242**	
			max	310	55 353	520	758 720	1 627 829	105 707	240 336	532	
	WFair	TGBA	avg	3	41	1	78 931	177 210	63 721	144 633	333	
			max	13	203	6	187 020	533 673	169 655	476 766	885	
		TA	avg	71	948	1	180 036	386 413	107 127	234 966	566	
			max	204	3 205	6	483 707	1 034 329	245 740	539 608	1 274	
		TGTA	avg	30	452	1	**78 297**	**167 718**	**63 504**	**138 717**	**334**	
			max	102	1 719	5	186 891	399 306	169 407	366 693	876	
MAPK 8	RND	TGBA	avg	9	321	3	20 240 822	245 726 563	730 667	3 534 127	894	
			max	41	3 012	68	77 833 929	1 306 979 302	6 547 219	51 103 385	12 377	
		TA	avg	62	3 294	12	17 057 208	219 712 950	425 966	2 967 248	872	
			max	314	26 248	163	81 988 164	1 146 895 628	4 859 218	31 565 707	8 378	
		TGTA	avg	50	3 376	14	**15 139 419**	**196 274 725**	**424 117**	**2 957 187**	**829**	
			max	229	29 261	195	47 574 695	644 775 125	4 859 218	31 565 707	8 192	
	WFair	TGBA	avg	4	52	1	15 189 697	191 231 582	1 049 014	6 622 077	1 405	
			max	13	221	3	53 575 313	764 671 391	7 387 837	64 113 702	12 094	
		TA	avg	76	986	2	22 798 869	291 635 140	638 187	5 996 662	1 444	
			max	245	3 322	5	64 763 487	858 891 872	5 203 746	59 642 224	13 725	
		TGTA	avg	34	503	1	**12 475 839**	**160 851 874**	**545 160**	**5 000 866**	**1 183**	
			max	87	1 536	4	47 723 696	662 295 506	4 859 666	47 087 918	10 019	

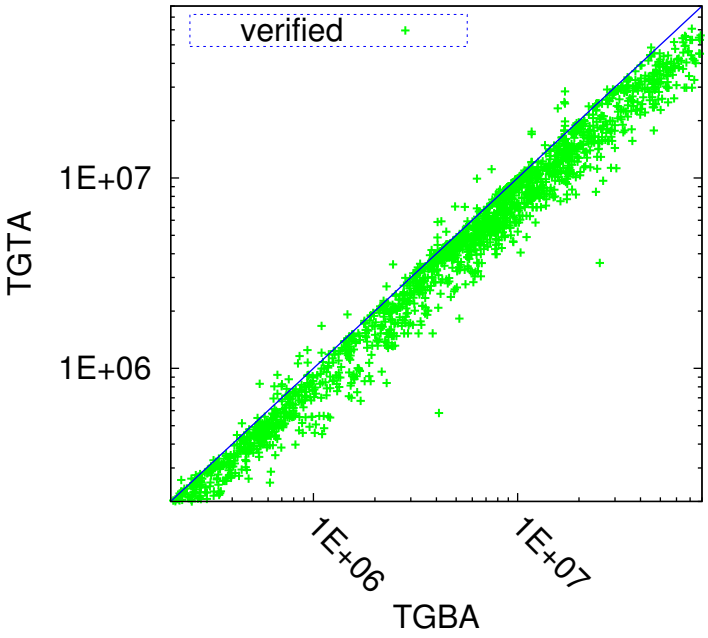
Table A.4: Comparison of the three approaches for the **case studies when counterexamples exist.**

			Verified properties (no counterexample)							
			Automaton			Full product		Emptiness check		
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	TGBA	avg	3	32	1	2334992	9214963	2334992	9214963	14219
		max	9	126	4	6197238	28678473	6197238	28678473	37883
	TA	avg	44	453	1	2594244	10057164	2915489	11303640	18390
		max	173	2360	4	10632587	41615545	10632587	41615545	66358
	TGTA	avg	22	291	1	**2205610**	**8569393**	**2205610**	**8569393**	**13744**
		max	72	1103	5	5458289	21093007	5458289	21093007	33731
Ring6	TGBA	avg	4	38	1	885670	5085387	885670	5085387	1309
		max	16	118	4	3095400	16216920	3095400	16216920	3934
	TA	avg	49	516	1	919252	5187969	1452203	8196805	2454
		max	149	2382	5	2120000	12695504	3565136	20381440	5234
	TGTA	avg	26	347	1	**725434**	**4184918**	**725434**	**4184918**	**1256**
		max	73	1208	5	1300256	7556256	1300256	7556256	2432
FMS5	TGBA	avg	4	33	1	2622090	18893040	2622090	18893040	3733
		max	11	177	2	8685055	78815271	8685055	78815271	14741
	TA	avg	46	483	1	2045253	15834452	2045253	15834452	3947
		max	167	4256	3	7504266	60774774	7504266	60774774	14895
	TGTA	avg	22	265	1	**1769028**	**13854565**	**1769028**	**13854565**	**3291**
		max	87	1430	4	6949551	56088708	6949551	56088708	14382
Kanban5	TGBA	avg	3	27	1	2679040	22247651	2679040	22247651	3464
		max	21	168	3	9183294	87988390	9183294	87988390	12204
	TA	avg	33	277	1	1887122	17126840	1887122	17126840	3506
		max	136	1901	3	8429960	80660846	8429960	80660846	13034
	TGTA	avg	17	177	1	**1711068**	**15682969**	**1711068**	**15682969**	**3072**
		max	62	912	3	6965000	65796520	6965000	65796520	11364
Philo10	TGBA	avg	3	28	1	3980003	23939240	3980003	23939240	7599
		max	11	154	2	12974557	112561242	12974557	112561242	31701
	TA	avg	45	474	1	3072870	26224843	3072870	26224843	9283
		max	289	8154	6	10987384	94141317	10987384	94141317	31547
	TGTA	avg	20	239	1	**2291618**	**19611203**	**2291618**	**19611203**	**6750**
		max	116	3303	3	9571804	85885107	9571804	85885107	28044
Robin15	TGBA	avg	3	35	1	1362987	11158101	1362987	11158101	3736
		max	10	150	4	3059706	28950456	3059706	28950456	8801
	TA	avg	46	489	1	1235467	9824773	1453687	11553559	4482
		max	185	2558	5	2684928	21341184	4696064	37017600	13856
	TGTA	avg	24	323	1	**1126156**	**9031138**	**1126156**	**9031138**	**3450**
		max	92	1334	5	2211840	17928192	2211840	17928192	6866

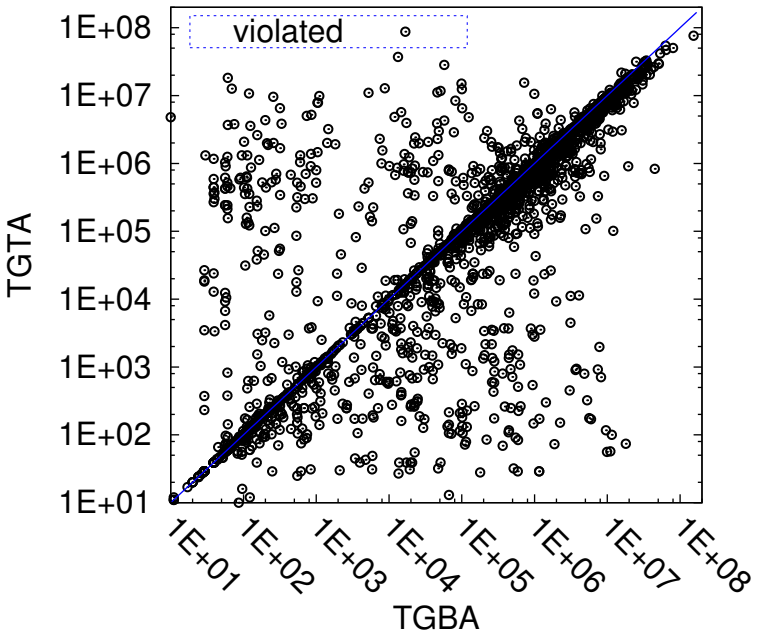
Table A.5: Comparison of the three approaches on toy examples with **weak-fairness formulæ, when counterexamples do not exist.**

			Violated properties (a counterexample exists)								
			Automaton			Full product		Emptiness check		
			st.	tr.	T_φ	st.	tr.	st.	tr.	T
Peterson5	TGBA	avg	4	48	1	**2 993 058**	**11 761 446**	471 940	1 504 950	**2 793**
		max	18	292	8	6 711 952	32 894 473	2 933 653	10 800 747	17 392
	TA	avg	81	1 156	1	9 841 905	38 327 384	567 949	1 843 300	3 488
		max	352	6 001	11	15 973 169	62 098 442	4 444 487	17 106 565	27 499
	TGTA	avg	34	534	1	3 055 186	11 931 357	**469 284**	**1 501 463**	2 828
		max	117	2 108	9	6 405 145	24 914 480	2 898 184	10 706 849	17 570
Ring6	TGBA	avg	5	68	1	1 429 086	8 845 224	254 444	1 395 698	364
		max	28	482	7	3 032 664	21 798 322	1 416 696	7 920 367	2 086
	TA	avg	107	1 477	2	2 871 519	16 768 200	309 870	1 710 585	526
		max	322	6 456	12	5 107 450	30 709 648	2 397 874	13 900 223	4 221
	TGTA	avg	44	697	2	**1 193 721**	**7 025 538**	**215 638**	**1 222 837**	**364**
		max	178	3 687	9	2 501 240	14 944 456	1 169 357	6 706 261	2 103
FMS5	TGBA	avg	4	41	1	6 039 757	48 073 683	684 361	3 016 021	660
		max	14	200	4	20 600 805	163 375 160	6 676 584	53 451 994	9 502
	TA	avg	62	719	1	8 698 172	69 734 251	494 830	3 078 804	799
		max	245	3 136	5	25 112 720	206 558 998	6 425 987	51 962 113	11 444
	TGTA	avg	29	393	1	**4 733 810**	**38 333 115**	**403 379**	**2 445 137**	**616**
		max	94	1 439	4	13 122 252	109 888 666	5 062 934	41 766 785	8 867
Kanban5	TGBA	avg	4	42	1	5 411 516	48 648 241	615 574	3 554 349	606
		max	15	161	3	17 194 688	155 902 603	7 950 633	80 041 971	10 547
	TA	avg	60	701	1	6 955 248	65 524 457	319 238	2 608 392	532
		max	208	3 163	4	19 860 365	194 830 038	5 321 598	52 675 827	9 208
	TGTA	avg	28	380	1	**4 023 776**	**38 395 802**	**287 093**	**2 354 941**	**457**
		max	85	1 228	4	10 822 336	106 435 273	5 049 942	50 190 351	8 374
Philo10	TGBA	avg	4	45	1	8 488 406	70 073 588	624 958	2 345 810	837
		max	12	164	4	21 620 277	175 367 424	4 214 484	21 334 294	6 487
	TA	avg	71	858	2	14 875 690	131 208 038	573 945	3 466 645	1 315
		max	248	3 614	6	42 242 159	375 507 068	6 494 793	50 994 037	14 508
	TGTA	avg	33	456	1	**6 888 581**	**61 271 105**	**394 840**	**2 161 711**	**837**
		max	94	1 431	4	16 961 885	150 416 553	2 787 217	21 746 406	6 346
Robin15	TGBA	avg	5	76	2	2 822 635	25 680 929	439 215	3 708 242	1 215
		max	28	482	7	8 204 253	114 291 493	3 244 925	30 724 614	9 443
	TA	avg	112	1 610	3	4 439 903	35 584 823	437 337	3 439 416	1 347
		max	361	7 372	12	16 381 952	141 807 616	3 174 178	25 462 895	10 666
	TGTA	avg	46	761	2	**2 333 623**	**18 914 483**	**364 484**	**2 908 781**	**1 109**
		max	178	3 687	9	6 628 864	56 228 864	2 678 258	21 824 221	8 646

Table A.6: Comparison of the three approaches on toy examples with **weak-fairness formulæ, when counterexamples exist.**

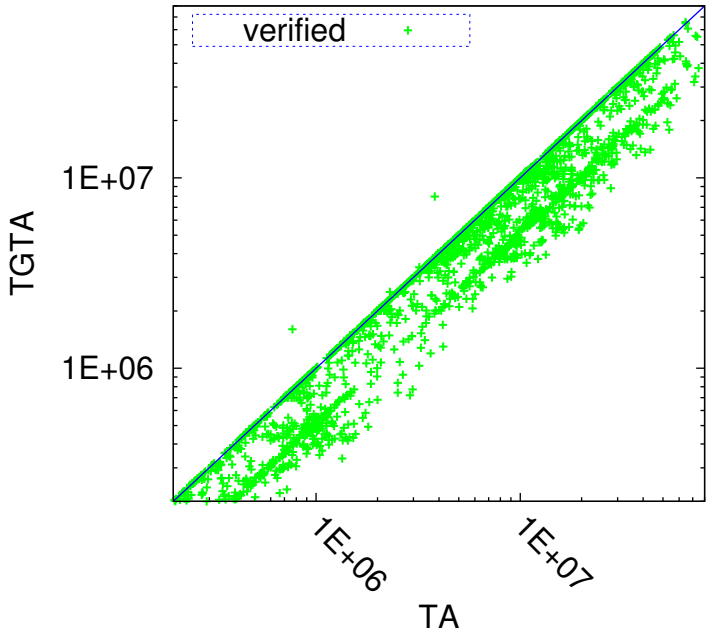


(a) TGTA against TGBA approaches for verified properties

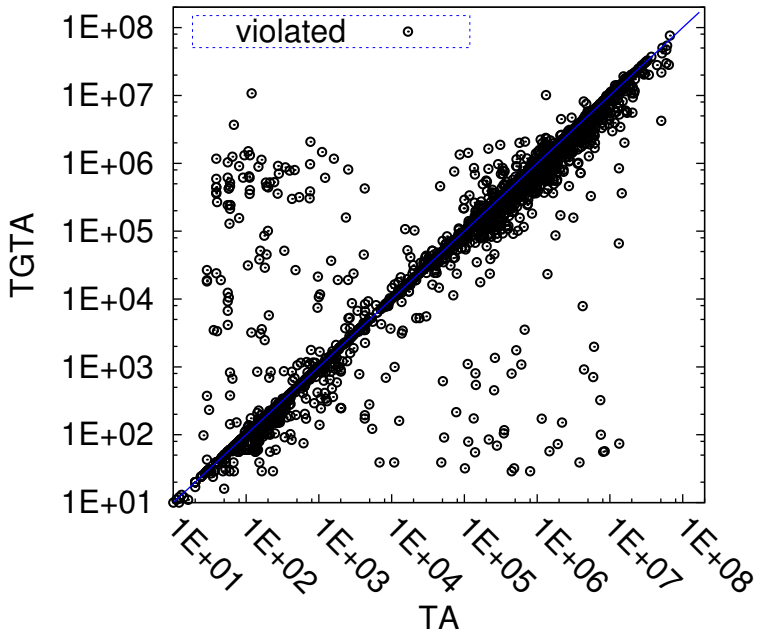


(b) TGTA against TGBA approaches for violated properties

Figure A.1: Performance of TGTA against TGBA, with TGBA improved using simulation-reduction.



(a) TGTA against TA approaches for verified properties



(b) TGTA against TA approaches for violated properties

Figure A.2: Performance of TGTA against TA (with TGBA improved using simulation-reduction).

Part IV

Summary in French

Résumé

1 Contexte

Les systèmes logiciels sont devenus omniprésents dans notre quotidien, ces systèmes se substituent à l'homme pour des tâches délicates, souvent critiques, mettant en jeu des coûts importants, mais aussi des vies humaines. C'est le cas dans les domaines suivants: les équipements médicaux, la téléchirurgie, les réacteurs nucléaires, l'aéronautique, etc.
Ces domaines d'applications démontrent la gravité des conséquences provoquées par les défaillances des ces systèmes, et imposent la recherche de méthodes rigoureuses pour leur validation. La vérification formelle propose des méthodes mathématiques pour garantir la correction d'un système par rapport à certains comportements spécifiés, comme des propriétés de sûreté, de vivacité, etc... Parmi ces méthodes, le model checking est une technique de vérification, qui a la particularité d'être complètement automatique. En effet, prenant en entrée un modèle décrivant toutes les exécutions possibles du système et une propriété exprimée sous la forme d'une formule de logique temporelle, un algorithme de model checking répond si le modèle satisfait ou non la formule. Lorsque la propriété n'est pas satisfaite, le model checker exhibe un contre-exemple (i.e. une exécution du modèle qui invalide la propriété). Dans ce travail, on s'intéresse uniquement aux systèmes à ensemble d'états fini, c'est-à-dire des systèmes pouvant atteindre un nombre fini de configurations. Cette restriction est due au fait que l'algorithme de model checking doit explorer l'ensemble des états accessibles du système. On distingue deux grandes approches de model-checking: explicite et symbolique.

1.1 Model checking explicite

Cette approche est basée sur l'exploration explicite de l'espace des états accessibles. Pour cela, le modèle et la propriété à vérifier sont transformés en automates, en respectant les étapes suivantes (figure 1) :

1. Génération de l'espace d'états du modèle M sous la forme d'une structure de Kripke (une variante d'ω-automate) $\mathcal{K}_M$ dont le langage $\mathscr{L}(\mathcal{K}_M)$ décrit toutes les exécutions possibles de M.

2. Traduction de la propriété φ à vérifier en formule de logique temporelle LTL, puis construction de l' ω-automate $A_{\neg\varphi}$ représentant la négation de la formule LTL obtenue. Le langage $\mathscr{L}(A_{\neg\varphi})$ de cet automate représente l'ensemble des exécutions qui invalident φ.

3. Calcul du produit synchronisé $\mathcal{K}_M \otimes A_{\neg\varphi}$ entre les deux automates $\mathcal{K}_M$ et $A_{\neg\varphi}$ obtenus lors des deux étapes précédentes. Ce produit est également un automate dont le langage $\mathscr{L}(\mathcal{K}_M \otimes A_{\neg\varphi})$ est égal à l'intersection $\mathscr{L}(\mathcal{K}_M) \cap \mathscr{L}(A_{\neg\varphi})$. Ainsi, cette opération nous permet d'obtenir l'ensemble des exécutions de M qui invalident φ.

4. Enfin, pour conclure que le modèle M vérifie φ, il suffit de tester si le langage de l'automate du produit est vide (i.e. $\mathscr{L}(\mathcal{K}_M \otimes A_{\neg\varphi}) = \emptyset$). Ce test de vacuité (emptiness check) est traditionnellement ramené à une exploration explicite du produit afin de vérifier s'il accepte ou non un mot infini. Dans le cas où un mot est accepté, l'algorithme d'emptiness check le retourne comme un contre-exemple.

1.2 Model checking symbolique

Cette seconde approche représente à la fois le modèle et la propriété sous la forme de diagrammes de décisions binaire (BDD). Le but est de ne pas représenter explicitement tous les états du système, mais de calculer des représentations symboliques plus compactes et ainsi réduire l'espace d'état à analyser. En effet, grâce à la représentation du modèle à l'aide de BDD, les états sont explorés de façon ensembliste, par groupe d'états plutôt qu'individuellement.

1.3 Problématique et objectif de la thèse

Le problème principal du model-checking est l'explosion combinatoire de l'espace d'états du modèle à vérifier, notamment pour les systèmes complexes. Par exemple, la modélisation d'un système concurrent peut générer un

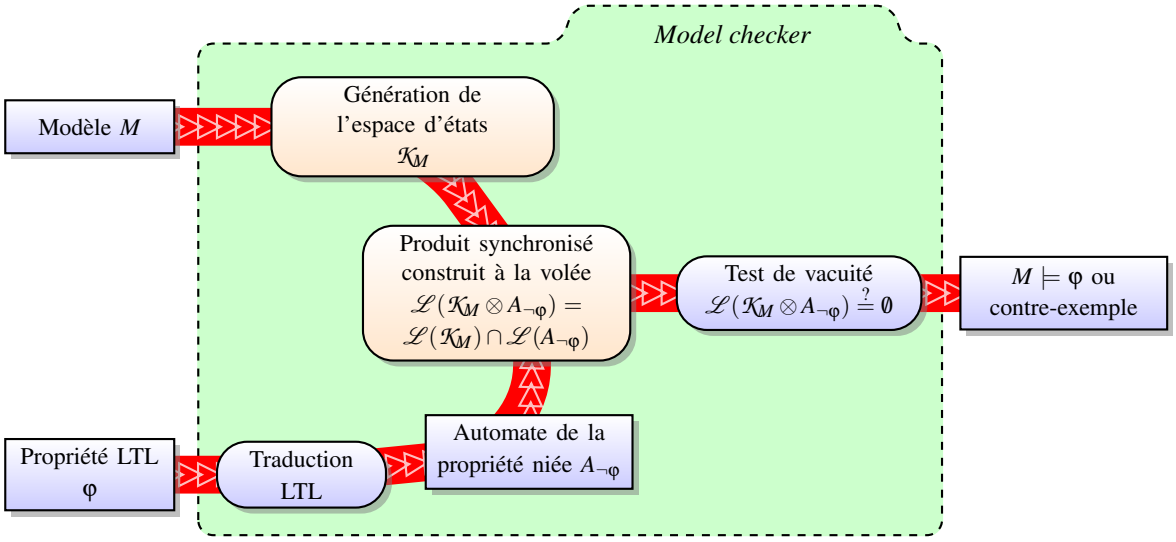


Figure 1: L'approche par automates du *model checking* de formules LTL.

nombre d'états très élevée, même si chaque composant (processus) du système ne peut avoir qu'un petit nombre d'états. Le produit entre l'automate de la formule et l'espace d'états du modèle est encore plus grand puisque il introduit un deuxième niveau de combinatoire. En effet, pour générer les états du produit, chaque état du modèle peut être combiné avec plusieurs états de l'automate de la formule.
Ainsi, la taille et surtout le déterminisme de l'automate de la formule est très important pour remédier au problème d'explosion combinatoire du produit, notamment dans l'approche explicite de model-checking qui nécessite l'exploration de l'automate produit. L'objectif principal de ce travail de thèse est d'améliorer les performances du model checking, dans le cas de la vérification des propriétés insensibles au bégaiement. Afin d'atteindre cet objectif, nous avons apporté quelques contributions, permettant essentiellement de réduire la taille du produit $\mathcal{K}_M \otimes A_{\neg\varphi}$ et de diminuer le temps d'exécution et la quantité de mémoire utilisés au cours de l'emptiness check de ce produit.

Dans ce rapport, après un bref état de l'art des approches classiques, nous détaillons nos contributions, ainsi que les résultats des évaluations expérimentales de ces contributions.

2 État de l'art

Cette section fournit un bref résumé de deux techniques classiques de model checking, basées respectivement sur deux variantes des automates de Büchi TGBA [17] et BA [5] et une troisième approche alternative basée sur les automates testeurs TA [19].

2.1 Préliminaires

Notons AP l'ensemble des propositions atomiques permettant de décrire les propriétés du modèle. Ainsi, la propriété LTL à vérifier est composée des propositions appartenant à AP, des connecteurs booléens $\wedge$ et $\neg$, et des opérateurs temporels X (*next*), U (*until*), ...
Chaque état du modèle est étiqueté par une valuation, représentant l'ensemble des propositions atomiques prenant la valeur booléenne vraie dans cet état. Nous désignons par $\Sigma = 2^{AP}$ l'ensemble de ces valuations, que nous interprétons soit comme un ensemble, soit en tant que conjonctions booléennes. Par exemple, si $AP = \{a,b\}$, alors $\Sigma = 2^{AP} = \{\{a,b\},\{a\},\{b\},\emptyset\}$, ou nous pouvons l'interpréter comme $\Sigma = \{ab, a\bar{b}, \bar{a}b, \bar{a}\bar{b}\}$. Une exécution du modèle est tout simplement une séquence infinie de telles valuations, c'est-à-dire un élément de Σ^ω.

De même, une propriété $\mathcal{P}$ peut être vu comme un sous-ensemble de Σ^ω, contenant les séquences de valuations satisfaisant $\mathcal{P}$. Parmi ces propriétés, nous voulons distinguer celles qui sont insensibles au bégaiement:

Définition 1 *Une propriété (ou un langage), i.e. un ensemble de séquences $\mathcal{P} \subseteq \Sigma^\omega$, est insensible au bégaiement si et seulement si toute séquence $k_0k_1k_2\ldots \in \mathcal{P}$ reste dans $\mathcal{P}$ après duplication de n'importe quelle valuation k_i et*

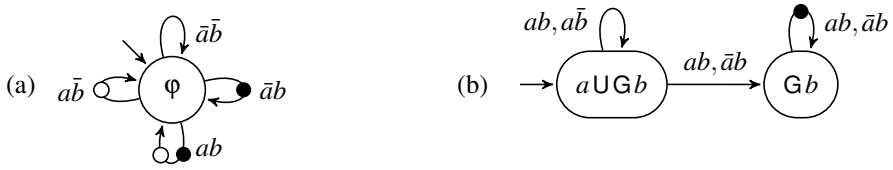


Figure 2: (a) TGBA de la propriété LTL $\varphi = \mathsf{GF}a \wedge \mathsf{GF}b$ avec conditions d'acceptation $\mathcal{F} = \{\bullet, \circ\}$. (b) Un TGBA de la propriété LTL $a\mathsf{U}\mathsf{G}b$ avec $\mathcal{F} = \{\bullet\}$.

inversement après suppression de n'importe quelle répétition. Formellement, $\mathcal{P}$ est insensible au bégaiement si et seulement si:

$$k_0 k_1 k_2 \ldots \in \mathcal{P} \iff k_0^{i_0} k_1^{i_1} k_2^{i_2} \ldots \in \mathcal{P} \text{ pour tous } i_0 > 0, i_1 > 0 \ldots$$

Théorème 1 *Toute formule LTL$\setminus$X (i.e. une formule LTL qui n'utilise pas l'opérateur* X*) décrit une propriété insensible au bégaiement. Inversement, toute propriété insensible au bégaiement peut être exprimée comme une formule LTL$\setminus$X* [21].

2.2 Approche TGBA

Nous commençons par définir les *automates de Büchi généralisé étiqueté sur les transitions* (TGBA) [17], utilisés dans notre contexte pour représenter la négation de la propriété LTL à vérifier. TGBA est une généralisation des automates de Büchi classiques (BA).

Définition 2 *Un TGBA sur l'alphabet $\Sigma = 2^{AP}$ est un quadruplet $\mathcal{G} = \langle Q, I, \delta, \mathcal{F} \rangle$ avec:*

- *Q est un ensemble fini d'états,*
- *$I \subseteq Q$ est l'ensemble des états initiaux,*
- *F est un ensemble fini d'éléments appelés conditions d'acceptation,*
- *$\delta \subseteq Q \times 2^{\Sigma} \times 2^{\mathcal{F}} \times Q$ est la relation de transition de l'automate (chaque transition $(s, K, F, d) \in \delta$ partant de l' état s vers d est étiquetée par un ensemble de valuations K ainsi qu'un ensemble de conditions d'acceptation $F \subseteq \mathcal{F}$).*

Une exécution $w = k_0 k_1 k_2 \ldots \in \Sigma^{\omega}$ est acceptée par $\mathcal{G}$ s'il existe un chemin infini :
$(s_0, K_0, F_0, s_1)(s_1, K_1, F_1, s_2)(s_2, K_2, F_2, s_3) \ldots \in \delta^{\omega}$ tel que :

- *$s_0 \in I$, and $\forall i \in \mathbb{N}$, $k_i \in K_i$ (l'exécution est reconnue par le chemin),*
- *$\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$ (le chemin visite infiniment souvent chaque condition d'acceptation de $\mathcal{F}$)*

Le langage $\mathcal{L}(\mathcal{G}) \subseteq \Sigma^{\omega}$ est l'ensemble des exécutions acceptées par $\mathcal{G}$.

Toute formule LTL φ peut être converti en un TGBA dont le langage est l'ensemble des exécutions qui valident φ. Il existe plusieurs algorithmes de traduction de formule LTL en TGBA [9, 14, 17, 1].

La figure 2 montre deux exemples de TGBA: un TGBA déterministe dérivé de la formule LTL $\mathsf{GF}a \wedge \mathsf{GF}b$, et un TGBA non-déterministe dérivé de $a\mathsf{U}\mathsf{G}b$. L'étiquette à l'intérieur de chaque état est une formule LTL représentant la propriété acceptée à partir de cet état de l'automate: ces étiquettes servent uniquement à faciliter la compréhension de l'automate mais ne sont pas utilisés par l'algorithme de model checking. Comme on peut le voir sur la figure 2.(a), une formule LTL de la forme $\bigwedge_{i=1}^{n} \mathsf{GF}\, p_i$ peut être représentée par un TGBA déterministe composé d'un seul état et de n conditions d'acceptation. Dans ces deux TGBA, tout chemin infini est acceptée s'il visite infiniment souvent toutes les conditions d'acceptation (représentées sur les transitions par des disques de couleurs différentes, ici noir et blanc).

Dans l'approche TGBA, l'emptiness check consiste à rechercher les SCCs (*Strongly Connected Components*) contenant au moins une occurrence de chaque condition d'acceptation. Cette recherche peut s'effectuer de différentes façons [10]. Dans Spot, l'implémentation de l'emptiness check est basée sur l'algorithme de Couvreur [9]. Ce dernier utilise la méthode de Dijkstra pour la détection des SCCs dans un graphe. Les détails des algorithmes de chaque approche seront publiés dans un numéro de ToPNoC [2].

Figure 3: Deux exemples de BA, avec les états acceptants représentés par des doubles cercles. (a) BA de la propriété $\varphi = \mathsf{GF}a \wedge \mathsf{GF}b$ obtenu par dégénéralisation du TGBA de la figure 2(a). (b) BA de la propriété $a \mathsf{U} \mathsf{G} b$.

2.3 Approche BA

Un automate de Büchi (BA) peut être vu comme un TGBA avec une seule condition d'acceptation portée par les états au lieu des transitions. Le processus de conversion d'un TGBA en BA est appelé *dégénéralisation*. Une technique courante permettant d'obtenir un BA à partir d'une formule LTL est de traduire d'abord la formule en automate de Büchi généralisée avec plusieurs conditions d'acceptation (ce pourrait être un TGBA [17, 14] ou un GBA étiquetée sur les états [16]), ensuite *dégénéraliser* cet automate pour obtenir un BA avec une seule condition d'acceptation.

Définition 3 *Un BA sur l'alphabet $\Sigma = 2^{AP}$ est un quadruplet $\mathcal{B} = \langle Q, I, \delta, \mathcal{F} \rangle$ avec:*

- *Q est un ensemble fini d'états,*
- *$I \subseteq Q$ est l'ensemble des états initiaux,*
- *$\mathcal{F}$ est un ensemble fini d'états acceptants,*
- *$\delta \subseteq Q \times 2^{\Sigma} \times Q$ est la relation de transition de l'automate (chaque transition $(s, K, d) \in \delta$ partant de l' état 's' vers 'd' est étiquetée par un ensemble de valuations $K \subseteq \Sigma$).*

Une exécution $w = k_0 k_1 k_2 \ldots \in \Sigma^{\omega}$ est acceptée par $\mathcal{B}$ s'il existe un chemin infini :
$(s_0, K_0, s_1)(s_1, K_1, s_2)(s_2, K_2, s_3) \ldots \in \delta^{\omega}$ tel que :

- *$s_0 \in I$, and $\forall i \in \mathbb{N}, k_i \in K_i$ (l'exécution est reconnue par le chemin),*
- *$\forall i \in \mathbb{N}, \exists j \geq i, s_j \in \mathcal{F}$ (le chemin visite infiniment souvent au moins un état de $\mathcal{F}$)*

Le langage $\mathcal{L}(\mathcal{B}) \subseteq \Sigma^{\omega}$ est l'ensemble des exécutions acceptées par $\mathcal{B}$.

Dans le pire des cas, la dégénéralisation d'un TGBA contenant s états et n conditions d'acceptation permettra d'obtenir un BA avec $s \times (n+1)$ états.

La figure 3 représente les mêmes propriétés que la figure 2, mais exprimées en automates de Büchi. L'automate de la figure 3(a) a été construit par dégénéralisation du TGBA de la figure 2(a), le pire cas de la dégénéralisation a eu lieu ici: en effet, un TGBA contenant 1 état et 2 conditions d'acceptation a été dégénéralisé en un BA contenant $2+1$ états. La propriété $a \mathsf{U} \mathsf{G} b$ est plus facile à exprimer: le BA (figure 2(b)) a la même taille que le TGBA.

Un BA peut être considéré comme un TGBA, en ajoutant une unique condition d'acceptation sur toutes les transitions sortant des états acceptants. Ainsi, les algorithmes qui prennent en entrée un TGBA peuvent être adaptés pour traiter un BA. Notamment, l'emptiness check TGBA est aussi utilisé dans l'approche BA.

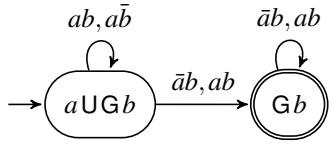
2.4 Approche TA

Les automates Testeur TA (Testing Automata) ont été introduits par Hansen et al. [19] pour représenter les propriétés insensibles au bégaiement. À la différence des automates de Büchi qui eux observent les valeurs des propositions atomiques (AP), l'idée de base des TA est de détecter les *changements* de ces valeurs. En effet, si les valeurs des propositions atomiques ne changent pas entre deux valuations consécutives d'une exécution (i.e. l'exécution bégaye), alors le TA reste dans le même état. Par conséquent, pour pouvoir accepter les exécutions qui restent à l'infini sur le même état, un nouveau type d'états acceptants a été introduit nommé *état livelock-acceptant*.

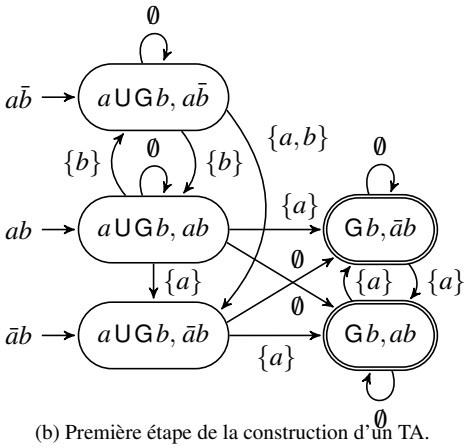
Soit A et B deux valuations de Σ, on désigne par $A \oplus B$ la différence symétrique $(A \setminus B) \cup (B \setminus A)$, c'est-à-dire l'ensemble des propositions atomiques qui changent (de valeur de vérité) entre A et B (par exemple, $a\bar{b} \oplus ab = \{b\}$).

Définition 4 *Un TA sur l'alphabet $\Sigma = 2^{AP}$ est un sextuplet $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, G \rangle$ tels que:*

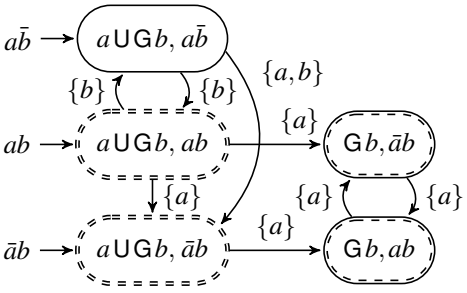
- *Q est un ensemble fini d'états,*



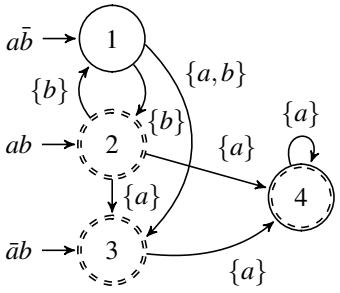
(a) BA de $a \mathsf{U} \mathsf{G} b$ à transformer en TA



(b) Première étape de la construction d'un TA.



(c) Deuxième étape de la construction d'un TA



(d) TA minimisé par bisimulation

Figure 4: Les étapes de la construction d'un TA à partir d'un BA. Les états de $\mathcal{F} \setminus G$ sont dessinés avec une bordure en double traits pleins, les états de $G \setminus \mathcal{F}$ bordure en double traits pointillés, et les états de $\mathcal{F} \cap G$ bordure mixte 'trait plein/trait pointillé'.

- $I \subseteq Q$ *est un ensemble d'états initiaux,*
- $U : I \to 2^{\Sigma}$ *est une fonction associant à chaque état initial, un sous-ensemble de valuations de* Σ*,*
- $\delta \subseteq Q \times \Sigma \times Q$ *est la relation de transition de l'automate, tel que chaque transition* $(s,k,d) \in \delta$ *partant de l'état* s *vers* d *est étiquetée par un* changeset*:* $k \in \Sigma$ *interprété comme l'ensemble des propositions atomiques qui changent entre* s *et* d*.*
- $\mathcal{F} \subseteq Q$ *est un ensemble d'états Büchi-acceptants,*
- $G \subseteq Q$ *est un ensemble d'états livelock-acceptants.*

On appelle transition bégayante*, toute transition étiquetée par un* changeset *égal à l'ensemble vide (*$(s,\emptyset,d) \in \delta$*). Par défaut dans un TA, on a implicitement une boucle bégayante sur chaque état.*
Une exécution $w = k_0 k_1 k_2 \ldots \in \Sigma^{\omega}$ *est acceptée par* $\mathcal{T}$ *s'il existe un chemin infini:*
$(s_0, k_0 \oplus k_1, s_1)(s_1, k_1 \oplus k_2, s_2) \ldots (s_i, k_i \oplus k_{i+1}, s_{i+1}) \ldots \in (Q \times \Sigma \times Q)^{\omega}$ *tels que:*

- $s_0 \in I$ *avec* $k_0 \in U(s_0)$*,*
- $\forall i \in \mathbb{N}$*, soit* $(s_i, k_i \oplus k_{i+1}, s_{i+1}) \in \delta$ *(le TA exécute une transition explicite), soit* $k_i = k_{i+1} \wedge s_i = s_{i+1}$ *(l'exécution bégaye et le TA reste dans le même état),*
- $\forall i \in \mathbb{N}$, $(\exists j \geq i, k_j \neq k_{j+1}) \wedge (\exists l \geq i, s_l \in \mathcal{F})$ *(le chemin visite infiniment souvent des états Büchi-acceptants et des transitions non bégayantes), ou,*
  $\exists n \in \mathbb{N}$, $(s_n \in G \wedge (\forall i \geq n, s_i = s_n \wedge k_i = k_n))$ *(le chemin atteint un état livelock-acceptant et reste bloqué dans cet état car l'exécution bégaye à l'infini).*

Le langage $\mathcal{L}(\mathcal{T}) \subseteq \Sigma^{\omega}$ *est l'ensemble des exécutions acceptées par* $\mathcal{T}$*.*

Pour illustrer cette définition, considérons la figure 4d, représentant le TA de la formule $a \mathsf{U} \mathsf{G} b$.

- L'exécution $ab;\bar{a}b;ab;\bar{a}b;ab;\bar{a}b;ab;\ldots$ est Büchi-acceptante. En effet, un chemin reconnaissant cette exécution, doit commencer à partir de l'état initial 2, puis changer la valeur de a à chaque transition (i.e. ne franchir que les transitions étiqueté par $\{a\}$) jusqu'à atteindre et boucler à l'infini dans le cycle $(4,\{a\},4)$ autour de l'état $4 \in \mathcal{F}$. Par exemple, le chemin $2 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4\cdots$ ou le chemin $2 \xrightarrow{\{a\}} 3 \xrightarrow{\{a\}} 4 \xrightarrow{\{a\}} 4\cdots$, les deux visitent l'état $4 \in \mathcal{F}$ infiniment souvent, ils sont donc Büchi-acceptants.
- L'exécution $ab;\bar{a}b;\bar{a}b;\bar{a}b;\ldots$ est livelock-acceptante: un chemin acceptant commence dans l'état 2, puis change la valeur de a pour se déplacer à l'état 4, et y reste pour bégayer à l'infini dans cet état livelock-acceptant. Un autre chemin acceptant passe de l'état 2 à l'état 3 et reste à l'infini dans $3 \in G$.
- L'exécution $ab;a\bar{b};ab;a\bar{b};ab;a\bar{b};\ldots$ n'est pas acceptée. Elle peut correspondre à un chemin contenant le cycle alternant entre les états 2 et 1, mais ce cycle n'est ni Büchi acceptant (ne visite pas d'états de $\mathcal{F}$), ni livelock-acceptant (il passe par l'état $2 \in G$, mais ne reste pas définitivement dans cet état).

Propriété 1 *Le langage accepté par un TA est insensible au bégaiement.*

Démonstration 1 *D'après la définition d'une exécution acceptée, un TA peut rester dans le même état quand l'exécution bégaye. Ainsi, le bégaiement est toujours possible dans un TA.*

2.4.1 Première étape de la construction d'un TA

Geldenhuys et Hansen [15] ont proposé une transformation permettant de convertir un BA en un TA équivalent, en plusieurs étapes. D'abord, le BA est converti en un automate étiqueté par des valuations sur les états (*State-Labelled Büchi Automaton*). Ensuite, ce dernier est converti dans une forme intermédiaire d'un TA, en calculant la différence symétrique entre les étiquettes de la source et de la destination de chaque transition. Ces deux étapes sont regroupées en une seule première étape représentée par la figure 4b. L'automate obtenu n'utilise pas encore les états livelock-acceptants de G. L'étape suivante explique comment l'ajout de certains états à l'ensemble G permet de supprimer les transitions étiquetées par $\emptyset$ (i.e. les transitions bégayantes).

2.4.2 Deuxième étape de la construction d'un TA

Afin d'obtenir la forme finale du TA, il faut d'abord calculer toutes les SCCs bégayantes, c'est-à-dire les SCCs composées de transitions étiquetées par $\emptyset$. Si une SCC bégayante est aussi acceptante (contenant un état de $\mathcal{F}$), alors tous ses états sont ajoutés à G. En plus, chaque état pouvant atteindre par une séquence de transitions bégayantes un état de G (resp. I) est lui aussi ajouté à G (resp. I). Enfin, on supprime toutes les transitions bégayantes et ensuite tous les état inutiles (à partir desquels on ne peut pas atteindre, ni un cycle Büchi-acceptant, ni un cycle livelock-acceptant). La figure 4c montre le résultat de cette deuxième étape.

Enfin, le TA obtenu, peut être minimisé en fusionnant les états bisimilaires (deux états sont bisimilaires si les mêmes chemins sont accepté par l'automate, à partir de l'un ou l'autre de ces deux états) [e.g., 25].

Figure. 4 montre les deux étape pour transformer le BA de la formule $a\,\mathsf{U}\,\mathsf{G}\,b$ en TA, et enfin la fusion des états bisimilaires (figure 4d).

Malheureusement, dans l'approche TA, les deux manières différentes pour accepter une exécution (livelock ou Büchi), impliquent un algorithme d'emptiness check en deux passes:

- Une première passe pour détecter les cycles Büchi-acceptants, contenant au moins un état de $\mathcal{F}$ et au moins une transition non bégayante.
- Une deuxième passe pour les cycles livelock-acceptants, composés uniquement d'états de G et de transitions bégayantes.

3 Contributions

Après avoir implémenté l'approche basée sur les TA, décrite dans la section précédente, nous l'avons améliorée de plusieurs façons:

- Amélioration de l'algorithme de vérification en deux-passes de l'approche TA.
- Proposition d'une méthode permettant de transformer un TA en un automate vérifiable en une seule passe, qu'on a appelé STA (*Single-pass Testing Automata*).

La contribution la plus significative est un nouveau type d'automate baptisé TGTA (*Transition-based Generalized Testing Automata*). L'apport principale de cette nouvelle approche consiste à proposer une méthode permettant de supprimer les transitions bégayantes dans un TGTA sans avoir besoin, ni d'ajouter une seconde passe comme c'est le cas dans l'approche TA, ni d'ajouter un état artificiel comme c'est le cas dans l'approche STA.

Par la suite, en s'appuyant sur l'observation que les TGTA sont plus efficaces que les autres automates évalués dans le contexte de l'approche explicite, on a utilisé les TGTA pour proposer d'autres améliorations du model checking dans deux autres contextes: les approches symbolique et hybride. En particulier, nous avons significativement amélioré les permformances du model checking symbolique en combinant les TGTA avec la technique de saturation [7].

Toutes ces contributions ont été implémentées en C++ et intégrées à SPOT [13], la bibliothèque de model checking développée au LRDE. Ensuite, à l'aide de cette bibliothèque, des expérimentations ont été menés afin d'évaluer l'apport de la nouvelle approche TGTA par rapport aux autres approches dans les trois contextes: explicite, symbolique et hybride. Les résultats de ces évaluations sont présentés à la fin de chaque partie.

Ces travaux ont donné lieu à trois publications: deux articles de conférences [22, 4] et un article de revue [3].

3.1 Amélioration de l'emptiness check dans l'approche TA

Lors du développement de l'approche TA dans Spot, nous avons implémenté l'algorithme d'emptiness check proposé par Geldenhuys et Hansen [15] et nous avons ajouté les optimisations suivantes à la première passe:

1. Si aucun état livelock-acceptant n'est visité au cours de la première passe, alors la seconde passe est désactivée. Les résultats de l'expérimentation montrent que cette optimisation améliore considérablement les performances de l'approche TA, notamment dans les cas où la formule est vérifiée.

2. Lors de la première passe, si l'on détecte un cycle contenant un état qui est à la fois Büchi et livelock acceptant, alors ce cycle est considéré comme acceptant sans vérifier s'il contient ou pas des transitions bégayantes. En effet, si le cycle détecté contient une transition non bégayante, alors il est Büchi-acceptant, sinon il est livelock-acceptant.

3. Un cycle détecté lors de la première passe est considéré comme acceptant dès qu'il contient un état livelock-acceptant (k,t) tel que l'état t n'a pas de successeur dans l'automate TA (ici k est un état du modèle). En effet, à partir de cet état, le produit ne peut exécuter que des transitions bégayantes. Par conséquent, le cycle détecté est livelock-acceptant car il est composé uniquement de transitions bégayantes et d'états livelock-acceptants.

Les deux dernières optimisations permettent de détecter une partie des cycles livelock-acceptants, mais peuvent aussi manquer certains cycles livelock-acceptants qui appartiennent à une SCC mélangeant des transitions bégayantes et non-bégayantes. Dans la section suivante, nous introduisons une transformation de TA permettant d'obtenir un automate où les deux dernières optimisations permettront de détecter tous les cycles livelock-acceptants dès la première passe.

3.2 Transformation d'un TA en un automate vérifiable en une seule passe (STA)

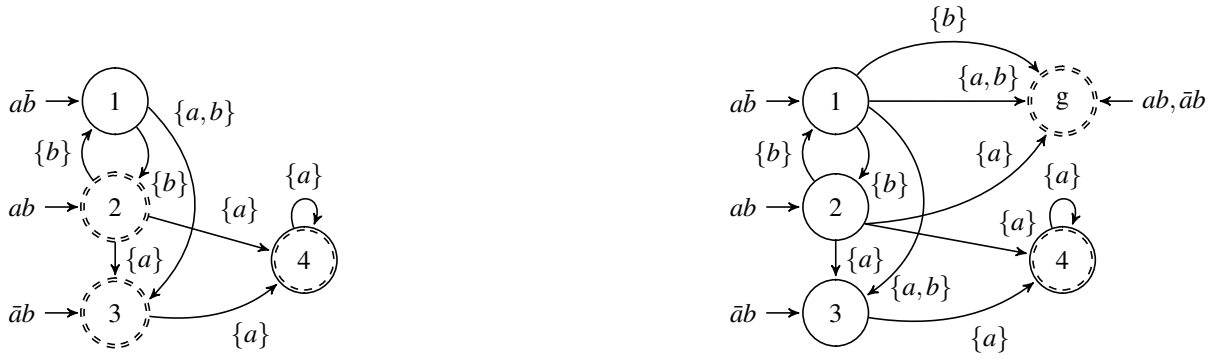
Dans la suite, on appelle état *livelock-acceptant pur*, tout état livelock-acceptant qui n'est pas Büchi-acceptant.

Dans cette section, nous proposons une transformation d'un automate de type TA en une forme normale appelée STA (*Single-pass Testing Automata*). Cette transformation permet d'améliorer les performances en supprimant la seconde passe de l'emptiness check.

D'abord, nous définissons un STA comme une restriction d'un TA respectant une contrainte supplémentaire sur les états livelock-acceptants. Ensuite, nous proposons une méthode pour transformer automatiquement n'importe quel TA en STA. Cette transformation nécessite l'ajout d'un état artificiel.

3.2.1 Définition d'un STA (*Single-pass Testing Automata*)

Un STA est un automate de type TA, dans lequel chaque état livelock-acceptant $(\in G)$ respecte au moins une de ces deux conditions: soit il est à la fois Büchi-acceptant et livelock-acceptant $(\in G \cup \mathcal{F})$, soit il n'a pas de successeurs explicites (la figure 5b montre un exemple de STA). En d'autres termes, dans un STA, tout état livelock-acceptant pur ne possède pas de successeurs explicites. Ainsi, à partir d'un état livelock-acceptant pur, un automate de type STA ne peut exécuter que des transitions bégayantes.



(a) TA de $a \mathsf{U} \mathsf{G} b$ à transformer en STA

(b) STA de $a \mathsf{U} \mathsf{G} b$

Figure 5: Transformation du TA de $a \mathsf{U} \mathsf{G} b$ en STA.

Formellement, un automate de type STA est un sextuplet $\langle Q, I, U, \delta, \mathcal{F}, G \rangle$ respectant la définition d'un TA et satisfaisant en plus la contrainte suivante : $\delta \cap ((G \setminus \mathcal{F}) \times 2^K \times Q) = \emptyset$.

3.2.2 Construction d'un STA à partir d'un TA (figure 5)

Afin de transformer un TA $\mathcal{T}$ en un STA $\mathcal{T}'$ équivalent, il faut d'abord ne pas considérer les états livelock-acceptants purs de $\mathcal{T}$ comme livelock-acceptant dans $\mathcal{T}'$ (cela change le langage). Ensuite, pour obtenir un $\mathcal{T}'$ acceptant le même langage que $\mathcal{T}$, on commence par ajouter un nouveau état livelock-acceptant sans successeurs à $\mathcal{T}'$ qu'on note g, puis pour chaque transition dont la destination est un état livelock-acceptant pur dans $\mathcal{T}$, on ajoute dans $\mathcal{T}'$ une transition équivalente vers l'état artificiel g. Cette transformation est illustrée par la figure 5 et elle est formellement définie par la propriété suivante:

Propriété 2 *A partir d'un automate de type TA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F}, G \rangle$, l'automate $\mathcal{T}' = \langle Q', I', U', \delta', \mathcal{F}, G' \rangle$ défini ci-après est un STA tel que $\mathscr{L}(\mathcal{T}') = \mathscr{L}(\mathcal{T})$, avec:*

- $G' = (G \cap \mathcal{F}) \cup \{g\}$ *avec $g \notin Q$ est un nouveau état (qui n'a pas de successeurs),*
- $Q' = (Q \setminus G_\emptyset) \cup \{g\}$ *avec $G_\emptyset = \{s \in G \mid (\{s\} \times K \times Q) \cap \delta = \emptyset\}$ est l'ensemble des états de G qui n'ont pas de successeurs dans δ.*
- $I' = I \cup \{g\}$ *si $G \cap I \neq \emptyset$, sinon $I' = I$,*
- $\delta' = (\delta \setminus (Q \times K \times G_\emptyset)) \cup \{(s,k,G) \mid (s,k,d) \in \delta, d \in (G \setminus \mathcal{F}) \cup G_\emptyset\}$,
- $\forall s \in I, U'(s) = U(s)$ *et si* $g \in I', U'(g) = \bigcup_{s \in (G \cap I)} U(s)$.

L'ensemble G' obtenu contient un unique état livelock-acceptant pur g (tous les états livelock-acceptants purs dans $\mathcal{T}$ ne le sont plus dans $\mathcal{T}'$). Ainsi, comme g n'a pas de successeurs, l'automate $\mathcal{T}'$ respecte bien la définition d'un STA.

L'automate $\mathcal{T}'$ est équivalent à $\mathcal{T}$. En effet, par définition de l'état g, chaque exécution livelock-acceptante dans $\mathcal{T}$ est associée à une exécution livelock-acceptante dans $\mathcal{T}'$ captée par l'état g, et vice versa.

3.2.3 Apport d'une approche de Model Checking basée sur les STA

Dans un automate STA, tous les états livelock-acceptants purs sont des états sans successeurs, c'est-à-dire, à partir de tels états l'automate ne peut exécuter que des transitions bégayantes. Par conséquent, dans le produit, toute SCC contenant un état livelock-acceptant pur est forcement composée uniquement de transitions bégayantes. Ainsi, au cours de l'emptiness check, si la première passe détecte une SCC non-triviale contenant un état livelock-acceptant pur, alors elle répond que le produit contient un cycle livelock-acceptant sans effectuer une deuxième passe. De la même manière, la première passe est aussi capable de détecter le reste de cycles livelock-acceptants ne contenant pas d'états livelock-accpetants purs. En effet, si une SCC contient un état à la fois livelock et Büchi acceptant alors elle contient un cycle acceptant indépendamment des transitions qui le composent.

Malgré que l'ajout d'un état artificiel augmente légèrement la taille de produit, un algorithme en une seul passe (STA) est plus performant qu'un algorithme explorant un produit plus petit deux fois (TA).

(a) Avant la réductions des transitions bégayantes ($\emptyset$). (b) Réduction des transitions bégayantes.

Figure 6: L'intuition de la réduction des transitions bégayantes dans un TGTA.

3.2.4 Construction d'un STA contenant moins de transitions (plus déterministe)

Afin d'obtenir un STA contenant moins de transitions, il est possible de réduire le nombre des états livelock-acceptants purs. En effet, lors de la construction d'un TA, les états appartenant à une SCC bégayante et acceptante, sont ajouté à G. Cependant, il est aussi possible d'ajouter ces états à la fois à G et à $\mathcal{F}$ car un cycle contenant l'un de ces états est acceptant indépendamment des transitions qui le composent.
L'idée de cette optimisation sera formalisé de façon plus claire lors de la section suivante décrivant la construction d'un nouveau type d'automates (TGTA) permettant d'avoir un algorithme d'emptiness check en une seule passe sans l'ajout d'un état artificiel.

3.3 TGTA un nouveau type d'automate pour la vérification des propriétés insensibles au bégaiement

Les résultats de nos expérimentations ont montré que malgré les différentes améliorations apportées à l'approche TA, l'approche TGBA reste meilleure dans certain cas.

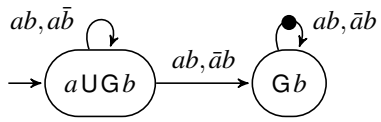
Cette section présente un nouveau type d'automate qui combine à la fois les avantages des TA et des TGBA. L'automate obtenu est un automate Testeur généralisé étiqueté sur les Transitions, que nous appelons TGTA (*Transition-based Generalized Testing Automaton*) :

- Des TA, nous avons repris l'idée de l'étiquetage des transitions par des 'changesets', mais nous avons supprimé le bégaiement implicite sur le même état. En effet, dans un TGTA, le bégaiement est explicitement représenté par des boucles bégayantes.
- Des TGBA, nous avons repris l'utilisation des conditions d'acceptation généralisées étiquetées sur les transitions.

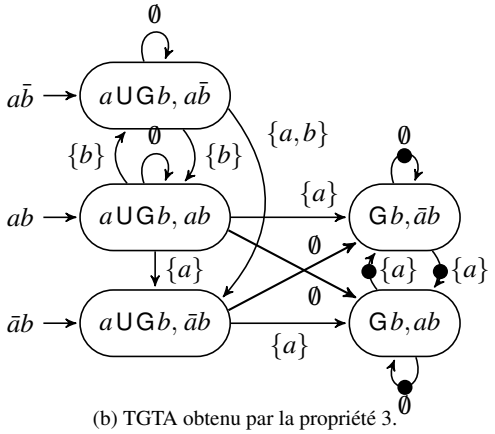
Grâce à l'utilisation de ces conditions d'acceptation généralisées, et en s'inspirant de la technique de réduction utilisée pour les TA, nous proposons une méthode permettant de supprimer les transitions bégayantes (entre deux états distincts) dans un TGTA, sans avoir besoin d'ajouter une seconde passe lors de l'emptiness check (voir figure 6). En effet, la réduction qu'on propose permet d'éviter de définir la notion d'état livelock-acceptant dans un TGTA, car nous avons déjà vu que l'utilisation de ce type d'états implique un algorithme d'emptiness check en deux passes dans l'approche TA ou l'ajout d'un état artificiel dans l'approche STA. Cette réduction étant basée sur l'élimination de transitions bégayantes, l'automate TGTA obtenu reconnaît uniquement les langages insensibles au bégaiement.
Un autre avantage de l'utilisation dans les TGTA, des conditions d'acceptation généralisées sur les transitions peut être déduit de la comparaison entre les TGBA et les BA. En effet, dans la section 2 on a déjà vu que les TGBA sont généralement plus petits que les BA, et comme les TA sont construits à partir des BA alors que les TGTA sont eux obtenus à partir des TGBA, on peut donc s'attendre à ce que les TGTA soit généralement plus petits que les TA.

Définition 5 *Un* automate Testeur généralisé sur les transitions *(TGTA) est un automate Testeur (sans états livelock-acceptants) dans lequel les conditions d'acceptation de Büchi généralisées portent sur les transitions. C'est-à-dire un quintuplet $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$ où :*

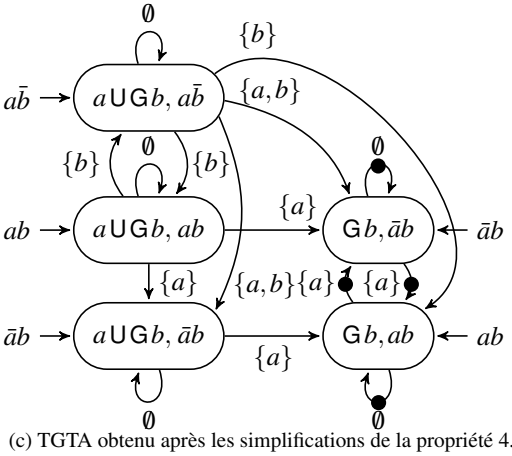
- *Q est un ensemble fini d'états,*
- *$I \subseteq Q$ est un ensemble d'états initiaux,*
- *$U : I \to 2^{\Sigma}$ est une fonction associant à chaque état initial, un sous-ensemble de valuations de Σ,*
- *$\mathcal{F}$ est un ensemble fini d'éléments appelés conditions d'acceptation,*
- *$\delta \subseteq Q \times \Sigma \times 2^{\mathcal{F}} \times Q$ est la relation de transition de l'automate (chaque transition (s,k,F,d) partant de l' état s vers d est étiquetée par un «changeset» k ainsi qu'un ensemble de conditions d'acceptation $F \subseteq \mathcal{F}$).*



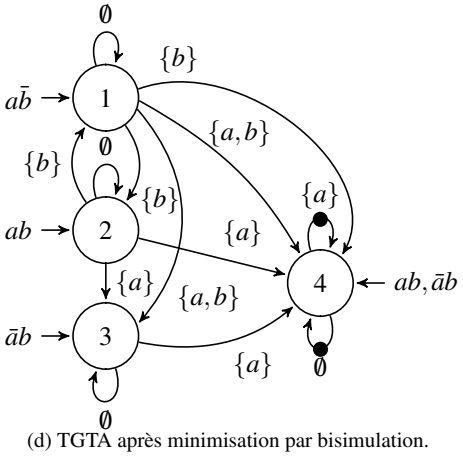
(a) TGBA de $a\mathsf{U}\mathsf{G}b$ à transformer en TGTA



(b) TGTA obtenu par la propriété 3.



(c) TGTA obtenu après les simplifications de la propriété 4.



(d) TGTA après minimisation par bisimulation.

Figure 7: Les différentes étapes de la construction d'un TGTA à partir du TGBA représentant $a\mathsf{U}\mathsf{G}b$, avec $\mathcal{F} = \{\bullet\}$.

- *En plus, δ doit satisfaire une contrainte de normalisation des transitions bégayantes suivantes:*
  1. *Toutes les transitions bégayantes sont des boucles (self-loops),*
  2. *et tous les états ont une boucle bégayante.*

  Formellement, cette contrainte de normalisation est exprimée par l'équivalence suivante: $\forall (q,q') \in Q^2 : \left(\exists F \in 2^{\mathcal{F}}, (q,\emptyset,F,q') \in \delta\right) \iff (q = q')$

Une exécution $w = k_0k_1k_2\ldots \in \Sigma^\omega$ est acceptée par $\mathcal{T}$ s'il existe un chemin infini : $(s_0, k_0 \oplus k_1, F_0, s_1)(s_1, k_1 \oplus k_2, F_1, s_2)(s_2, k_2 \oplus k_3, F_2, s_3)\ldots \in \delta^\omega$ *tel que :*

- $s_0 \in I$ *with* $k_0 \in U(s_0)$ *(l'exécution est reconnue par le chemin),*
- $\forall f \in \mathcal{F}, \forall i \in \mathbb{N}, \exists j \geq i, f \in F_j$ *(le chemin visite infiniment souvent chaque condition d'acceptation de $\mathcal{F}$)*

Le lanagage $\mathscr{L}(\mathcal{T}) \subseteq \Sigma^\omega$ est l'ensemble des exécutions acceptées par $\mathcal{T}$.

La figure 7 montre les étapes de construction d'un TGTA reconnaissant la formule $a\mathsf{U}\mathsf{G}b$ (comme nous l'avons déjà fait pour un TA dans la figure 4d). Les exécutions acceptées sont ceux qui visitent l'unique condition d'acceptation $\bullet$ infiniment souvent. Il est possible de vérifier que toutes les exécutions prises comme exemple dans la section 2.4 (TA) sont aussi acceptées par le TGTA construit, mais pas toujours avec les mêmes exécutions (par exemple $ab;\bar{a}b;\bar{a}b;\bar{a}b;\ldots$ est acceptée par l'exécution $2,4,4,4,\ldots$ dans le TGTA alors qu'il accepté par l'exécution $2,3,3,3,\ldots$ dans le TA). Cette différence est due à la façon dont nous émulons les états livelock-acceptants, comme nous allons le voir dans la Propriété 4.

3.4 Construction d'un TGTA à partir d'un TGBA

Dans cette section, nous présentons deux propriétés définissant les étapes de la transformation d'un TGBA en TGTA. Cette transformation est inspirée de la construction d'un TA à partir d'un BA présentée dans la section 2.4. La propriété qui suit est inspirée de la section 2.4.1, elle présente la première étape de construction d'un TGTA à partir d'un TGBA, en poussant les étiquettes des transitions vers les états, et en étiquetant chaque transition par la différence (changeset) entre les étiquettes des deux états source et destination de la transition. Ces transformations des étiquettes sont effectuées en gardons les conditions d'acceptation généralisées sur les transitions. La figure 7b présente un exemple de TGTA obtenu à la suite de cette première étape de la transformation.

Propriété 3 (Transformation d'un TGBA en TGTA) *À partir d'un TGBA* $\mathcal{G} = \langle S_\mathcal{G}, I_\mathcal{G}, R_\mathcal{G}, \mathcal{F} \rangle$ *sur l'alphabet* $\Sigma = 2^{AP}$ *tel que* $\mathscr{L}(\mathcal{G})$ *est insensible au bégaiement, on construit un TGTA* $\mathcal{T} = \langle S_\mathcal{T}, I_\mathcal{T}, U_\mathcal{T}, R_\mathcal{T}, \mathcal{F} \rangle$ *tels que:* $S_\mathcal{T} = S_\mathcal{G} \times \Sigma$, $I_\mathcal{T} = I_\mathcal{G} \times \Sigma$ *and*

(i) $\forall (s,k) \in I_\mathcal{T}, U_\mathcal{T}((s,k)) = \{k\}$

(ii) $\forall (s,k) \in S_\mathcal{T}, \forall (s',k') \in S_\mathcal{T}$,
$((s,k), k \oplus k', F, (s',k')) \in R_\mathcal{T} \iff \exists K \in 2^\Sigma, ((s,K,F,s') \in R_\mathcal{G}) \wedge (k \in K)$

Alors $\mathscr{L}(\mathcal{G}) = \mathscr{L}(\mathcal{T})$.

La deuxième propriété est inspirée de la section 2.4.2, elle permet de supprimer les transitions bégayantes à l'exception des boucles. L'intuition derrière cette simplification est illustrée par la figure 6a: un état s_0 est accessible à partir d'un état s par une transition non-bégayante ($k \neq \emptyset$). En plus, s_0 peut atteindre, à travers une suite de transitions bégayantes, un cycle bégayant et acceptant (autour de s_n). Dans le cas des TA, il fallait déclarer s_0 et s_n comme étant des états livelock-acceptants. Pour les TGTA, nous remplaçons le cycle bégayant et acceptant par l'ajout d'une boucle étiquetée par toutes les conditions d'acceptation sur s_n, puis les prédécesseurs de s_0 sont connectés à s_n comme c'est le cas pour l'état s sur la figure 6b.

Propriété 4 (Réduction des transitions bégayantes dans un TGTA) *Soit* $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$ *un TGTA tel que* $\mathscr{L}(\mathcal{T})$ *est insensible au bégaiement. En combinant les trois premières opérations suivantes, nous pouvons supprimer toutes les transitions bégayantes qui ne sont pas des boucles (voir figure. 6). À la suite de ces modifications, une quatrième opération est proposée pour nettoyer le TGTA obtenu en supprimant les états qui sont devenus inutile.*

1. *Soit* $Q \subseteq Q$ *une SCC tel que pour tout couple d'états* $q, q' \in Q$, *il existe une séquence de transitions bégayantes* $(q, \emptyset, F_0, q_1)(q_1, \emptyset, F_1, q_2) \cdots (q_n, \emptyset, F_n, q') \in \delta^*$ *avec* $F_0 \cup F_1 \cup \cdots \cup F_n = \mathcal{F}$. *Alors on peut ajouter une boucle bégayante et acceptante* $(q, \emptyset, \mathcal{F}, q)$ *sur chaque état* $q \in Q$. *Ainsi, le TGTA* $\mathcal{T}' = \langle Q, I, U, \delta \cup \{(q, \emptyset, \mathcal{F}, q) \mid q \in Q\}, \mathcal{F} \rangle$ *est tel que* $\mathscr{L}(\mathcal{T}') = \mathscr{L}(\mathcal{T})$. *Dans la suite, on appelle une telle composante Q une* ***SCC bégayante et acceptante****.*

2. *S'il existe une SCC bégayante et acceptante Q et une séquence de transitions bégayantes* $(s_0, \emptyset, F_1, s_1)(s_1, \emptyset, F_2, s_2) \cdots (s_{n-1}, \emptyset, F_n, s_n) \in \delta^*$ *tels que* $s_n \in Q$ *et* $s_0, s_1, \ldots s_{n-1} \notin Q$ *(Fig. 6a), Alors:*
   - *Pour toute transition non-bégayante* $(s, k, f, s_0) \in \delta$ *tel que* $k \neq \emptyset$, *le TGTA* $\mathcal{T}'' = \langle Q, I, U, \delta \cup \{(s, k, f, s_n)\}, \mathcal{F} \rangle$ *est tel que* $\mathscr{L}(\mathcal{T}'') = \mathscr{L}(\mathcal{T})$.
   - *Si* $s_0 \in I$, *le TGTA* $\mathcal{T}'' = \langle Q, I \cup \{s_n\}, U'', \delta, \mathcal{F} \rangle$ *avec* $\forall s \neq s_n, U''(s) = U(s)$ *et* $U''(s_n) = U(s_n) \cup U(s_0)$, *est tel que* $\mathscr{L}(\mathcal{T}'') = \mathscr{L}(\mathcal{T})$.

3. *Soit* $\mathcal{T}^\dagger = \langle Q, I^\dagger, U^\dagger, \delta^\dagger, \mathcal{F} \rangle$ *le TGTA obtenu après répétition des deux opérations précédentes autant que possible (c'est à dire jusqu'à obtenir un* $\mathcal{T}^\dagger$ *contenant toutes les transitions et tous les états initiaux qui peuvent être ajoutés par les deux opérations ci-dessus). Ensuite, on ajoute une boucle bégayante non-acceptante* $(s, \emptyset, \emptyset, s)$ *à chaque état qui ne dispose pas de boucle bégayante (acceptante ou pas), cette opération ne change pas* $\mathscr{L}(\mathcal{T}^\dagger)$ *car il est insensible au bégaiement. Ainsi, nous pouvons supprimer toutes les transitions bégayantes qui ne sont pas des boucles, car toutes les exécutions acceptantes et bégayantes peuvent être capturées par des boucles bégayantes à la suite de toutes les opérations précédentes. Plus formelement, si on pose* $\delta''' = \{(s, k, F, d) \in \delta^\dagger \mid k \neq \emptyset \vee (s = d \wedge F = \mathcal{F})\} \cup \{(s, \emptyset, \emptyset, s) \mid (s, \emptyset, \mathcal{F}, s) \notin \delta^\dagger\}$, *alors le TGTA* $\mathcal{T}''' = \langle Q, I^\dagger, U^\dagger, \delta''', \mathcal{F} \rangle$ *est tel que* $\mathscr{L}(\mathcal{T}''') = \mathscr{L}(\mathcal{T}^\dagger) = \mathscr{L}(\mathcal{T})$.

4. *Tout état à partir duquel on ne peut pas atteindre un cycle acceptant peut être supprimé du TGTA.*

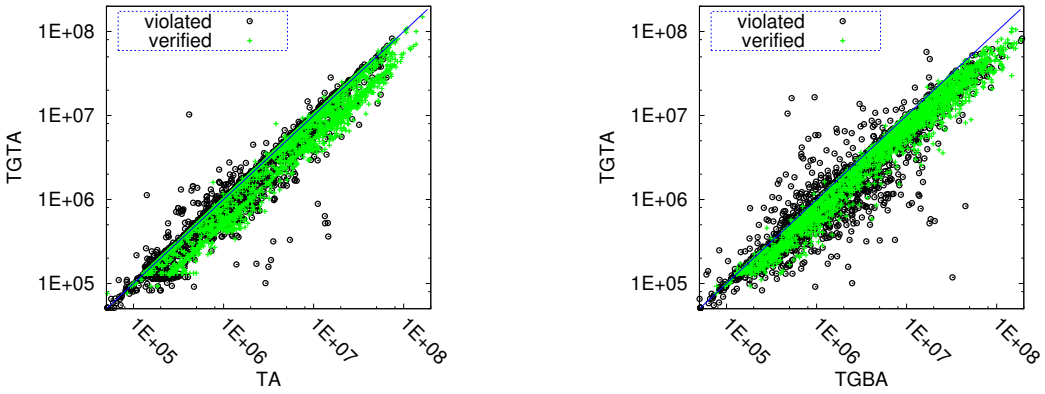


Figure 8: Evaluation des performances (en nombre de transitions explorées au cours de l'emptiness check) des TGTA par rapport aux TA et TGBA.

Une simplification supplémentaire des TGTA consiste à fusionner les états bi-similaires, cela peut être réalisé en utilisant le même algorithme utilisé pour minimiser un TA, en prenant en compte les conditions d'acceptation sur les transitions. Toutes les étapes de construction et de simplification d'un TGTA sont illustrées sur la figure 7.

On peut voir un TGTA comme un TGBA dont les transitions sont étiquetées par des changesets au lieu de valuations de propositions atomiques. Dans l'approche TGBA, l'algorithme d'emptiness check recherche un cycle contenant toutes les conditions d'acceptation, et accessible à partir d'un état initial. L'emptiness check de l'approche TGTA recherche exactement la même chose. Ainsi, Le même algorithme d'emptiness check peut être utilisé dans les deux approches, car cet algorithme ignore les étiquettes des transitions. Il s'agit d'une propriété intéressante des TGTA, car non seulement elle implique un emptiness check en une seul passe, mais aussi parce qu'elle facilite l'implémentation de l'approche TGTA en réutilisant un model checker basé sur les TGBA.

3.5 Comparaison expérimentale des approches explicites TGBA, BA, TA et TGTA

Cette section présente les résultats de l'expérimentation menée afin de comparer les différents types d'approches, implémentées dans notre outil Spot [20].

Les deux *scatter-plots* de la figure 8 affichent les résultats de la vérification d'un total de 4100 formules sur différents modèles (Peterson, Ring, FMS, Kanban,...). Pour chaque formule, un point donne le nombre de transitions explorées lors de l'emptiness check dans l'approche TGTA, comparé à deux autres approches: TA (figure 8 à gauche) et TGBA (Figure 8 à droite); 2050 points sous la forme de croix vertes correspond à des formules vérifiées, et 2050 formules non vérifiées représentées par des cercles noirs. Chaque point en dessous de la diagonale est en faveur de l'approche TGTA, et inversement. Les axes utilisent une échelle *logarithmique*. La comparaison avec l'approche BA n'est présentée, car elle moins efficace que TGBA [22].

Pour les **propriétés vérifiées**, les résultats sont faciles à interpréter, l'approche TGTA est meilleure que TGBA et TA dans la majorité des cas. Notamment, dans la figure 8(gauche), le nuage linéaire de croix vertes au dessous de la diagonale, représente les cas où l'emptiness check de l'approche TA s'effectue en deux passes.

Les résultas sont plus difficiles à interpréter pour les **propriétés non vérifiées**. En effet, pour ces derniers, l'emptiness check s'arrête dès qu'il trouve un contre-exemple. Ainsi, la modification de l'ordre dans lequel les transitions non-déterministes sont explorées, est suffisante pour changer le nombre total des transitions explorées: dans le cas le plus favorable, l'ordre des transitions conduira directement à un cycle acceptant; dans le pire des cas, l'algorithme va explorer l'ensemble du produit jusqu'à ce qu'il trouve enfin un cycle acceptant. Cependant, comme les automates de la famille automates Testeurs (TGTA et TA) sont plus déterministes [15], elles sont généralement meilleures que les BA et TGBA.

3.6 Utilisation des TGTA pour améliorer l'approche Symbolique

Dans cette partie, nous avons combiné les TGTA avec la technique de saturation [7] dans le but d'améliorer les performances de l'approche symbolique du model checking, ceci nous a permis de vérifier des espaces d'états plus grands et d'obtenir des résultats nettement meilleures que dans le cas du model checking explicite. En particulier,

l'approche TGTA nous a permis d'obtenir des résultats meilleures que l'approche TGBA pour les deux types de formules: les formules satisfaites et les formules non satisfaites.

Cette partie commence par une présentation de l'approche symbolique traditionnelle, basée sur les automates de Büchi TGBA. Ensuite, nous présentons nos encodages symboliques des TGTA et d'une structure de Kripke adapté aux TGTA, appelée *Changeset-based symbolic Kripke structure*. En utilisant ces deux encodages, nous montrons ensuite comment encoder les transitions bégayantes dans le produit synchronisé pour améliorer les performances de la technique de saturation. Finalement, nous présentons les résultats de notre comparaison expérimentale entre les deux approches TGBA et TGTA, notamment l'effet de chaque approche sur les performances de la saturation.

3.6.1 Model Checking Symbolique

Nous commençons par présenter l'encodage symbolique de l'approche automate de model checking, basée sur un produit synchronisé entre une structure de Kripke et un TGBA symboliques.

Dans une approche symbolique, on réprésente la structure de Kripke à l'aide de prédicats [23] encodant des ensembles d'états et de transitions. L'implémentation de ces prédicats utilise des diagrammes de décision (BDDs) [6].

Définition 6 (Structure de Kripke Symbolique) *Une structure de Kripke $\mathcal{K}_M = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ est symboliquement représentée par les prédicats $P_{\mathcal{S}_0}$, $P_{\mathcal{R}}$ et P_l encodant respectivement l'ensemble des états initiaux, la relation de transition et les valuations des états. Ces prédicats portent sur les variables d'états $s, s' \in \mathcal{S}$ et $\ell \in \Sigma$ tels que:*

- $P_{\mathcal{S}_0}(s)$ *est vrai* ssi $s \in \mathcal{S}_0$,
- $P_{\mathcal{R}}(s,s')$ *est vrai* ssi $(s,s') \in \mathcal{R}$,
- $P_l(s,\ell)$ *est vrai* ssi $l(s) = \ell$.

Dans la suite, nous utilisons les notations $S_0(s)$, $R(s,s')$ et $L(s,\ell)$ à la place de $P_{\mathcal{S}_0}(s)$, $P_{\mathcal{R}}(s,s')$ et $P_l(s,\ell)$. Ainsi, une structure de Kripke symbolique est un triplet de prédicats $K = \langle S_0, R, L \rangle$.

Typiquement, les ensembles d'états de la structure de Kripke représentés symboliquement par les variables s et s' sont encodés par des conjonctions de variables booléennes. Par exemple, supposons $\mathcal{S} = \{0,1\}^3$, l'état $s = (1,0,1)$ est encodée par la conjonction $s_1 \bar{s}_2 s_3$ et l'ensemble d'états $\{(1,0,1),(1,1,1)\}$ est encodé par la conjonction $s_1 s_3$. En utilisant ce type d'encodage, S_0, R et L sont exprimés à l'aide de formules propositionnelles et peuvent donc être implémentées à l'aide de BDDs ou d'autres types de diagrammes de décision (comme les SDDs [24] utilisés dans notre implémentation).

Un TGBA peut lui aussi être encodé symboliquement par des prédicats [23].

Définition 7 (TGBA Symbolique) *Un TGBA $\langle Q, I, \delta, \mathcal{F} \rangle$ est symboliquement encodé par les prédicats $\langle I, \Delta, \{\Delta_f\}_{f \in \mathcal{F}} \rangle$ tels que:*

- $I(q)$ *est vrai* ssi $q \in I$,
- $\Delta(q,\ell,q')$ *est vrai* ssi $\big(\exists F \in 2^{\mathcal{F}}, (q,\ell,F,q') \in \delta\big)$,
- *Pour tout $f \in \mathcal{F}$, le prédicat Δ_f est défini par:*
  $\Delta_f(q,\ell,q')$ *est vrai* ssi $\big(\exists (q,\ell,F,q') \in \delta, f \in F\big)$.

En composant les représentations symboliques de la structure de Kripke et du TGBA, on obtient la définition suivante de leur produit synchronisé symbolique :

Définition 8 (Produit Synchronisé Symbolique) *Soit une structure de Kripke symbolique $K = \langle S_0, R, L \rangle$ et un TGBA symbolique $A = \langle I, \Delta, \{\Delta_f\}_{f \in \mathcal{F}} \rangle$, leur produit synchronisé symbolique $K \otimes A = \langle P_0, T, \{T_f\}_{f \in \mathcal{F}} \rangle$ est défini par les prédicats P_0, T et T_f encodant respectivement l'ensemble des états initiaux, la relation de transition et les transitions acceptantes, tels que:*

- (s,q) *encode les variables d'état du produit (s pour la Kripke et q pour le TGBA),*
- $P_0(s,q) = S_0(s) \wedge I(q)$,
- $T((s,q),(s',q')) = \exists \ell \big[R(s,s') \wedge L(s,\ell) \wedge \Delta(q,\ell,q')\big]$, *avec (s',q') encode les variables de l'état successeur,*
- $\forall f \in \mathcal{F}, T_f((s,q),(s',q')) = \exists \ell \big[R(s,s') \wedge L(s,\ell) \wedge \Delta_f(q,\ell,q')\big]$.

Dans cet définition du produit symbolique, la valuation ℓ est utilisé pour synchroniser la transition (q,ℓ,q') du TGBA avec l'état s de la kripke K telle que $L(s,\ell)$. Ceci permet de garantir que le produit symbolique reconnaît uniquement les exécutions de K acceptées par A.

3.6.2 Model Checking Symbolique basée sur TGTA

Dans cette section, nous proposons une approche symbolique basée sur les TGTA. nous commençons par présenter notre encodage symbolique des TGTA. Ensuite, nous proposons une adaptation de la structure de Kripke à l'approche TGTA et enfin nous montrons comment améliorer l'encodage du produit symbolique en exploitant les transitions bégayantes des TGTA.

L'encodage symbolique des TGTA est similaire à celui des TGBA, avec une différence dans l'encodage des transitions qui sont basées sur des changesets dans le cas des TGTA:

Définition 9 (TGTA Symbolique) *Un TGTA $\mathcal{T} = \langle Q, I, U, \delta, \mathcal{F} \rangle$ est encodé par les prédicats $\langle U_0, \Delta^{\oplus}, \{\Delta_f^{\oplus}\}_{f \in \mathcal{F}} \rangle$ tels que:*

- *$U_0(q, \ell)$ est vrai* ssi $(q \in I) \wedge (U(q) = \ell)$
- *$\Delta^{\oplus}(q, k, q')$ est vrai* ssi $\big(\exists F \in 2^{\mathcal{F}}, (q, k, F, q') \in \delta\big)$,
- *Pour chaque $f \in \mathcal{F}$, le predicat $\Delta_f^{\oplus}$ est défini par:*
  $\Delta_f^{\oplus}(q, k, q')$ est vrai ssi $\big(\exists (q, k, F, q') \in \delta, f \in F\big)$.

Le prédicat $U_0(q, \ell)$ encode l'ensemble des états initiaux et leur valuations. Le prédicat $\Delta^{\oplus}(q, k, q')$ encode l'ensemble des transitions du TGTA, avec la variable k encodant les changesets entre les pairs d'états successives q et q'. Enfin, pour chaque condition d'acceptation $f \in \mathcal{F}$, un prédicat $\Delta_f^{\oplus}(q, k, q')$ encode l'ensemble des transitions étiquetées par f.

Avant de définir le produit symbolique basé sur les TGTA, nous proposons une nouvelle variante de la structure de Kripke mieux adaptée à l'encodage de ce produit. Cette nouvelle variante est étiquetée par des changesets sur les transitions comme les TGTA.

Définition 10 (Structure de Kripke Symbolique basée sur les Changesets) *Une structure de Kripke $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, l \rangle$ peut être encoder une structure de kripke symbolique basée sur les changesets $K^{\oplus} = \langle S_0, R^{\oplus}, L \rangle$, telle que:*

- *Le prédicat $R^{\oplus}(s, k, s')$ est vrai* ssi $((s, s') \in \mathcal{R} \wedge (l(s) \oplus l(s')) = k)$,
- *Les prédicats S_0 et L sont définis de la même façon que la structure de kripke symbolique K de la Définition 6.*

Le produit symbolique entre un TGTA et une structure de Kripke est basé sur la synchronisation des changesets. Les transitions (q, k, q') d'un TGTA sont synchronisées avec les transitions entre les états s et s' d'une structure de Kripke tels que les valuations de s et s' diffèrent par le changeset k (i.e., $R^{\oplus}(s, k, s')$ est vrai). Formellement, on obtient la définition suivante:

Définition 11 (Produit Symbolique utilisant TGTA) *Soit une structure de Kripke symbolique basée sur les changesets $K^{\oplus} = \langle S_0, R^{\oplus}, L \rangle$ et un TGTA symbolique $A^{\oplus} = \langle U_0, \Delta^{\oplus}, \{\Delta_f^{\oplus}\}_{f \in \mathcal{F}} \rangle$, leur produit symbolique $K^{\oplus} \otimes A^{\oplus} = \langle P_0, T, \{T_f\}_{f \in \mathcal{F}} \rangle$ est défini par les prédicats suivants:*

- *L'ensemble des états initiaux est encodé par: $P_0(s, q) = \exists \ell \left[S_0(s) \wedge L(s, \ell) \wedge U_0(q, \ell) \right]$*
- *La relation de transition symbolique du produit est: $T((s, q), (s', q')) = \exists k \left[R^{\oplus}(s, k, s') \wedge \Delta^{\oplus}\ (q, k, q') \right]$*
- *La définition des T_f est similaire à T en remplaçant $\Delta^{\oplus}$ par $\Delta_f^{\oplus}$.*

3.6.3 Utilisation des transitions bégayantes des TGTA pour améliorer les performances de la Saturation symbolique

Dans ce section, nous proposons une amélioration des performances du model checking symbolique en exploitant la normalisation des transitions bégayantes dans un TGTA.

Parmi les techniques symboliques classiques, l'algorithme de saturation [7] permet d'obtenir des gains de plusieurs ordres de grandeur à la fois pour le temps d'exécution et la quantité de mémoire, en particulier lors de la vérification des systèmes asynchrones [8].

Nous montrons dans la suite que les performances de cette technique de saturation bénéficient grandement d'une séparation de la relation de transition du produit en deux termes, dont l'un d'eux exploite la normalisation des transitons bégayantes des TGTA.

Nous commençons par séparer les transitions bégayantes et les transitions non-bégayantes dans la relation de transition T du produit symbolique $K^{\oplus} \otimes A^{\oplus}$ (Définition 11):

$$T((s,q),(s',q')) = \big(R^{\oplus}(s,\emptyset,s') \wedge \Delta^{\oplus}(q,\emptyset,q')\big) \vee \big(\exists k \big[R^{\oplus}_{*}(s,k,s') \wedge \Delta^{\oplus}_{*}(q,k,q')\big]\big)$$

avec $R^{\oplus}_{*}$ et $\Delta^{\oplus}_{*}$ encodent respectivement les transitions non-bégayantes de la structure de Kripke et du TGTA:

- $\Delta^{\oplus}_{*}(q,k,q')$ est vrai *ssi* $\Delta^{\oplus}(q,k,q') \wedge (k \neq \emptyset)$
- $R^{\oplus}_{*}(s,k,s')$ est vrai *ssi* $R^{\oplus}(s,k,s') \wedge (k \neq \emptyset)$

D'après la Définition 9, le prédicat $\Delta^{\oplus}(q,\emptyset,q')$ est définie par l'équivalence:

$$\forall (q,q') \in Q^2 : \big[\Delta^{\oplus}(q,\emptyset,q') \Longleftrightarrow \big(\exists F \in 2^{\mathcal{F}}, (q,\emptyset,F,q') \in \delta\big)\big] \tag{1}$$

En plus, d'après la normalisation des transitions bégayantes de la Définition 5 de TGTA:

$$\forall (q,q') \in Q^2 : \big[\big(\exists F \in 2^{\mathcal{F}}, (q,\emptyset,F,q') \in \delta\big) \Longleftrightarrow (q = q')\big] \tag{2}$$

En combinant les deux équations (1) et (2), on obtient l'équivalence suivante :

$$\forall (q,q') \in Q^2 : \big[\Delta^{\oplus}(q,\emptyset,q') \Longleftrightarrow (q = q')\big] \tag{3}$$

En d'autres termes, le prédicat $\Delta^{\oplus}(q,\emptyset,q')$ encode l'ensemble de tous les boucles (self-loops) du TGTA et peut donc être remplacé par le prédicat identité: $equal(q,q')$. Nous obtenons ainsi la simplification suivante de l'expression de T :

$$T((s,q),(s',q')) = \underbrace{\big(R^{\oplus}(s,\emptyset,s') \wedge equal(q,q')\big)}_{T_{\emptyset}((s,q),(s',q'))} \vee \underbrace{\big(\exists k \, \big[R^{\oplus}_{*}(s,k,s') \wedge \Delta^{\oplus}_{*}(q,k,q')\big]\big)}_{T_{*}((s,q),(s',q'))} \tag{4}$$

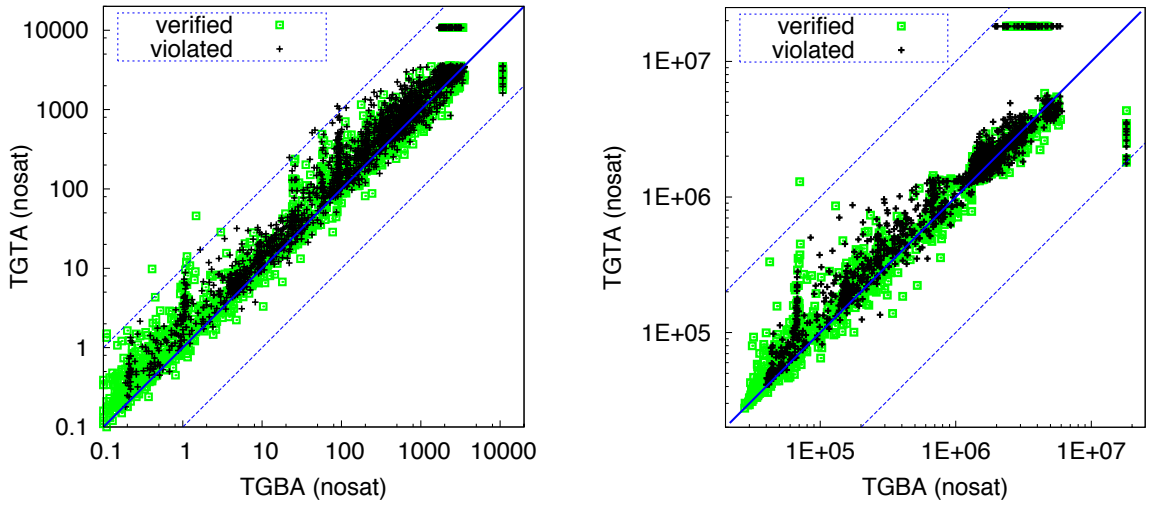
Dans la relation de transition de l'équation (4), le terme $T_{\emptyset}$ correspondant à la synchronisation des transitions bégayantes ne dépend pas de l'état du TGTA. En effet, dans ce terme, le prédicat identité $equal(q,q')$ sera ignoré [8] par l'algorithme de saturation (i.e., le terme $T_{\emptyset}$ est appliqué sans modifier la variable q [18]). Cette simplification favorise grandement les performances de la saturation comme nous allons le voir dans l'évaluation expérimentale de la section suivante.

3.6.4 Evaluation Expérimentale de l'approche TGTA Symbolique

Les résultats de notre évaluation expérimentale sont présentés par les Figures 9 et 10. Les nuages de points de la Figures 9 comparent les performances (temps d'exécution à gauche et quantité de mémoire à droite) entre les approches symboliques basée sur TGBA et TGTA. Cette première comparaison est effectuée avec la technique de saturation désactivée. Les nuages de points de la Figures 10 affichent les résultats d'une deuxième comparaison effectuée dans le cas où la saturation a été activée pour les deux approches. Ces figures permettent de montrer l'influence de la combinaison de la saturation avec les deux approches TGBA et TGTA.

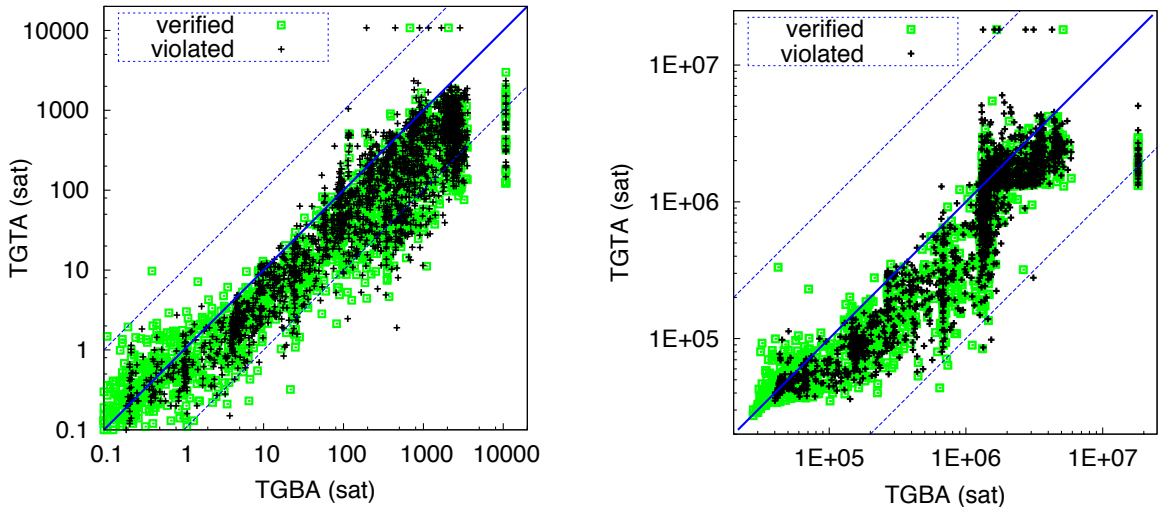
Chaque point des figures représente une mesure pour une paire $(modle, formule)$. Les axes sont affichés en échelle logarithmique. Les couleurs distinguent les formules satisfaites (carrés verts) et non-satisfaites (croix noires).

Les Figure 9 et 10 montrent que grâce à la combinaison de la saturation et de l'optimisation basée sur les transitions bégayantes (Section 3.6.3), l'approche TGTA est clairement meilleure que l'approche traditionnelle TGBA par au moins un ordre de grandeur.



(a) Time (b) Memory

Figure 9: Comparaison des approches TGBA et TGTA, avec la saturation désactivée “(nosat)”.



(a) Time (b) Memory

Figure 10: Comparaison des approches TGBA et TGTA, avec la saturation activée “(sat)”.

3.7 Trois approches Hybrides basées sur les TGTA

Dans le dernier chapitre, nous avons combiné les TGTA avec trois approches hybrides de model checking.

Le model checking hybride [23, 12] combine des idées des deux approches explicite et symbolique.

Dans une approche hybride, l'automate de la propriété est généralement représenté par un graphe explicite car dans la plupart des cas sa taille n'est pas très grande (et en plus il existe dans l'état de l'art plusieurs optimisations efficaces de ces automates explicites). Cependant, l'espace d'état du modèle est en générale très grand et doit donc être encodé symboliquement.

Dans ce travail, nous nous intéressons aux trois techniques [11] hybrides suivantes : SOG, SOP et SLAP.

La technique SOG (*Symbolic Observation Graph*) exploite le fait qu'uniquement une partie des propositions atomiques du modèle sont observées par la propriété LTL à vérifier. Une structure SOG est une abstraction d'une structure de Kripke dans laquelle les états successives sont agrégés tant qu'ils partagent les mêmes valeurs des propositions atomiques observées (par la propriété LTL).

SOP (*Symbolic Observation Product*) permet d'agréger dynamiquement plus d'états que SOG, ceci en exploitant le fait que le nombre de propositions atomiques observées décroit en progression dans l'exploration de l'automate de la propriété LTL.

Similairement à SOP, SLAP (*Self-Loop Aggregation Product*) est un graphe d'agrégats alternatif au produit synchronisé traditionnel. Dans un SLAP, les états de la structure de Kripke sont agrégés en fonction des valuations des boucles (self-loops) de l'automate de la propriété LTL.

Les trois approches SOG, SOP et SLAP sont basées [11] sur TGBA. Dans ce travail, nous avons défini et implémenté des variantes de ces trois approches en se basant sur TGTA (à la place de TGBA). Ensuite, nous avons expérimentalement comparé les performances de chaque approche hybride (SOG, SOP et SLAP) contre sa variante basée sur TGTA (SOG-TGTA, SOP-TGTA et SLAP-TGTA).

3.7.1 Comparaison SOG vs. SOG-TGTA

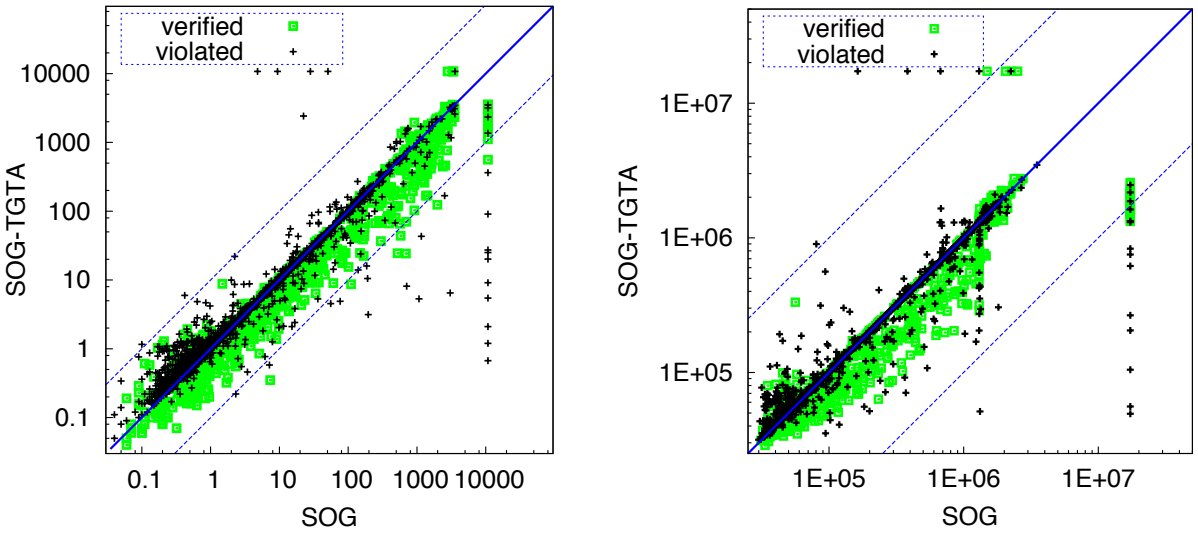


Figure 11: Comparaison des performances de SOG contre SOG-TGTA. À gauche: le temps d'exécution (en secondes); À droite: la quantité de mémoire (en kilobytes).

Les nuages de points de la Figure 11 comparent les performances des deux approches hybrides SOG et SOG-TGTA. Chaque point du nuage à gauche et à droite compare respectivement le temps d'exécution et la quantité de mémoire utilisés pour le model checking de chaque paire "(formule, modéle)". Les points au dessous de la diagonale correspondent aux cas où l'approche SOG-TGTA est meilleure. Les points représentés par des carrés verts correspondent aux formules satisfaites et les croix noires aux formules non satisfaites. Cependant, pour les formules non satisfaites, l'approche SOG est meilleure mais uniquement dans les cas où le temps d'exécution est faible (inférieur à une seconde). Pour les cas difficiles, il y a plus d'échec pour l'approche SOG.

3.7.2 Comparaison SOP vs. SOP-TGTA

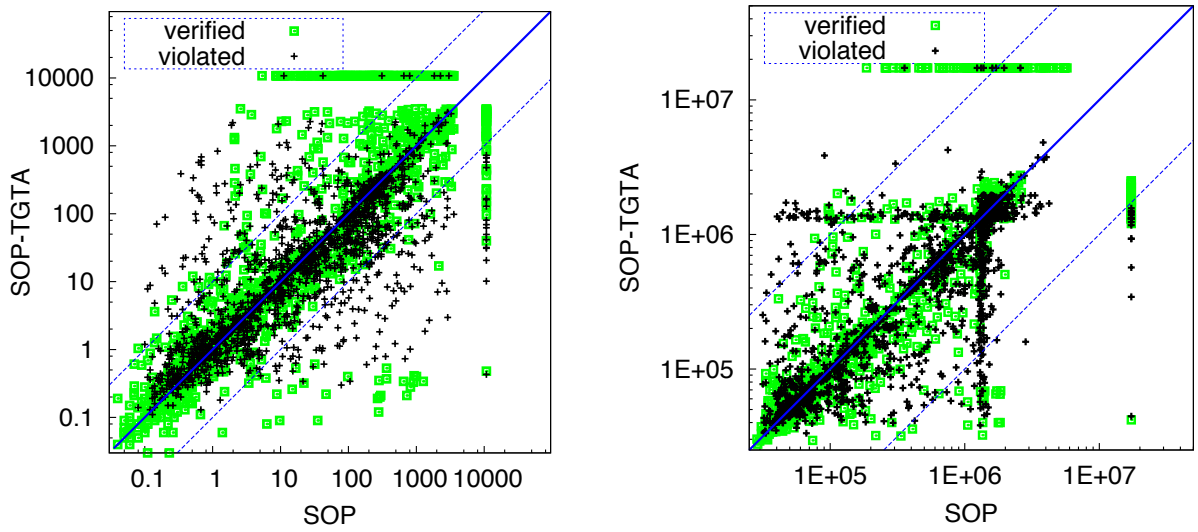


Figure 12: Comparaison des performances de SOP contre SOP-TGTA. À gauche: le temps d'exécution (en secondes); À droite: la quantité de mémoire (en kilobytes).

Les nuages de points de la Figure 12 comparent les performances entre SOP et SOP-TGTA. Les résultats sont difficiles à interpréter, car il y a beaucoup de points où SOP est meilleure et beaucoup d'autres points où c'est SOP-TGTA qui est le meilleur. Cependant, si nous analysons plus en détail les nuages de points, nous pouvons observer qu'il y a plus de cas d'échec pour SOP-TGTA, ces cas sont représentés par le nuage linéaire en haut.

3.7.3 Comparaison SLAP vs. SLAP-TGTA

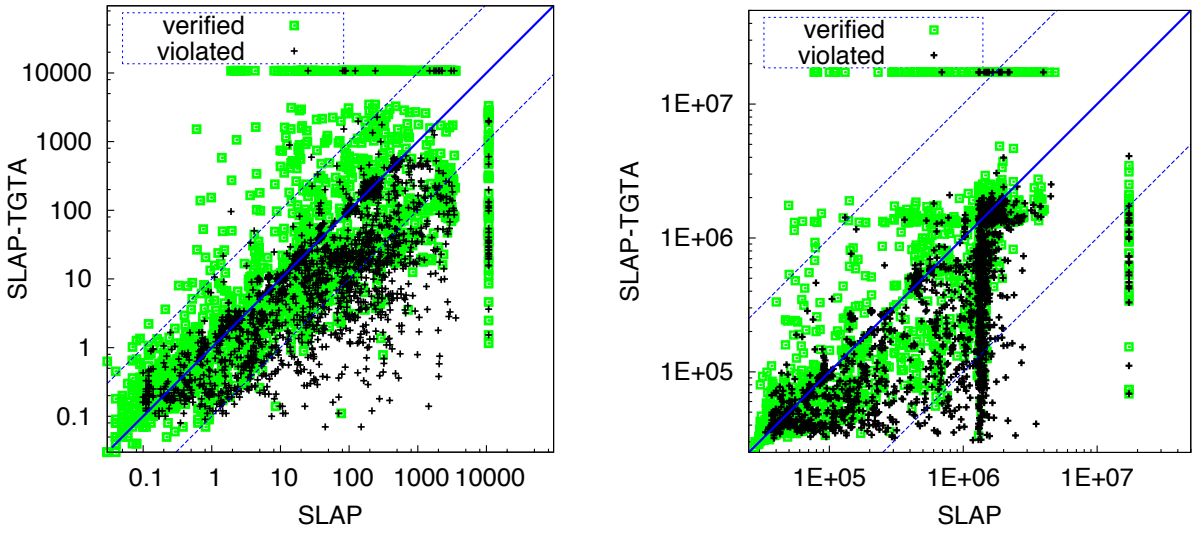


Figure 13: Comparaison des performances de SLAP contre SLAP-TGTA. À gauche: le temps d'exécution (en secondes); À droite: la quantité de mémoire (en kilobytes).

La Figure 13 présente les nuages de points comparant SLAP et SLAP-TGTA. Les points au dessous de la diagonale correspondent aux cas où l'approche SLAP-TGTA est meilleure.

L'interprétation des résultats dépends des couleurs des points. Pour les croix noires correspondant aux formules non satisfaites, l'approche SLAP-TGTA est plus efficace que l'approche SLAP dans la majorité des cas. Pour les points carrés verts, les résultats sont plus difficiles à interpréter, il y a beaucoup de points des deux cotés de la

diagonale. Globalement, les deux approches SLAP et SLAP-TGTA sont complémentaires, et peuvent donc être exécutés en parallèle dans le but de récupérer le résultat de l'approche la plus rapide des deux.

4 Conclusion

L'approche par automates du model-checking est la plus classique des approches de vérification automatique. Elle prend en entrée un modèle du système et une propriété, et permet de savoir si cette dernière est vérifiée. Pour cela un model-checker traduit la négation de la propriété en un automate et vérifie si le produit du système et de cet automate est vide. Hélas, bien qu'automatique, cette approche souffre d'une explosion combinatoire du nombre d'états du produit obtenu.

Afin de combattre ce problème, en particulier lors de la vérification de formules insensibles au bégaiement, nous proposons la première évaluation d'automates testeur (TA) sur des modèles réalistes, une amélioration de l'algorithme de vérification pour ces automates et une méthode permettant de transformer un TA en un automate (STA) permettant une vérification en une seule passe. Nous proposons aussi une nouvelle classe d'automates: les TGTA. Ces automates permettent une vérification en une seule passe sans ajouter d'états artificiels. Cette classe combine les avantages des TA et des TGBA. Les TGTA permettent d'améliorer les approches explicite et symbolique de model-checking. Notamment, en combinant les TGTA avec la technique de saturation, les performances de l'approche symbolique sont améliorées d'un ordre de grandeur par rapport aux TGBA. Utilisés dans l'approche hybride les TGTA se révèlent complémentaires aux TGBA.

Une piste pour améliorer l'approche TGTA serait de la combiner avec la technique d'ordre partiel.

References

[1] T. Babiak, M. Křetínský, V. Řehák, and J. Strejček. LTL to Büchi automata translation: Fast and more deterministic. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, volume 7214 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2012.

[2] A. E. Ben Salem, A. Duret-Lutz, and F. Kordon. Model Checking using Generalized Testing Automata. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, ?:?, 2012.

[3] A. E. Ben Salem, A. Duret-Lutz, and F. Kordon. Model Checking using Generalized Testing Automata. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC VI)*, 7400:94–122, 2012.

[4] A. E. Ben Salem, A. Duret-Lutz, F. Kordon, and Y. Thierry-Mieg. Symbolic Model Checking of stutter invariant properties Using Generalized Testing Automata. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume ? of *Lecture Notes in Computer Science*, page to be published, Grenoble, France, April 2014. Springer.

[5] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, Berkley, 1960*, pages 1–11. Standford University Press, 1962.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[7] G. Ciardo, R. M. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 379–393. Springer-Verlag, 2003.

[8] G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Proceedings of the 13 IFIP WG 10.5 international conference on Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 146–161. Springer-Verlag, 2005.

[9] J.-M. Couvreur. On-the-fly verification of temporal logic. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, Toulouse, France, Sept. 1999. Springer-Verlag.

[10] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In P. Godefroid, editor, *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, volume 3639 of *Lecture Notes in Computer Science*, pages 143–158. Springer, Aug. 2005.

[11] A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg. Combining explicit and symbolic approaches for better on-the-fly LTL model checking. Technical Report 1106.5700, arXiv, June 2011. Extended version of our ATVA'11 paper, presenting two new techniques instead of one.

[12] A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg. Self-loop aggregation product — a new hybrid approach to on-the-fly LTL model checking. In *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11)*, volume 6996 of *Lecture Notes in Computer Science*, pages 336–350, Taipei, Taiwan, Oct. 2011. Springer.

[13] A. Duret-Lutz and D. Poitrenaud. SPOT: an extensible model checking library using transition-based generalized Büchi automata. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 76–83. IEEE Computer Society Press, Oct. 2004.

[14] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, 2001. Springer-Verlag.

[15] J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In *Proceedings of the 13th International SPIN Workshop (SPIN'06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2006.

[16] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th Workshop on Protocol Specification Testing and Verification (PSTV'95)*, pages 3–18, Warsaw, Poland, 1996. Chapman & Hall.

[17] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In D. Peled and M. Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326, Houston, Texas, Nov. 2002.

[18] A. Hamez, Y. Thierry-Mieg, and F. Kordon. Hierarchical set decision diagrams and automatic saturation. In *Proceedings of the 29th international conference on Applications and Theory of Petri Nets*, PETRI NETS '08, pages 211–230. Springer-Verlag, 2008.

[19] H. Hansen, W. Penczek, and A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. In R. Cleaveland and H. Garavel, editors, *Proceedings of the 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (FMICS'02)*, volume 66(2) of *Electronic Notes in Theoretical Computer Science*, Málaga, Spain, July 2002. Elsevier.

[20] MoVe/LRDE. The Spot home page: `http://spot.lip6.fr`, 2014.

[21] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, Sept. 1995.

[22] A.-E. B. Salem, A. Duret-Lutz, and F. Kordon. Generalized Büchi automata versus testing automata for model checking. In *Procedings of the 2nd workshop on Scalable and Usable Model Checking for Petri Nets and other models of Concurrency (SUMo'11)*, volume 726, pages 65–79, Newcastle, UK, June 2011. CEUR.

[23] R. Sebastiani, S. Tonetta, and M. Y. Vardi. Symbolic systems, explicit properties: on hybrid approches for LTL symbolic model checking. In K. Etessami and S. K. Rajamani, editors, *Proceedings of 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 350–363, Edinburgh, Scotland, UK, July 2005. Springer.

[24] Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical set decision diagrams and regular models. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2009.

[25] A. Valmari. Bisimilarity minimization in $O(m \log n)$ time. In *Proceedings of the 30th International Conference on the Applications and Theory of Petri Nets (ICATPN'09)*, volume 5606 of *Lecture Notes in Computer Science*, pages 123–142. Springer, 2009.