Thèse d'habilitation de l'Université Pierre et Marie Curie
spécialité informatique

présentée par

Alexandre Duret-Lutz

pour l'obtention d'une

habilitation à diriger des recherches

Contributions to LTL and ω-Automata for Model Checking

soutenue le 10 février 2017 devant un jury composé de :

M. **Javier Esparza**, Professor (Technische Universität München, Germany)	Rapporteur
M. **Radu Mateescu**, Directeur de recherche (INRIA Grenoble, France)	Rapporteur
M. **Moshe Y. Vardi**, Professor (Rice University, Houston, Texas, USA)	Rapporteur
M. **Rüdiger Ehlers**, Professor (Universität Bremen, Germany)	Examinateur
M. **Stephan Merz**, Directeur de recherche (INRIA Nancy & LORIA, France)	Examinateur
M. **Jaco van de Pol**, Professor (University of Twente, Netherlands)	Examinateur
M. **Fabrice Kordon**, Professeur (UPMC, France)	Examinateur

Contents

Acknowledgments

I am grateful to Étienne Renault, Maximilien Colange, and Jim Newton for their helpful comments on earlier versions of this document; and to Olivier Ricou, Fabrice Kordon, and Thierry Gérault for pressing me to finish it.

The work summarized here is the result of collaborations with many people. As the reader shall soon discover, I have made arrangements so that those co-authors cannot be ignored while reading the document. Nonetheless, let me express some particular thanks to some colleagues from Brno (Jan Strejček, Tomáš Babiak, František Blahoudek) and Dresden (Joachim Klein and David Müller), who have inspired several recent developments, and with whom I enjoy working, even if I had less time to do so recently.

1 Introduction

This document is a synthesis of the work I have done since my Ph.D. Because of size constraints, it is written as a teaser, trying to lure the reader into reading the cited papers to get the details.[1]

[1] All these papers can be downloaded from `https://www.lrde.epita.fr/~adl/dl/adl.html`.

The citations in this report have received a special treatment: references to papers I co-authored are shown in the margins inside such a blue frame, while third-party papers only appear in the final bibliography. In addition to separating my contributions from the existing literature, this presentation also shows that these are not precisely my contributions, but the results of collaborative work with several people.

Since this document is just baiting material, high-level explanations and examples have been preferred over formal definitions, and the reader is assumed to have some basic knowledge of linear-time temporal logic (LTL) and ω-automata. Most of the work presented revolves around the automata-theoretic approach for LTL model checking (presented in the next section), and is implemented in the Spot library (presented in the following section).

1.1 *Automata-theoretic approach to LTL model checking*

The automata-theoretic approach to LTL model checking [Vardi, 2007] is a way to decide whether a model M satisfies an LTL formula φ. The model M is first converted into an automaton (usually a Kripke structure) K_M whose language $\mathscr{L}(K_M)$ represents the set of all (infinite) behaviors of M. The negation of the formula φ is converted into an automaton $A_{\neg\varphi}$ whose language $\mathscr{L}(A_{\neg\varphi})$ captures the forbidden behaviors. With these objects, testing whether M satisfies φ amounts to checking the emptiness of the product $K_M \otimes A_{\neg\varphi}$ of these two automata: if $\mathscr{L}(K_M \otimes A_{\neg\varphi}) = \emptyset$, then $M \models \varphi$.

In LTL model checking, words are infinite, and K_M and $A_{\neg\varphi}$ are ω-automata. This approach can be declined in various ways depending on the nature of these two automata, and how we implement their product and its emptiness check.

In what follows, we will assume that K_M is always a Kripke structure, but we consider various natures for $A_{\neg\varphi}$, different products, and different emptiness checks (Fig. 1.1).

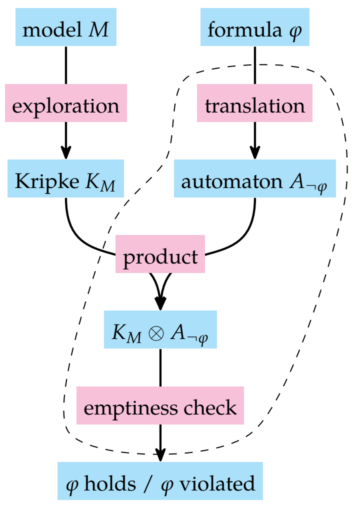

model M
formula φ
exploration
translation
Kripke K_M
automaton $A_{\neg\varphi}$
product
$K_M \otimes A_{\neg\varphi}$
emptiness check
φ holds / φ violated

Figure 1.1: The automata-theoretic approach to LTL model checking. The dashed curve shows the parts we focus on.

1.2 *Spot*

Spot is a C++ library of model-checking algorithms that we first presented in 2004[2]. It contains algorithms to perform the steps discussed in the previous section. It was purely a library until Spot 1.0, when we started distributing command-line tools for LTL manipulation[3] and translation of LTL to some generalizations of Büchi automata.

[2] A. Duret-Lutz and D. Poitrenaud. SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In MASCOTS'04, pp. 76–83. IEEE Computer Society Press, 2004

[3] A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In ATVA'13, vol. 8172 of LNCS, pp. 442–445. Springer, 2013

The latest version[4], Spot 2.0, is a very large rewrite of the core of the library, in C++11, with a focus on supporting automata with arbitrary acceptance conditions, as we shall discuss in Chapter 4.

[4] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In ATVA'16, vol. 9938 of LNCS, pp. 122–129. Springer, 2016b

Spot now offers convenient command-line tools and Python binding to access most of its algorithms. A large part of my time over the last years has been spent trying to make these tools robust, useful, and accessible. Even if this sounds like engineering, building such a framework actually enables both research and teaching. Today, we have reached the point where several research teams are using Spot in part of their research or experiments[5], and we have started using it at Epita for teaching LTL, ω-automata and model checking.

[5] Cf. Appendix A, p. 51.

Spot is a free software that can be downloaded from `https://spot.lrde.epita.fr/`. There is also a live installation of IPython/Jupyter (a web application for interactive programming [Pérez and Granger, 2007]) that can be used to experiment with Spot's command-line tools or Python bindings without installing anything: see `http://spot-sandbox.lrde.epita.fr/`.

To emphasize that what we describe is not only implemented, but also easily reusable, this document is littered with examples using Spot from the command line. These examples are shown on a yellow background, as this paragraph.

Spot provides the following tools, that will be used in the examples:

`randltl` generates random LTL/PSL[6] formulas,
`genltl` generates LTL formulas from scalable patterns,
`ltlfilt` filters, converts, and transforms LTL/PSL formulas,
`ltl2tgba` translates LTL/PSL formulas into generalized Büchi automata or deterministic parity automata,
`ltl2tgta` translates LTL/PSL formulas into testing automata,
`ltlcross` cross-compares LTL/PSL-to-automata translators to find bugs,
`ltlgrind` mutates LTL/PSL formulas to help reproduce bugs on smaller ones,
`dstar2tgba` converts the Rabin or Streett automata output by the `ltl2dstar` tool [Klein and Baier, 2006] into generalized Büchi automata,
`randaut` generates random ω-automata,
`autfilt` filters, converts, and transforms ω-automata,
`ltldo` runs LTL/PSL formulas through other translators, providing uniform input and output interfaces.

[6] PSL is a reference to the linear fragment of the Property Specification Language [Eisner and Fisman, 2006] which will only be mentioned in Section 7.2 (p. 40).

1.3 *Overview of this document*

Besides the introduction and conclusion, this habilitation thesis contains five core chapters. Each of them summarizes several related articles I co-authored.

In Chapter 2 we discuss how to translate φ into $A_{\neg\varphi}$, either as a Transition-based Generalized Büchi Automaton, a Büchi automaton, or a monitor. We summarize all the algorithms that have been implemented to make Spot's `ltl2tgba` one of the best off-the-shelf LTL translator tools.

In Chapter 3 we discuss various ways to test the emptiness of the above automata. This chapter was the topic of a Ph.D. I co-supervised [Renault, 2014], and introduces a new emptiness check procedure based on the union-find data structure.

In Chapter 4 we discuss a generic acceptance condition that we introduced in the Hanoi Omega-Automata (HOA) format in order to favor the interactions between ω-automata tools. As these acceptance conditions supersede all conditions discussed previously, we present some of the acceptance transformations and adjustments we had to use to support algorithms over arbitrary acceptance in Spot.

In Chapter 5 we discuss a SAT-based technique for minimizing deterministic automata with arbitrary acceptance conditions.

In Chapter 6 we discuss several results related to stutter-invariant properties: how to test them, and how to implement the entire automata-theoretic approach using another type of automata called "testing automata". The second part of this chapter was the topic of a another Ph.D. I co-supervised [Ben Salem, 2014].

Some perspectives and omitted topics are gathered in Chapter 7. Finally, a list of selected uses of Spot is provided in Appendix A.

2
Translation improvements

2.1 *Transition-based and generalized Büchi acceptance*

Figure 2.1 shows three equivalent deterministic ω-automata that are minimal for their respective acceptance conditions.

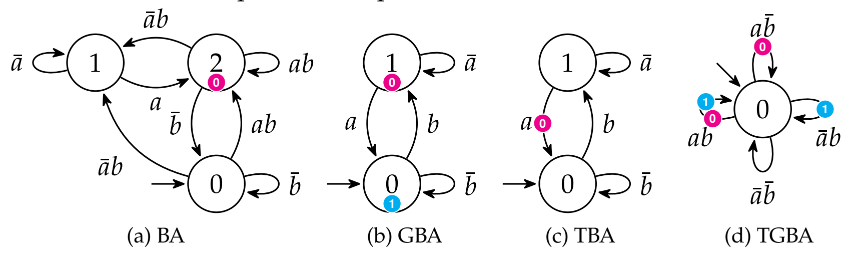


Figure 2.1: Minimal deterministic automata recognizing the LTL formula $\mathsf{GF}a \wedge \mathsf{GF}b$. These automata recognize infinite words over the alphabet $\{\bar{a}\bar{b}, \bar{a}b, a\bar{b}, ab\}$ (i.e., all possible assignments of a and b). For conciseness, edges are labeled by Boolean formulas over a and b: they match all compatible assignments.

Small acceptance marks like ⓿ or ❶ are used to denote the membership of the states or transitions to different acceptance sets. In automata using state-based acceptance (Fig. 2.1(a)–(b)), these marks are on states. Automata with transition-based acceptance have them on transitions (Fig. 2.1(c)–(d)).

In an automaton with Büchi acceptance (Fig. 2.1(a) and (c)), a run is accepting if it visits infinitely many states or transitions with the mark ⓿. Runs of automata with generalized Büchi acceptance (Fig. 2.1(b) and (d)) have to visit one ⓿ and one ❶ infinitely often. Generalized Büchi automata can use arbitrarily many acceptance marks; but in this particular example using more than 2 would not reduce the number of states.

We abbreviate Büchi automaton as BA, and prefix it with G or T to denote generalized or transition-based acceptance.

Any of these automaton types can be used as $A_{\neg\varphi}$ in the model checking approach outlined in Fig. 1.1. However, the number of states of the product $S = |K_M \otimes A_{\neg\varphi}|$ is such that $S \leq |K_M| \cdot |A_{\neg\varphi}|$ so it seems better to pick the smallest automaton possible. Example 2.2 shows how to compute these sizes with a random K_M using Spot.

```
$ ltl2tgba 'GFa&GFb' >tgba
$ ltl2tgba -B 'GFa&GFb' >ba
$ randaut -Q100 -e.05 a b >k
$ autfilt --product=k tgba ba\
> --stats='%F: %s st, %e ed'

tgba: 100 st, 589 ed
ba: 284 st, 1681 ed
```

Example 2.2: Commands for computing the size of $K_M \otimes A_{\neg\varphi}$ when K_M is a random 100-state automaton with an edge density of 0.05 (each state has 5 successors on average), and K_M is either a TGBA (Fig. 2.1(d)) or a BA (Fig. 2.1(a)).

Clearly, BAs are a particular case of GBAs. Furthermore, any automaton with state-based acceptance can easily be converted into an automaton with transition-based acceptance without changing its transition structure: it suffices to push the marks of each state to all their outgoing transitions (Fig. 2.3). With this interpretation, we can consider automata with state-based acceptance as particular cases of automata with transition-based acceptance.

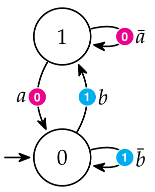


Figure 2.3: A TGBA interpretation of the state-based GBA of Fig. 2.1(b).

Finally, a TGBA with n marks and s states can be converted into a TBA with at most $n \cdot s$ states or into a BA with at most $(n+1) \cdot s$ states, using a degeneralization procedure we will discuss later. This shows that all these automata are as expressive, but that TGBAs can be more concise. For this reason, most of our work was focused on transition-based acceptance.[1]

[1] Spot actually supports only transition-based acceptance. When it outputs automata with state-based acceptance, it is just pretending: internally those automata are stored using transition-based marks, as in Figure 2.3.

2.2 *Pros and cons of transition-based acceptance*

From an implementation point of view, using transition-based acceptance requires more memory, because acceptance marks have to be stored on transitions, which are more numerous than states[2]. This is true as long as the automaton is explicitly stored in memory, and this cost cannot always be compensated by the smaller size of automata with transition-based acceptance. However, when implementing an explicit model checker using the automata-theoretic approach of Fig. 1.1, one should keep in mind that the product of $K_M \otimes A_{\neg\varphi}$ is computed on the fly, as needed by the emptiness-check procedure, and that its transitions need not be stored: only the states of the product have to be kept in order to remember which ones were already visited. In that context, the only memory cost of using transition-based acceptance marks is in the representation of $A_{\neg\varphi}$, which is negligible compared to the size of $K_M \otimes A_{\neg\varphi}$.

[2] Except in some degenerate cases, we work with automata that have at least one outgoing transition per state.

Using transition-based acceptance often leads to simpler or more natural algorithms. This observation has been made several times in the past: for instance it is discussed by Michel [1984], Kurshan [1987], Le Saëc and Litovsky [1994], Giannakopoulou and Lerda [2002], and Varghese [2014]. After having taken several algorithms originally introduced in a state-based setting and implemented them in a transition-based framework, I can only second these observations.

However, we should keep in mind that there is a large subset of properties that do not benefit from transition-based Büchi acceptance in any way. (Inherently) Weak automata are automata in which any strongly connected component (SCC) contains only accepting cycles, or only non-accepting cycles. In these automata, the acceptance condition could be expressed directly in terms of SCCs rather than states or transitions: an infinite run is accepting if it ultimately stays in one of the accepting SCCs. There is also nothing to be gained by using generalized Büchi acceptance in weak automata, or by trying to represent the same properties using non-weak Büchi automata. This class of automata represents properties from the persistence class defined by Manna and Pnueli [1990] (Fig. 2.4), and can be associated with a syntactic fragment of LTL [Černá and Pelánek, 2003].

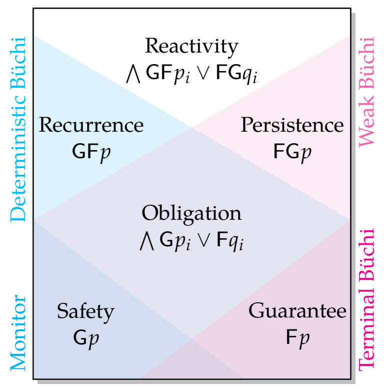


Figure 2.4: The temporal hierarchy of Manna and Pnueli [1990], labeled by canonical LTL properties where p can be any subformula that uses only Boolean operators, X, or past LTL operators. Any persistence property (this includes the obligation, safety, and guarantee subclasses) can be represented by a weak Büchi automaton and cannot benefit from transition-based Büchi acceptance.

```
$ genltl --dac |\
> ltlfilt --format='%[v]h' |\
> sort | uniq -c
      1 guarantee
      2 obligation
      1 persistence
      2 reactivity
     12 recurrence
     37 safety
```

Example 2.5: Classification of the 55 LTL formulas of Dwyer et al. [1998] into their most precise class of the temporal hierarchy. Of these formulas, only 14 (12 recurrence and 2 reactivity properties) may not be represented by weak Büchi automata.

2.3 *Benchmark of BA translators*

Before we discuss how we translate LTL into TGBA or BA, let us make a quick review of existing tools. Most off-the-shelf translators focus on producing (state-based) Büchi automata, because that is what is still commonly used.

We consider the following translators:

`spin`	6.4.5	[Holzmann, 2003]
`ltl2ba`	1.2b1	[Gastin and Oddoux, 2001]
`modella`	1.5.9[3]	[Sebastiani and Tonetta, 2003]
`trans`	Mar.2013[3]	[Mochizuki et al., 2014]
`ltl3ba`	1.1.3	[Babiak et al., 2012]
`ltl2tgba`	Spot 2.3	[Duret-Lutz, 2014]

[3] Note that these versions of `modella` and `trans` are both known to be bogus: `modella` mistranslates one formula from our benchmark while `trans` produces correct results for this benchmark but mistranslates some other formulas.

`ltl2ba` was created as a drop-in replacement for the translator of `spin`, and reuses part of its source. `modella`'s motto was that larger automata are OK as long as they are more deterministic, because improved determinism should benefit the product performed by a model checker. `ltl3ba` is a drop-in replacement for `ltl2ba` where the core of the translation has been rewritten and some optimization were added. In particular it tries to favor more deterministic automata by default (this can be disabled with option `-M0`). `ltl2tgba` is our own translator: by default it favors small automata over deterministic ones, but this can be changed with option `-D`. Finally, `trans` is a more recent translator that, unlike `ltl2ba`, `ltl3ba` and `ltl2tgba`, does not use transition-based acceptance internally.

Table 2.7 compares these tools on the production of BAs. For a given formula φ and a tool t we compute a BA A^t_φ and count its number of states, non-deterministic states, edges, and transitions. The number of transitions is obtained by counting each edge for the number of compatible assignments of all atomic propositions.[4] To measure the behavior of A^t_φ in a model checking context, we compute 100 products $M_i \otimes A^t_\varphi$, using 100 randomly generated automata $(M_i)_{1\leq i\leq 100}$ (the same automata are used for all tools), and retain the median size of these products. For each tool, the table sums all the sizes over 178 formulas taken from the literature.

[4] An edge labeled by a stands for two transitions labeled by ab and $a\bar{b}$. For instance, the BA in Fig. 2.1(a) has 8 edges that represent 12 transitions.

```
$ genltl --dac --sb --eh |
> ltlcross -T60 --prod=+100 \
> --csv=output.csv \
> spin ltl2ba \
> 'modella -r12 -g -e' \
> 'trans -f %[e]s >%O' \
> 'ltl3ba -M0' ltl3ba \
> 'ltl2tgba -s' \
> 'ltl2tgba -s -D'
```

Example 2.6: This command produces a CSV file (`output.csv`) containing all data summarized in Table 2.7. In addition `ltlcross` will cross-compare all produced automata and search for bogus translations.

Spot's `ltl2tgba` produces Büchi automata that are noticeably smaller and more deterministic than those produced by other tools, but the additional simplifications it performs have a cost. It takes nearly two seconds to translate all formulas, and that is three times the time needed by `ltl3ba`.

Table 2.7: Comparison of the Büchi automata produced by various translators from 89 formulas (and their negation) taken from the literature. Column # shows how many formulas could be translated within the 60 seconds timeout; except for that column, smaller values are always better. *ndet* counts the number of **n**on-**det**erministic automata produced.

				automaton size				product size	
	#	ndet	time (s)	states	ndstates	edges	trans	states	trans
`spin`	167	162	220.66	1440	1236	7238	46033	259313	9433430
`ltl2ba`	178	169	0.32	1000	801	3338	29974	190898	5616566
`modella -r12 -g -e`	178	109	18.47	1244	577	4116	23474	210494	4033414
`trans`	178	119	0.46	957	398	4674	16798	172246	3276714
`ltl3ba -M0`	178	167	0.72	794	591	2193	21170	150776	4325221
`ltl3ba`	178	115	0.72	829	307	2389	14322	155220	2913043
`ltl2tgba -s`	178	49	1.85	666	102	1643	10346	129419	2399328
`ltl2tgba -s -D`	178	44	1.87	671	96	1667	10456	129804	2401471

2.4 *Translating LTL formulas into Büchi automata: an overview*

The operations chained in Spot to translate an LTL formula into a Büchi automaton can be grouped into four steps:

1. Simplify the LTL formula syntactically. E.g., rewrite $\mathsf{FF}a$ (a 3-state automaton) into $\mathsf{F}a$ (2 states).
2. Translate the simplified formula into a TGBA.
3. Post-process the resulting TGBA, e.g., by pruning useless SCCs, or running various simulation-based reductions or minimizations.
4. If desired (and still needed after the previous post-processing) degeneralize the TGBA into a Büchi automaton.

These steps are summarized in the next four sections, but more details can be found in our IJCCBS paper[5].

[5] A. Duret-Lutz. LTL translation improvements in Spot 1.0. Int. J. on Critical Computer-Based Systems, 5 (1/2):31–54, 2014

2.5 *LTL simplifications*

Spot implements several types of rewritings:

Trivial identities are applied at any time during the construction of a formula (e.g., while they are parsed). These are all based on idempotence of some operators (e.g., $\mathsf{FF}a \equiv \mathsf{F}a$), or neutral/absorbent operands (e.g., $\mathsf{X}\bot \equiv \bot$, $f \land \bot \equiv \bot$, $f \land \top \equiv \top$, etc.).

Basic rewritings are unconditional rewriting rules, like moving the X operators to the front (as in $\mathsf{GX}f \equiv \mathsf{XG}f$).

Eventual and universal rewritings apply only when some subformulas are *purely universal* or *pure eventualities* [Etessami and Holzmann, 2000], or are what Babiak et al. [2012] have called *alternating* formulas. As an example $\mathsf{FGF}a$ can be rewritten as $\mathsf{GF}a$ because the latter is a pure eventuality.

Implication-based rewritings apply only in cases where one subformula can be shown to imply another subformula. For instance, under the hypothesis that $f \to g$, we have $f \mathbin{\mathsf{U}} g \equiv g$. There are two ways to detect such implications: they can be approximated syntactically [Somenzi and Bloem, 2000], or decided exactly using automata-based language containment checks [Tauriainen, 2006].

```
$ ltlfilt --simplify \
> -f 'a U (b | G(a) | c)'
a W (b | c)
```

Example 2.8: `ltlfilt --simplify` will apply all rules to simplify the given formula(s). This could be used as a preprocessor for another tool.

Not all the rewriting rules found in the literature are necessarily good to apply[6]: some were written with the goal of reducing the size or the depth of the formula, but these metrics do not always correlate with the size of the automaton. For instance the rewriting of $\mathsf{F}(\varphi \land \mathsf{GF}\psi)$ into $(\mathsf{F}\varphi) \land (\mathsf{GF}\psi)$ suggested by Somenzi and Bloem [2000] is actually harmful in our context: our translation works better with the first formula, where it is clear that $\mathsf{GF}\psi$ does not need to be checked before φ holds.

[6] An exhaustive list of all simplification rules implemented in Spot can be found in `https://spot.lrde.epita.fr/tl.pdf`

2.6 *Translating LTL to TGBA*

The core translation algorithm implemented is based on an original algorithm by Couvreur [1999] that uses binary decision diagrams

(BDDs) to represent the outgoing transition of each individual state. With this representation two states that have exactly the same successors have the same BDD representation and can be merged. This algorithm has been improved in a couple of ways:

- It uses the BDD representation to improve the determinism of the resulting automaton. Briefly, this comes from the fact that a formula $a \lor (b \land \mathsf{X}\varphi)$ where the choice between a and b is not exclusive (both could hold) can also be interpreted as the equivalent $a \lor (\bar{a} \land b \land \mathsf{X}\varphi)$ where the choice between a and $\bar{a}b$ is exclusive.
- When it is syntactically obvious that the property translated is persistent (Fig. 2.4), it forces the output automaton to be weak, as suggested by Černá and Pelánek [2003].

Our IJCCBS paper[7] describes a few other minor improvements (faster translation for subformulas of G, early simplification of the number of acceptance sets) that are harder to present without going into details.

```
import spot
f = spot.formula('GFa->GFb')
aut = spot.ltl_to_tgba_fm(f,
  spot._bdd_dict)
print(aut.to_str('HOA'))
```

Example 2.9: The core LTL translation algorithm cannot be easily accessed from the command-line: the `ltl2tgba` tool will always perform some amount of formula and automaton simplification, even in its lowest settings. Experiments with this low-level algorithm can be done in C++ or Python (above) by calling the `ltl_to_tgba_fm()` function.

[7] A. Duret-Lutz. LTL translation improvements in Spot 1.0. Int. J. on Critical Computer-Based Systems, 5 (1/2):31–54, 2014

2.7 *Simplifying TGBAs*

The TGBAs resulting from the above translation can often be simplified. We have implemented three types of simplifications. The first three of these steps are described in detail in our Spin'13 paper.[8]

1. SCC-based simplifications use the fact that accepting runs of Büchi automata will ultimately end in one (accepting) SCC of the automaton. States that cannot reach an accepting SCC are useless and can be removed. SCCs that are not accepting do not need acceptance marks.
2. Acceptance mark simplifications: if the set of transitions with ❶ includes the set of transitions with ⓿, then the mark ❶ is superfluous and can be removed from the transition structure as well as from the acceptance condition. This inclusion check can be refined on an SCC-per-SCC basis.
3. Simulation-based relations can be used to merge states and remove transitions. Spot implements direct and reverse simulations; and those work with arbitrary acceptance conditions. The implementation of this reduction uses a BDD-based representation of a "signature" of each state, that (as in the case of the main translation), automatically prunes redundant transitions, and improves the determinism of the resulting automaton.
4. Weak and deterministic Büchi automata (WDBA) can be minimized efficiently by an algorithm due to Löding [2001]. Even if the result of the core translation does not give a WDBA, any obligation property (cf. Fig. 2.4) can be represented by a WDBA and Dax et al. [2007] show how to obtain it, if it exists.

[8] T. Babiak, T. Badie, A. Duret-Lutz, M. Křetínský, and J. Strejček. Compositional approach to suspension and other improvements to LTL translation. In SPIN'13, vol. 7976 of LNCS, pp. 81–98. Springer, 2013a

All these optimizations can be applied to any automaton using the `autfilt` tool (Ex. 2.10).

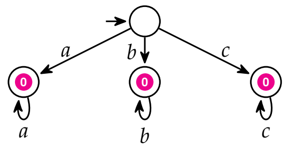
The use of WDBA-minimization is probably the main reason for the success of `ltl2tgba` in Table 2.7. In many cases, the minimal WDBA obtained is smaller than the non-deterministic automaton

```
$ spin -f 'X<>a' |
> autfilt --stats=%s
4
$ spin -f 'X<>a' | autfilt \
> -B --small --stats=%s
3
```

Example 2.10: Spin translates $\mathsf{XF}a$ into a 4-state BA. Simplifying that automaton with `autfilt` produces a 3-state BA.

produced by the core translation algorithm. The other cases raise an interesting question (cf. Fig. 2.11): should a small non-deterministic automaton be preferred to a larger deterministic one? `ltl2tgba` and `autfilt` leave that decision to the user: the `-D` option favors deterministic automata even when larger.

```
$ ltl2tgba 'Ga | Gb | Gc'
```



```
$ ltl2tgba -D 'Ga | Gb | Gc'
```

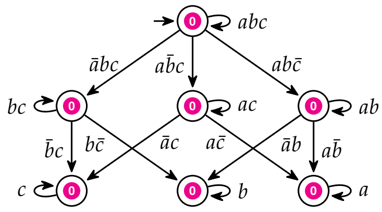


Figure 2.11: Top: non-deterministic BA for Ga ∨ Gb ∨ Gc. Bottom: minimal equivalent WDBA.

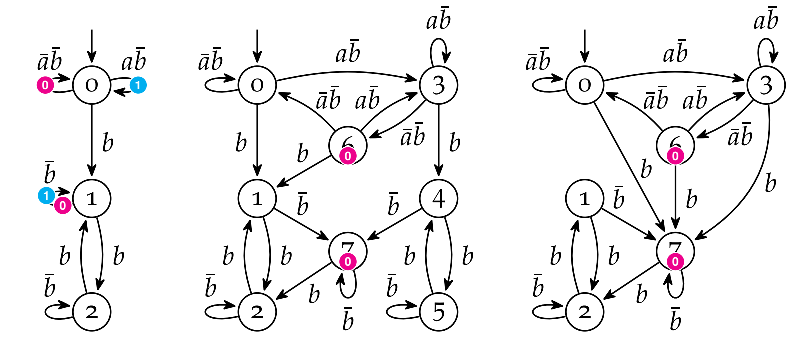
2.8 *From TGBA to BA*

Because several model checking tools only work with (state-based) Büchi automata, Spot has a rather evolved degeneralization routine.

Transforming the TGBA of Figure 2.1(d) (p. 9) into the BA of Figure 2.1(a) can be done with a classical procedure that duplicates the original TGBA $n + 1$ times where n is the number of acceptance marks. Improvements to this text-book procedure were discussed by Gastin and Oddoux [2001], and this can be used to build the BA of Figure 2.12(b).

At SPIN'13 we presented two additional optimizations that use information about SCCs in the TGBA to avoid creating some useless states in the resulting BA.[9] This optimized degeneralization procedure builds the BA of Figure 2.12(c).

[9] T. Babiak, T. Badie, A. Duret-Lutz, M. Křetínský, and J. Strejček. Compositional approach to suspension and other improvements to LTL translation. In SPIN'13, vol. 7976 of LNCS, pp. 81–98. Springer, 2013a



(a) `input` TGBA (b) Equivalent BA obtained with classical degeneralization (c) Equivalent BA produced by SCC-aware degeneralization

Figure 2.12: An `input` TGBA, and two degeneralizations into BA. Assuming automaton (a) is `input`, the Büchi automaton (b) can be built by disabling some optimizations with

```
$ autfilt -B input \
-x'!degen-reset,!degen-lcache'
```

while automaton (c) is built by default with

```
$ autfilt -B input
```

2.9 *Optimizing BA for Spin*

The previous result was a collaboration with the authors of `ltl3ba`, and we continued this collaboration by exploring how the shape of the BA could influence the performance of the `spin` model checker. This line of work produced two articles, at Spin'14 and Spin'15.

In the first one[10], we compared several automata produced by off-the-shelf translators, as done for example in Fig. 2.13, and observed that using them with Spin to model-check an actual model from the BEEM benchmark [Pelánek, 2007] gives some counterintuitive results. As can be seen in Table 2.14 the size of the property automaton does not correlate with the run time of Spin. Automata $\mathcal{D}_2$ and $\mathcal{D}_4$ have the same size (for all metrics used in this table), but show a two-fold difference in model-checking performance. Automaton $\mathcal{D}_7$ is slightly worse than $\mathcal{D}_6$ despite having fewer edges.

[10] F. Blahoudek, A. Duret-Lutz, M. Křetínský, and J. Strejček. Is there a best Büchi automaton for explicit model checking? In SPIN'14, pp. 68–76. ACM, 2014

Considering the problem from the point of view of the emptiness

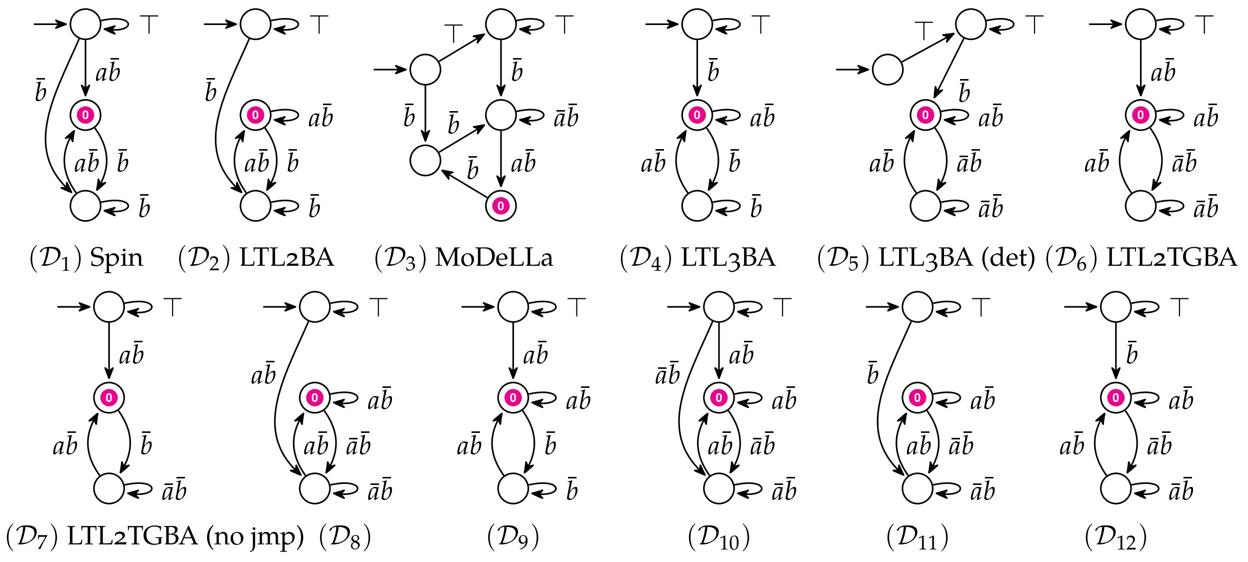


Figure 2.13: BAs for the formula $\neg(\mathsf{GF}a \to \mathsf{GF}b)$, as produced by off-the-shelf translators, or by hand.

	automaton size				statistics from Spin's execution		
	states	ndst	edges	trans	stored states	visited trans	time
$\mathcal{D}_1$ Spin	3	2	6	12	1577846	7680k	6.04s
$\mathcal{D}_2$ LTL2BA	3	3	6	12	1577440	7684k	5.95s
$\mathcal{D}_3$ MoDeLLa	5	2	8	18	1580893	7670k	6.13s
$\mathcal{D}_4$ LTL3BA	3	3	6	12	2299250	15583k	12.10s
$\mathcal{D}_5$ LTL3BA (det)	4	1	7	14	2297625	15561k	12.00s
$\mathcal{D}_6$ LTL2TGBA	3	1	6	9	848641	2853k	2.26s
$\mathcal{D}_7$ LTL2TGBA (no jmp)	3	1	5	9	852094	2863k	2.34s
$\mathcal{D}_8$	3	1	6	9	848641	2853k	2.43s
$\mathcal{D}_9$	3	3	6	11	852094	2878k	2.43s
$\mathcal{D}_{10}$	3	1	7	10	1575844	7658k	7.38s
$\mathcal{D}_{11}$	3	1	6	10	1577440	7657k	7.07s
$\mathcal{D}_{12}$	3	1	6	10	2297625	15561k	12.30s

Table 2.14: Statistics about generated automata in Fig. 2.13 and the corresponding run of Spin on the empty product with model `peterson.4.pm`.

check algorithm used by Spin, which is based on two nested depth-first searches [Holzmann et al., 1996], we realized that the location of accepting states in the automaton is important.

We concluded that when the product $K_M \otimes A_{\neg\varphi}$ is expected to be empty (φ should hold), the best automaton $A_{\neg\varphi}$ for Spin should have accepting states that are hard to reach from the initial state as it will lessen the chance to trigger a nested DFS. On the contrary, if the product $K_M \otimes A_{\neg\varphi}$ contains an accepting cycle, Spin can find it faster if the accepting states of $A_{\neg\varphi}$ are easy to reach from the initial state and the accepting cycles are short. It turns out that the degeneralization procedure discussed in the previous section can be adjusted for both cases.

In a second paper[11], we remarked that if an automaton $A_{\neg\varphi}$ is going to be used to verify a model M by constructing the product

[11] F. Blahoudek, A. Duret-Lutz, V. Rujbr, and J. Strejček. On refinement of Büchi automata for explicit model checking. In SPIN'15, vol. 9232 of LNCS, pp. 66–83. Springer, 2015

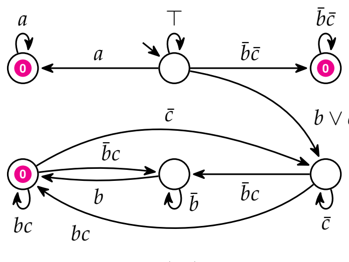
$K_M \otimes A_{\neg\varphi}$, then $A_{\neg\varphi}$ can be simplified by taking into account some knowledge of M. Typically we can use the fact that some of the atomic propositions used in φ are mutually exclusive in M: for instance $x = 5$ and $x > 10$ cannot both hold at once, and a process cannot be in two different locations at the same time.

Using such a priori knowledge about the model, we can simplify the automaton, as illustrated by Fig. 2.15. The simplifications performed are two-fold.

First, the automaton can be refined by removing or merging useless transitions: for instance under the assumption that two propositions among $\{a, b, c\}$ cannot both hold at once, the behaviors captured by the top-left state of automaton B_1 are included in those captured by its top-right state, allowing the former state to be removed. This type of simplification entailed an average 28% reduction of the number of transitions visited by Spin on the empty products of our benchmark.

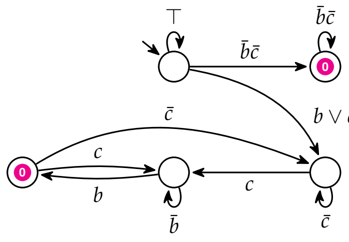
A second simplification is to shorten the Boolean formulas which label the transitions in order to limit the number of evaluations of atomic propositions performed by the model checker: for instance under the same assumptions, $\bar{b}c$ can be rewritten c, testing a single atomic proposition instead of two. This type of subtle change improved the run time of Spin by 3.5%.

```
$ f='F(Ga | (GFb <-> GFc))'
$ ltl2tgba -B "$f"
```



(B_1)

```
$ f='F(Ga | (GFb <-> GFc))'
$ ltl2tgba -B "$f" |
> autfilt -B --small \
> --exclusive-ap=a,b,c \
> --symplify-exclusive-ap
```



(B_2)

Figure 2.15: Automata for $\varphi = \mathsf{F}(\mathsf{G}a \vee (\mathsf{GF}b \leftrightarrow \mathsf{GF}c))$. Automaton B_2 is a simplified version of B_1 assuming that a, b and c are mutually exclusive.

2.10 *Building unambiguous automata*

An automaton is unambiguous if any word can be recognized by at most one accepting run. This can be understood as a weaker notion than determinism: in a deterministic automaton any word should be recognized by at most one run of the automaton (that could be a rejecting run). Unambiguous Büchi automata are as expressive as Büchi automata [Arnold, 1984, Theorem IV-1], and can be exponentially more succinct than deterministic BA [Bousquet and Löding, 2010, Remark 2].

Probabilistic model checking of an LTL property φ against a Markov chain $\mathcal{M}$ is classically done using an ω-automaton A_φ that is deterministic in order to guarantee that $\mathcal{M} \otimes A_\varphi$ is a Markov chain [Baier and Katoen, 2008, Section 10.3]. However, unambiguous automata are an attractive alternative, since the probabilities really only have to be computed along accepting runs [Baier et al., 2016b].

As seen in Table 2.7, Spot is already good at producing small automata that are often deterministic (hence unambiguous). However, it also implements a variant of its translation algorithm that always outputs unambiguous TGBAs. The technique extends an idea described by Benedikt et al. [2013] to all operators supported by Spot.

This idea can be quickly explained as follows. When translating the LTL formula $\alpha \mathsf{U} \beta$ as an automaton, this formula is normally interpreted as the equivalent formula $\beta \vee (\alpha \wedge \mathsf{X}(\alpha \mathsf{U} \beta))$, meaning that a state recognizing the language $\mathscr{L}(\alpha \mathsf{U} \beta)$ will usually have two outgoing transitions for $\mathscr{L}(\beta)$ and $\mathscr{L}(\alpha \wedge \mathsf{X}(\alpha \mathsf{U} \beta))$. To obtain an unambiguous automaton, it suffices to make these two later languages disjoint. So in practice $\alpha \mathsf{U} \beta$ would now be seen as $\beta \vee (\alpha \wedge \neg\beta \wedge \mathsf{X}(\alpha \mathsf{U} \beta))$, therefore representing two outgoing transitions for the languages $\mathscr{L}(\beta)$ and $\mathscr{L}(\alpha \wedge \neg\beta \wedge \mathsf{X}(\alpha \mathsf{U} \beta))$. This idea has to be generalized to all operators that involve a non-deterministic choice.

Figure 2.16 shows an ambiguous BA resulting from the translation of $\mathsf{GF}a \rightarrow \mathsf{GF}b \equiv \mathsf{FG}\neg a \vee \mathsf{GF}b$. One can readily see that the sub-automaton consisting of the states $\{0, 1\}$ recognizes $\mathscr{L}(\mathsf{GF}b)$ while the sub-automaton $\{0, 2\}$ recognizes $\mathscr{L}(\mathsf{FG}\neg a)$. Those two languages both contain the word $\bar{a}b; \bar{a}b; \bar{a}b; \ldots$, and the $\top$-labeled self-loop on state 0 makes it possible to build an infinite number of accepting runs over this word in any of these two sub-automata.

In the unmabiguous BA of Figure 2.17, the sub-automaton $\{0, 1\}$ still recognizes $\mathscr{L}(\mathsf{GF}b)$, but the sub-automaton $\{0, 2, 3, 4\}$ now recognizes $\mathscr{L}(\mathsf{FG}\neg a \wedge \neg\mathsf{GF}b)$ which is disjoint. The word $\bar{a}b; \bar{a}b; \bar{a}b; \ldots$ is now accepted by a unique run that goes through states 0 and 1.

Baier et al. [2016b] used unambiguous Büchi automata generated by Spot in their experiments[12], showing that they were a viable alternative to probabilistic model checking using deterministic Rabin automata. To our knowledge, the only other existing tool that can build unambiguous automata from LTL is Tulip [Lenhardt, 2013] however its website is not responsive anymore, and it was not clear if using its built-in LTL translator as a standalone tool was possible. Therefore we have never performed any tool comparison between unambiguous automata produced by Spot and those of any other tool.

```
$ ltl2tgba 'GFa -> GFb'
```

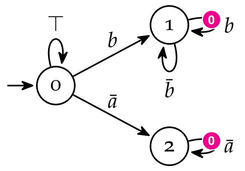


Figure 2.16: An ambiguous BA for $\mathsf{GF}a \rightarrow \mathsf{GF}b$. Note that the word $\bar{a}b; \bar{a}b; \bar{a}b; \ldots$ has an infinite number of accepting runs.

```
$ ltl2tgba -U 'GFa -> GFb'
```

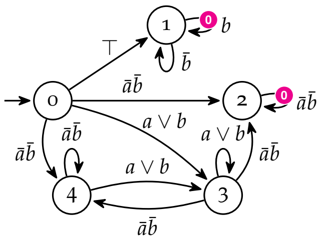


Figure 2.17: An unambiguous BA for $\mathsf{GF}a \rightarrow \mathsf{GF}b$. Here the word $\bar{a}b; \bar{a}b; \bar{a}b; \ldots$ has a unique accepting run.

[12] Additionally, Baier et al. [2016a, Appendix F.4] shows some experimental data comparing the size of the (non-deterministic) Büchi automata and unmabiguous Büchi automata produced by Spot.

2.11 *Building monitors*

Let us define monitors as a subclass of generalized Büchi automata where no acceptance sets are used, i.e., all runs are accepting. Since the only way to reject words is to use automata that are incomplete, monitors can be used to detect "bad prefixes": the expressive power of monitors corresponds to the Safety class of Figure 2.4 (p. 10).

In the context of online runtime verification, a monitor for a correctness property φ can be run alongside a live system to monitor its events, and signal any execution whose current (finite) prefix cannot be extended into an infinite word that is correct for φ.

D'Amorim and Roşu [2005] and Tabakov and Vardi [2010] explore this usage, and describe algorithms to construct monitors (and deterministic monitors) that reject the bad prefixes of any LTL formula φ. Once a TGBA[13] for φ has been obtained, it suffices to remove all useless SCCs, then remove all acceptance sets, and then optionally determinize and minimize the result. Even if φ is not a safety property this procedure will build a monitor recognizing a superset of $\mathscr{L}(\varphi)$, effectively capturing the monitorable parts of φ.

These transformations are implemented in Spot; `ltl2tgba` can generate monitors with option `-M`, and deterministic monitors with options `-M -D`, as illustrated by Figures 2.18 and 2.19.

```
$ ltl2tgba -M '(Xa&Fb)|Gc'
```

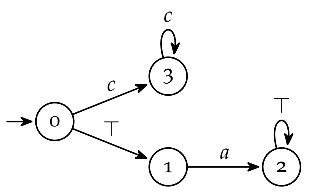


Figure 2.18: A non-deterministic monitor generated for $(\mathsf{X}a \wedge \mathsf{F}b) \vee \mathsf{G}c$. Note that $\mathsf{F}b$ is not a monitorable property, so this automaton accepts $\mathscr{L}(\mathsf{X}a \vee \mathsf{G}c)$, which is a superset of the original language.

```
$ ltl2tgba -M -D '(Xa&Fb)|Gc'
```

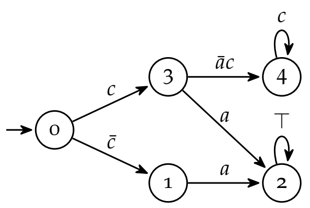


Figure 2.19: A minimal deterministic monitor generated for $(\mathsf{X}a \wedge \mathsf{F}b) \vee \mathsf{G}c$.

[13] Tabakov and Vardi [2010] start from a BA instead, but since the BA is obtained by degeneralizing a TGBA in our case, it is more efficient to start from that TGBA.

3
Emptiness checks

This chapter was the subject of E. Renault's Ph.D thesis [2014] which I co-supervised with F. Kordon and D. Poitrenaud.

3.1 *Introduction*

If an ω-automaton accepts a word, it necessarily contains an accepting cycle (i.e., a cycle visiting all acceptance marks in the case of generalized Büchi) that is reachable from the initial state. Emptiness check algorithms search for the existence of such a cycle.

Automata with generalized Büchi acceptance can be tested for emptiness by a slight modification of SCC-enumeration algorithms [Couvreur, 1999, Geldenhuys and Valmari, 2005]. Büchi automata can be checked using the same algorithms, but their simpler acceptance can also be verified using two nested depth-first search (NDFS) [Holzmann et al., 1996] with a smaller memory footprint. When the property to verify is more concisely expressed using generalized Büchi acceptance, it is preferable to use SCC-based algorithms over degeneralizing the automaton to use an NDFS.[1]

[1] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In SPIN'05, vol. 3639 of LNCS, pp. 143–158. Springer, 2005

3.2 *Strength-based decompositions*

Some subclasses of automata enable more efficient emptiness checks. In particular, weak automata (already discussed page 10, along with Fig. 2.4) can be tested for emptiness using a simple DFS [Černá and Pelánek, 2003].

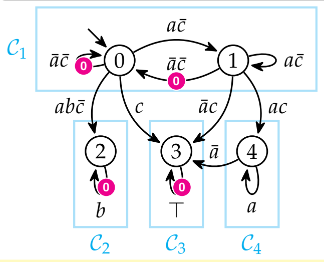
Terminal automata form a subclass of weak automata in which accepting SCCs are complete. When the model M has no deadlock (each state has at least one successor) and the property automaton $A_{\neg\varphi}$ is known to be terminal, the emptiness check of the product becomes a simple reachability problem: detecting cycles is not even necessary.

Based on these notions, we proposed[2] to split an arbitrary automaton $A_{\neg\varphi}$ into possibly three automata A_S, A_W, and A_T of different strengths: A_T contains the terminal behaviors, A_W the weak behaviors that are not terminal, and A_S the rest (the strong part).

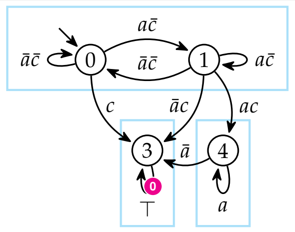
[2] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Strength-based decomposition of the property Büchi automaton for faster model checking. In TACAS'13, vol. 7795 of LNCS, pp. 580–593. Springer, 2013b

This is illustrated by Figure 3.1: the initial automaton, at the top, contains four SCCs. $\mathcal{C}_1$ is strong, because it mixes accepting and

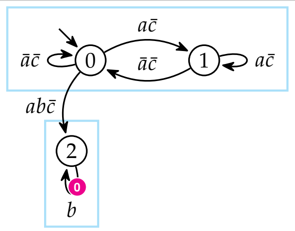
```
$ ltl2tgba '(Ga->Gb)W c' >aut
```



```
$ autfilt --decompose=t aut
```



```
$ autfilt --decompose=w aut
```



```
$ autfilt --decompose=s aut
```

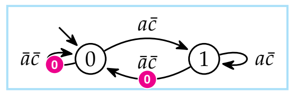


Figure 3.1: Decomposition of $(\mathsf{G}a \to \mathsf{G}b)\,\mathsf{W}\,c$ into three automata: a terminal, a weak, and a strong one.

rejecting cycles. $\mathcal{C}_2$ is weak, because all its transitions have the same marks. $\mathcal{C}_3$ is terminal, because it is weak and accepts all words. $\mathcal{C}_4$ is rejecting. For any automaton using generalized Büchi acceptance, extracting a subautomaton of a given force can be done by taking all SCCs of that force unchanged, as well as their parents stripped from acceptance marks. For instance, the terminal automaton of Figure 3.1 has a verbatim copy of the terminal component $\mathcal{C}_3$, plus "naked" versions of its parents $\mathcal{C}_1$ and $\mathcal{C}_4$.

This decomposition naturally ensures that $\mathscr{L}(A_{\neg\varphi}) = \mathscr{L}(A_T) \cup \mathscr{L}(A_W) \cup \mathscr{L}(A_S)$. In particular, instead of checking $\mathscr{L}(K_M \otimes A_{\neg\varphi}) = \emptyset$, a model checker could check $\mathscr{L}(K_M \otimes A_S) = \emptyset \wedge \mathscr{L}(K_M \otimes A_W) = \emptyset \wedge \mathscr{L}(K_M \otimes A_T) = \emptyset$ instead. This has several advantages:

- In the case of $K_M \otimes A_W$, and $K_M \otimes A_T$, emptiness checks specialized for weak or terminal automata can be used.
- The three emptiness checks can be parallelized.
- Each one of A_S, A_W, and A_T is simpler than the original automaton, so it may be easier to simplify than the original (Fig. 3.2). This is particularly true for the weak (and terminal) automata, where the WDBA-minimization discussed on page 13 can be applied.

```
$ autfilt --decompose=s \
> --small aut
```

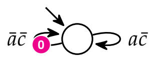


Figure 3.2: Simplified automaton for the strong part of $(\mathsf{G}a \to \mathsf{G}b)\,\mathsf{W}\,c$. Compare with the last automaton of Fig. 3.1.

It should be noted that the languages of these three automata are not necessarily disjoint: some words could be accepted by more than one automaton. Nonetheless each product $K_M \otimes A_S$, $K_M \otimes A_W$, and $K_M \otimes A_T$ is at most as big as the original $K_M \otimes A_{\neg\varphi}$, so testing the emptiness of the former three products by running three model-checkers in parallel can only be faster than checking $K_M \otimes A_{\neg\varphi}$ using one model checker. In our benchmark, we observed speedup factors between 2 and 4 when the products are empty (i.e., the emptiness checks have to construct the entire product). Speedups greater than 3 come from the fact that decomposed automata can be simplified further.

This decomposition technique could be generalized to produce one automaton for each accepting SCC of the original automaton.

3.3 *Using Union-find for Emptiness Checks*

SCC-based emptiness checks use SCC enumeration algorithms such as the algorithm of Tarjan [1971, 1972] or the lesser known algorithm of Dijkstra [1973, 1976]. For instance, Couvreur [1999] implements Dijkstra's algorithm (despite the paper presenting it as a variant of Tarjan) while Geldenhuys and Valmari [2004] uses Tarjan.

Another SCC enumeration algorithm was suggested by Gabow [2000, p. 109][3] and uses the union-find data structure [Tarjan, 1975]. In this algorithm, a DFS of a graph is performed, and the union-find data structure is used to build a partition of the vertices of the graph into SCCs: initially each node is considered to be in its own class of the partition, but whenever a cycle is detected, all states on that cycle are merged into the same class.

In a paper presented at LPAR'13[4], we described a Union-Find-based emptiness check extending Gabow's algorithm, and suggested

[3] The main algorithm of Gabow's paper is a reinvention of Dijkstra's algorithm. What we call Gabow's algorithm here is the idea evoked on page 109 of that paper.

[4] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In LPAR'13, vol. 8312 of LNCS, pp. 668–682. Springer, 2013a

some optimizations for Tarjan and Dijkstra-based algorithms.

This was presented in a sequential context, where the three algorithm give roughly equivalent run times; however our ultimate goal was to investigate these algorithms in a parallel context.

3.4 *Parallelisation of SCC-based emptiness checks*

Several parallel emptiness checks have been proposed over the last years [Brim et al., 2001, 2004, Černá and Pelánek, 2003, Barnat et al., 2003, 2005, 2009]. These algorithms are mainly BFS-like (or at least "non-DFS") explorations of the state space. Multi-core adaptations of these algorithms with lock-free data structure have been discussed, but not evaluated [Barnat et al., 2010a].

Recent publications show running multiple NDFS-based algorithms with the swarming technique [Holzmann et al., 2011] scale better in practice [Evangelista et al., 2011, Laarman et al., 2011, Laarman and van de Pol, 2011, Evangelista et al., 2012]. In swarming, each thread executes the same algorithm, but uses a different exploration order.

We investigated what could be done with SCC-based emptiness checks in a parallel setting.[5] Although Gabow's algorithm can be seen as an implementation of Dijkstra's algorithm using a union-find, we realized that in a parallel context the union-find could serve as a global data structure in which all threads can contribute knowledge about the partition of the states into different SCCs. This shared data structure could be filled by threads running either some variant of Dijkstra or of Tarjan; possibly mixing the two algorithms.

In a parallel setting with N threads, it makes sense to combine this with the decomposition technique described Section 3.2. We tried that[6] using two different strategies:

S1: if the property automaton can be decomposed into $P \in \{1,2,3\}$ automata, then each product will be checked using $\frac{N}{P}$ threads. In this context the decomposition improves the spread of the swarming technique by ensuring that groups of threads focus on behaviors of different strengths.

S2: use all N threads for the product $K_M \otimes A_T$, then if no counterexample is found, use N threads for $K_M \otimes A_W$, and finally $K_M \otimes A_S$. In this strategy, the automata are ordered by strengths so that more complex emptiness checks are avoided when possible.

Figure 3.3 shows the effect of these two strategies compared to our base parallelization of Dijkstra, in a benchmark derived from BEEM [Pelánek, 2007]. Strategy S1 is not used with 1 or 2 threads, as the decomposition could produce 3 automata. Nonetheless, S1 appears good in both empty products (the full product had to be explored to prove the formula satisfied) and non-empty product (one thread found a counterexample and other threads could be stopped immediately).

[5] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Parallel explicit model checking for generalized Büchi automata. In TACAS'15, vol. 9035 of LNCS, pp. 613–627. Springer, 2015

[6] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Variations on parallel explicit model checking for generalized Büchi automata. Int. J. on Software Tools for Technology Transfer (STTT), 2016. DOI 10.1007/s10009-016-0422-5

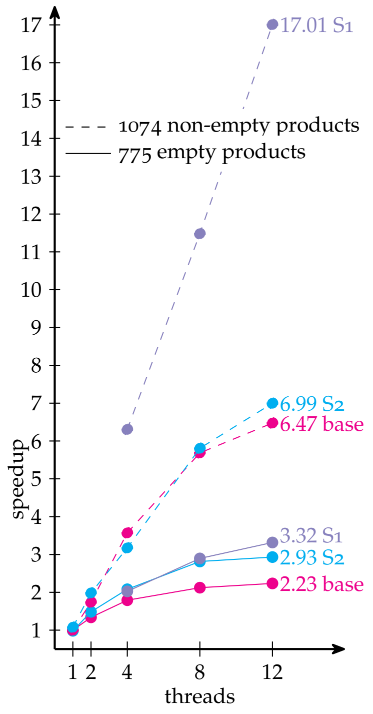


Figure 3.3: Speedups of running Dijkstra with 1, 2, 4, 8, and 12 threads (base). Without decomposition, or with decomposition using one of the two described strategies. All speedups are relative to Dijkstra with 1 thread, and averaged over 775 empty products (plain lines) or 1074 non-empty product (dashed lines).

4
Generic acceptance

So far, we have discussed automata with generalized Büchi acceptance, or subclasses of this acceptance (like plain Büchi). In this chapter we enlarge the notion of acceptance condition to also include acceptance conditions such as co-Büchi, Rabin, Streett, or any Boolean combination of those.

4.1 *The HOA format*

We shall specify the acceptance condition of an automaton as a formula, using a syntax introduced in the Hanoi Omega Automaton (HOA) format[1]. In this format, the acceptance condition is a formula over terms such as $\mathsf{Inf}(x)$ or $\mathsf{Fin}(x)$ indicating respectively that for a run to be accepting the acceptance mark x should be visited infinitely often, or finitely often.[2]

Table 4.1 shows some examples.

[1] T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Křetínský, D. Müller, D. Parker, and J. Strejček. The Hanoi Omega-Automata format. In CAV'15, vol. 9206 of LNCS, pp. 479–486. Springer, 2015. See also `http://adl.github.io/hoaf/`

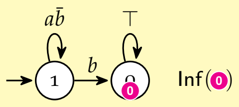
[2] A similar formalism was already used long ago by Emerson and Lei [1987]: they write $\overset{\infty}{\mathsf{F}}(p_0) \wedge \overset{\infty}{\mathsf{G}}(p_1)$ when we write $\mathsf{Inf}(⓿) \wedge \mathsf{Fin}(\neg ❶)$. Safra and Vardi [1989] named automata with similar acceptance conditions "EL automata".

Büchi:	$\mathsf{Inf}(0)$
generalized Büchi 3:	$\mathsf{Inf}(0) \wedge \mathsf{Inf}(1) \wedge \mathsf{Inf}(2)$
co-Büchi:	$\mathsf{Fin}(0)$
generalized co-Büchi 3:	$\mathsf{Fin}(0) \vee \mathsf{Fin}(1) \vee \mathsf{Fin}(2)$
Rabin (2 pairs):	$(\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)) \vee (\mathsf{Fin}(2) \wedge \mathsf{Inf}(3))$
Streett (2 pairs):	$(\mathsf{Fin}(0) \vee \mathsf{Inf}(1)) \wedge (\mathsf{Fin}(2) \vee \mathsf{Inf}(3))$
parity min even 5:	$\mathsf{Inf}(0) \vee (\mathsf{Fin}(1) \wedge (\mathsf{Inf}(2) \vee (\mathsf{Fin}(3) \wedge \mathsf{Inf}(4))))$

Table 4.1: Example of traditional acceptance conditions, specified using the HOA syntax.

The HOA format was created jointly with other tools authors, as a way to facilitate the interactions between our tools. Existing formats, such as the DSTAR format of `ltl2dstar` [Klein and Baier, 2006] or the XML-based format of Goal [Tsai et al., 2013] support different acceptance conditions that can only be selected from a predefined list of names. One particular motivation for HOA was that tools like `ltl3dra` [Babiak et al., 2013b] and `Rabinizer` [Křetínský and Esparza, 2012] had introduced a new acceptance condition that they called generalized Rabin and were each using a custom output format for this new type of automata. The HOA format was designed not to be restricted to a list of known acceptance conditions: specifying the acceptance condition using a formula rather than by name, so that the semantics are always known.

```
$ ltl2tgba 'a U b'
HOA: v1
name: "a U b"
States: 2
Start: 1
AP: 2 "a" "b"
acc-name: Buchi
Acceptance: 1 Inf(0)
properties: trans-labels
  explicit-labels state-acc
  deterministic
  stutter-invariant terminal
--BODY--
State: 0 {0}
[t] 0
State: 1
[1] 0
[0&!1] 1
--END--
```



Example 4.2: A Büchi automaton generated for $a \mathbin{\mathsf{U}} b$ in the HOA format, and its graphical representation.

4.2 *Why use generic acceptance?*

First, it should be pointed out that using more complex acceptance conditions does not necessarily imply slower procedures. In the context of model checking, we have already discussed the fact that using generalized-Büchi instead of Büchi could be faster.[3] Similarly, it has been shown that using the generalized Rabin condition can speed up probabilistic model checking by orders of magnitude [Chatterjee et al., 2013, Komárková and Křetínský, 2014].

[3] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In SPIN'05, vol. 3639 of LNCS, pp. 143–158. Springer, 2005

Second, not being tied to a particular acceptance condition makes some ω-automata operations easier. For instance, the product or union of two deterministic (and complete) automata is trivial to define without the constraint of producing the same acceptance (as we will discuss below).

Finally, providing algorithms that support generic acceptance opens up a wide research area, the results of which can also benefit to people working on specific subclasses of acceptance.

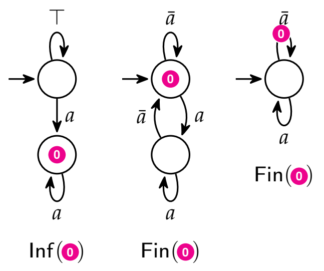


Figure 4.3: Three minimal automata for $\mathsf{F}\mathsf{G}a$, using: state-based Büchi acceptance, state-based co-Büchi acceptance, and transition-based co-Büchi acceptance.

4.3 *Transition-based acceptance revisited*

In section 2.4 we discussed the fact that transition-based acceptance was not useful to weak Büchi automata, and therefore to all persistence properties. That is true as long as it is clear we are talking about automata with Büchi acceptance (even generalized). However, persistence properties can benefit from transition-based acceptance if we consider acceptance conditions involving $\mathsf{Fin}(x)$. The typical example is the property $\mathsf{FG}a$ that can be reduced to a 1-state (deterministic) transition-based co-Büchi automaton (Fig. 4.3).

4.4 *Generic algorithms*

Figure 4.4 shows two deterministic and complete Streett automata $\mathcal{A}$ and $\mathcal{B}$, along with their synchronous product. If we name $\mathcal{C}_1$ the product equipped with the acceptance condition $(\mathsf{Fin}(0) \vee \mathsf{Inf}(1)) \wedge (\mathsf{Fin}(2) \vee \mathsf{Inf}(3))$, then $\mathcal{C}_1$ is a Streett automaton verifying $\mathscr{L}(\mathcal{C}_1) = \mathscr{L}(\mathcal{A}) \cap \mathscr{L}(\mathcal{B})$. However, if we call $\mathcal{C}_2$ the product equipped with the acceptance condition $\mathsf{Fin}(0) \vee \mathsf{Inf}(1) \vee \mathsf{Fin}(2) \vee \mathsf{Inf}(3)$, then $\mathcal{C}_2$ verifies $\mathscr{L}(\mathcal{C}_2) = \mathscr{L}(\mathcal{A}) \cup \mathscr{L}(\mathcal{B})$. Note that the acceptance condition of $\mathcal{C}_2$ is not Streett, and does not correspond to any "traditional" acceptance condition.

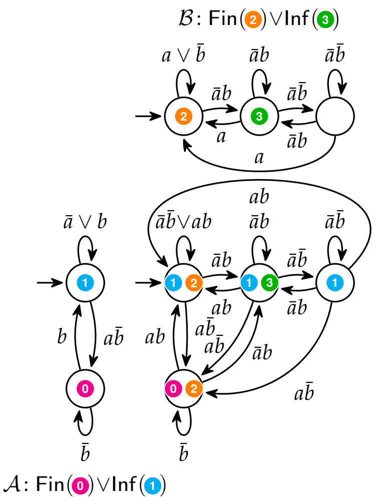


Figure 4.4: Depending on how we define the acceptance condition of the synchronous product of $\mathcal{A}$ and $\mathcal{B}$ we can intersect or merge their languages.

This construction for the intersection or union of two ω-automata works regardless of the acceptance condition, and also on non-deterministic automata. The union only requires complete automata.

```
$ S="ltl2dstar \
> --automata=streett"
$ ltldo "$S" -f 'G(a->Fb)' >A
$ ltldo "$S" -f 'GFb->GFa' >B
$ autfilt --product-and A B>C1
$ autfilt --product-or  A B>C2
```

Example 4.5: How to construct the two input automata of Fig. 4.4, using `ltl2dstar`, and build the two flavors of the product.

Complementing a deterministic and complete automaton is as simple as complementing its acceptance condition and applying the rules $\neg\mathsf{Fin}(x) \equiv \mathsf{Inf}(x)$ and $\neg\mathsf{Inf}(x) \equiv \mathsf{Fin}(x)$.

Completing an ω-automaton is trickier than it sounds: after a sink state is added, it should be given a set of acceptance marks that does not satisfy the acceptance condition. An automaton can therefore be completed without changing its acceptance condition unless that acceptance is a tautology. If the acceptance is a tautology, like $\mathsf{Fin}(0) \lor \mathsf{Inf}(0)$, then the input automaton contains only accepting cycles, and the output automaton can be given the Büchi condition in order to mark all states except the sink as accepting.

Many of the simplification algorithms that we discussed in Chapter 2 (such as simulation-based reductions, WDBA minimization, SCC-based simplifications) can be easily adapted to generic acceptance: in Spot they are therefore implemented for automata with arbitrary acceptance (Example 4.6).

For algorithms that are harder to generalize (e.g., determinization, emptiness checks), it is always possible to build an equivalent automaton with a more constrained acceptance condition, as discussed in the next section.

```
$ echo G a | ltl2dstar - - |
> autfilt -D -C -S \
> --stats='in %S, out %s'
in 3, out 2
```

Example 4.6: `ltl2dstar` 0.5.3 translates G*a* into a 3-state complete deterministic state-based Rabin automaton. Passing this automaton to `autfilt` for simplification with instructions to output a deterministic (`-D`) and complete (`-C`) automaton with state-based acceptance (`-S`), yields a 2-state automaton. In this case, `autfilt` applied the algorithm of Dax et al. [2007] (generalized to work on Rabin automata) to detect that the input automaton represents an obligation property and can therefore be minimized into a 1-state weak deterministic Büchi automaton. This automaton has 2 states once completed.

4.5 Fin-*less acceptance*

The SCC-based emptiness-check discussed in Chapter 3 were presented in the context of automata using generalized Büchi acceptance. However, these emptiness checks will also work with automata whose acceptance condition is a disjunction of generalized Büchi acceptance conditions. We call this subclass of acceptance conditions "Fin-less acceptance", because is it simply a positive Boolean combination of $\mathsf{Inf}(x)$ terms.

A simple transformation can be used to transform an automaton with arbitrary acceptance condition into an automaton with Fin-less acceptance. Let us assume, without loss of generality[4], that the input acceptance condition is under the following disjunctive normal form:

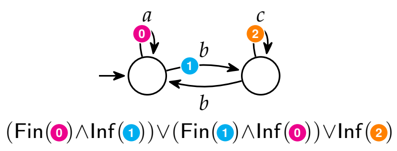
$$\bigvee_i \left(\mathsf{Fin}(x_{i1}) \land \ldots \land \mathsf{Fin}(x_{in_i}) \land \mathsf{Inf}(y_{i1}) \land \ldots \land \mathsf{Inf}(y_{im_i})\right)$$

where $n_i \geq 0$ and $m_i \geq 1$ for all i, and that the sets of Inf marks of each clause $Y_i = \{y_{i1}, \ldots, y_{im_i}\}$ are such that no set is a superset of another one: $\forall i, \nexists j, Y_i \subseteq Y_j$.

[4] If the input acceptance does not satisfy these assumptions, it can be easily modified by adding some $\mathsf{Inf}(z)$ terms and adding the mark z everywhere.

Under these assumptions, the input automaton can be transformed into another automaton with $\bigvee_i \left(\mathsf{Inf}(y_{i1}) \land \ldots \land \mathsf{Inf}(y_{im_i})\right)$ as acceptance condition. The transformation is as follows: For each SCC C_j of the original automaton and each clause i whose Fin marks are present in C_j, create a clone D_{ij} of the SCC C_j, remove from that clone all transitions (or states) that have some marks in $X_i = \{x_{i1}, \ldots, x_{in_i}\}$, and add at least one non-deterministic jump from C_j to D_{ij} per elementary cycle of D_{ij}. Finally, in the SCCs that correspond to the original automaton, leave only the marks that belong to some X_i for which $Y_i = \emptyset$.

To avoid the complexity of enumerating the elementary cycles of D_{ij}, our implementation creates one non-deterministic jump each time a back-edge is found during a depth-first search of C_j. This potentially creates more nondeterministic jumps than needed. Fig. 4.7 gives an example of such transformation (note that the bottom right state generated by this construction is superfluous).



```
$ autfilt --remove-fin in >out
```

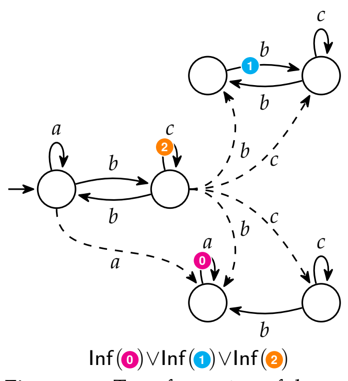


Figure 4.7: Transformation of the top automaton into an equivalent automaton (at the bottom) with Fin-less acceptance. The dashed transitions correspond to non-deterministic jumps added between the original automaton and its clones, each recognizing a different clause of the acceptance condition.

Spot implements a variant of this construction that guarantees that if the input acceptance needs to be "fixed" to satisfy the assumptions, the output acceptance uses at most one additional mark. Additionally, we have implemented some transformations that are specific to some subclasses of acceptance conditions and that are automatically used when applicable: for instance state-based Rabin automata can be converted to Büchi automata, preserving determinism whenever possible, using a variant of an algorithm by Krishnan et al. [1994].

As already mentioned, automata with Fin-less acceptance conditions can be easily checked for emptiness using readily available emptiness checks, such as those discussed in Chapter 3. Those emptiness checks are not appropriate for automata that use Fin. The emptiness check of automata with generic acceptance has been considered long ago: Emerson and Lei [1985] studied the problem of detecting fair states in a finite system under a fairness hypotheses defined as a Boolean function using operators similar to Inf and Fin. Their result translates immediately to the emptiness check problem for ω-automata: emptiness check with generalized acceptance is NP-complete, but they provide a polynomial algorithm for a large subclass of acceptance conditions: all conditions that can be encoded as a disjunction of Streett acceptance (this encompasses Rabin and generalized-Rabin).

We plan to investigate these "generic emptiness checks" in the future. In the meantime, using our Fin-removal procedure in front of more traditional emptiness checks is enough to provide a usable system. The only part of our procedure that does not have a polynomial complexity is the conversion of the acceptance condition into disjunctive normal form, which we implement using binary decision diagrams to prune redundant terms.

Finally, it should be noted that a Fin-less acceptance can be converted into generalized Büchi acceptance by first converting it into conjunctive normal form, and then replacing each clause $\mathsf{Inf}(x_{1i}) \vee \ldots \vee \mathsf{Inf}(x_{in_i})$ by a single $\mathsf{Inf}(y_i)$ where y_i is a mark that should label all transitions (or states) marked with any of $x_{1i}, \ldots, x_{1n_i}$. In the case of the automaton of Figure 4.7, the resulting acceptance is already a single-clause CNF, so it can be trivially converted into Büchi acceptance as in Figure 4.8.

```
$ autfilt --tgba in >out
```

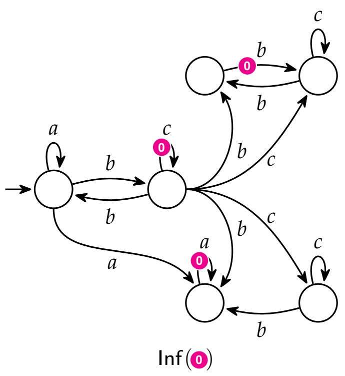


Figure 4.8: Converting the automaton of Figure 4.7 to use generalized Büchi acceptance, or in this case, Büchi acceptance.

4.6 *Example tool benefiting from generic acceptance conditions*

Spot distributes a tool called `ltlcross`[5], that started as a reimplementation of the no-longer-maintained `lbtt`: a randomized test-bench for translators of LTL formulas into (generalized) Büchi automata [Tauriainen and Heljanko, 2002]. The current version of `ltlcross` of course supports arbitrary acceptance conditions. It is regularly used to test Spot's translation algorithms, but also has been used for instance by the authors of `ltl3ba` [Babiak et al., 2012], `ltl3dra` [Babiak et al., 2013b], `ltl2dstar` [Klein and Baier, 2006, 2007], or `Rabinizer3` [Komárková and Křetínský, 2014] to test

[5] A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In ATVA'13, vol. 8172 of LNCS, pp. 442–445. Springer, 2013

```
$ genltl --eh=9 | ltlcross --verbose -D modella ltl2tgba
-:1: G((p0) & (X(F((p1) & (X(F((p2) & (X(F(p3)))))))))))
Running [P0]: modella 'lcr-i0-pMSiff' 'lcr-o0-8NvGid'
Running [P1]: ltl2tgba -H 'G((p0) & (X(F((p1) & (X(F((p2) & (X(F(p3)))))))))))'>'lcr-o1-Dsr1mb'
Running [N0]: modella 'lcr-i0-BZJdu9' 'lcr-o0-eVhqB7'
Running [N1]: ltl2tgba -H '!(G((p0) & (X(F((p1) & (X(F((p2) & (X(F(p3))))))))))))'>'lcr-o1-vDjcJ5'
info: collected automata:
info:   P0      (1 st.,0 ed.,1 sets) deterministic
info:   N0      (10 st.,18 ed.,1 sets)
info:   P1      (5 st.,19 ed.,3 sets)
info:   N1      (5 st.,9 ed.,1 sets)
Performing sanity checks and gathering statistics...
info: complementing non-deterministic automata via determinization...
info:   N0      (10 st.,18 ed.,1 sets) -> (27 st.,432 ed.,2 sets) Comp(N0)
info:   P1      (5 st.,19 ed.,3 sets) -> (16 st.,136 ed.,1 sets) Comp(P1)
info:   N1      (5 st.,9 ed.,1 sets) -> (8 st.,128 ed.,2 sets) Comp(N1)
info: getting rid of any Fin acceptance...
info:   Comp(N0)        (27 st.,432 ed.,2 sets) -> (38 st.,220 ed.,2 sets)
info:   Comp(P1)        (16 st.,136 ed.,1 sets) -> (30 st.,163 ed.,1 sets)
info:   Comp(N1)        (8 st.,128 ed.,2 sets) -> (14 st.,55 ed.,2 sets)
info: check_empty P0*N0
info: check_empty Comp(N0)*Comp(P0)
error: Comp(N0)*Comp(P0) is nonempty; both automata accept the infinite word
       cycle{p0 & !p1 & p2; p0 & !p1 & p3; p0 & p1}
info: check_empty P0*N1
info: check_empty P1*N0
info: check_empty P1*N1
info: check_empty Comp(N1)*Comp(P1)
```

The formula to translate

Positive and negative translations are obtained from the two translators. Here `ltl2tgba` outputs generalized Büchi acceptance with up to 3 sets, and `modella` outputs Büchi automata. Note that `P0` has no edges.

Complementing by determinization creates "parity min even" automata. This is equivalent to "co-Büchi" for 1 acceptance set, and "1-pair Streett" for two sets.

Removing Fin sets as seen on Fig. 4.7.

Testing various products for emptiness, a bug is quickly found: the product `Comp(N0)*Comp(P0)` should have been empty. The issue obviously comes from the fact that automaton `P0` has no edge, so it recognizes the empty language.

Example 4.9: Verbose execution of `ltlcross -D` finding a bug in `modella 1.5.9`.

recent versions of their tools.

Given a formula φ and a list of translators $T_1, \ldots, T_m$ producing ω-automata from formulas, `ltlcross` calls each tool on φ and its negation, yielding a positive automaton $P_i = T_i(\varphi)$ and a negative automaton $N_i = T_i(\neg\varphi)$.

Now `ltlcross` (like `lbtt` did for generalized Büchi) will make sure that $\mathscr{L}(P_i \otimes N_j) = \emptyset$ for all i and j. It could be the case that P_i is a Rabin automaton, and that N_j is a Streett automaton, but clearly building the product $P_i \otimes N_j$ and testing its emptiness for arbitrary acceptance condition can be done using the techniques discussed in this chapter. However, this test is not enough to prove that all translators produce equivalent automata: for instance if one translator always returns empty automata, this test will pass. Therefore `ltlcross` also tests $\mathscr{L}(\overline{P_i} \otimes \overline{N_i}) = \emptyset$: when P_i and N_i are deterministic, complementing them is as simple as completing the automaton and complementing its acceptance condition. For non-deterministic automata, `ltlcross` offers the (potentially costly) option to determinize them, as illustrated by Figure 4.9.

Our determinization procedure currently takes transition-based Büchi automata as input (so we may have some preprocessing to do if the input has a different acceptance), and outputs automata with transition-based parity acceptance. It mixes the construction of Redziejowski [2012] with some optimizations of `ltl2dstar` [Klein and Baier, 2006, 2007] and a few of our own[6]. It would be nice to be able to determinize automata with arbitrary acceptance conditions; Varghese [2014] offers some clues.

[6] Yet unpublished.

5
SAT-based minimization of deterministic automata

LTL Synthesis [Finkbeiner and Schewe, 2005] and Probabilistic LTL Model Checking [Baier and Katoen, 2008] are two areas where it is useful to express linear-time temporal properties as deterministic ω-automata. Because it is well known that not all Büchi automata can be made deterministic, these applications use other acceptance conditions such as Rabin or Streett.

In this chapter we discuss the minimization of such deterministic automata, with arbitrary acceptance conditions. This is the result of an ongoing collaboration with Soheib Baarir.

5.1 Existing results

Minimizing deterministic Büchi automata, deterministic co-Büchi automata, and deterministic parity automata is known to be NP-complete [Schewe, 2010], so we should not hope for efficient minimizations that work for arbitrarily complex acceptance conditions.

However, and as already pointed out on page 13, minimizing automata that are weak and deterministic can be done in polynomial time [Löding, 2001]. For this subclass of automata, it does not even matter what the acceptance condition is: any (inherently) weak and deterministic can be rewritten as a Büchi or co-Büchi automaton without changing the transition structure.

For more complex automata, we do not know of efficient minimization procedures. Tools usually apply simplifications that do not guarantee a minimal result. Being able to see how far those simplifications are from the minimal automata would provide a good help to improve them.

5.2 Minimization via SAT-solving

A technique (and tool) for minimizing (state-based) deterministic Büchi automata was first presented by Ehlers [2010]. His tool, called `DBAminimizer`, will take deterministic Rabin automata produced by `ltl2dstar`, will convert it into a deterministic Büchi automaton if it exists [Krishnan et al., 1994], and finally will minimize the resulting Büchi automaton using a SAT-solver.

The tool implements a minimization loop, in which each iteration tries to synthesize an automaton that has one less state than the previous automaton.

The SAT-based synthesis of an equivalent automaton works as follows. A set of variables denoting the existence of each possible transition and the acceptance of each state in the "candidate" automaton. These variables are used to derive the values of another set of variables encoding a product between the reference automaton, and the candidate automaton. Finally, constraints are added such that for each cycle of the product automaton, its projection on the reference automaton is accepting if and only if its projection on the candidate automaton is also accepting.

[1] S. Baarir and A. Duret-Lutz. Mechanizing the minimization of deterministic generalized Büchi automata. In FORTE'14, vol. 8461 of LNCS, pp. 266–283. Springer, 2014

[2] S. Baarir and A. Duret-Lutz. SAT-based minimization of deterministic ω-automata. In LPAR'15, vol. 9450 of LNCS, pp. 79–87. Springer, 2015

5.3 *Our improvements*

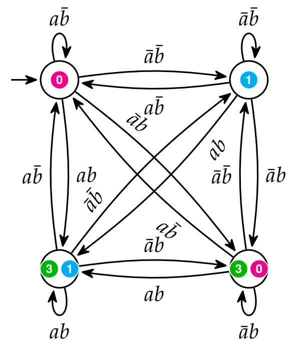
We added several improvements to the above minimization technique when we implemented it in Spot.

- First, Spot is able to detect obligation properties, so we can avoid SAT-solvers and use a polynomial minimization in this case.
- Second, while Spot will not always produce a deterministic Büchi automaton from an LTL formula when such automaton exists, it will often do so. And since the deterministic automata produced this way are usually much smaller than the Rabin automata produced by `ltl2dstar` from determinization, it made sense to first try that, and to adapt the minimization technique to transition-based acceptance.
- Third, we improve Ehlers [2010]'s encoding by adding several optimizations to reduce the number of variables and clauses needed by the encoding. Some of these optimizations are based on the knowledge of the strongly connected components of the original automaton (there is no need to track cycles outside of those), and knowing which strongly connected components of the original automaton are weak.
- Finally, we generalize the technique to deal with transition-based generalized Büchi acceptance[1], and then arbitrary acceptance conditions[2]. These generalizations come at the cost of additional clauses and variables.

Currently, the implementation has two different encodings: a simple one that works with deterministic transition-based Büchi automata as input and output, and a more general encoding that can input deterministic automata with an arbitrary acceptance, and output possibly another arbitrary acceptance. Both encodings can be configured to use state-based acceptance if needed.

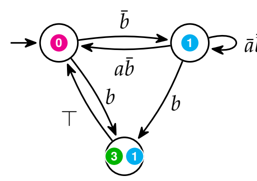
Figure 5.1 shows how a 4-state deterministic Rabin automaton constructed by `ltl2dstar` can be minimized into a deterministic Rabin automaton with 3 states if state-based acceptance is used, or just 1-state if transition-based acceptance is used. The bottom

```
$ ltlfilt -l -f 'GFa->GFb' |
> ltl2dstar - dra
```



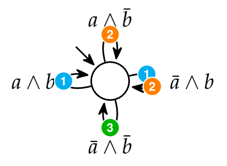
$(\mathsf{Inf}(0) \wedge \mathsf{Fin}(1)) \vee (\mathsf{Inf}(2) \wedge \mathsf{Fin}(3))$

```
$ autfilt -S --sat-min dra
```



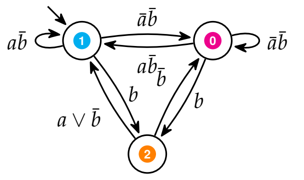
$(\mathsf{Inf}(0) \wedge \mathsf{Fin}(1)) \vee (\mathsf{Inf}(2) \wedge \mathsf{Fin}(3))$

```
$ autfilt --sat-min dra
```



$(\mathsf{Inf}(0) \wedge \mathsf{Fin}(1)) \vee (\mathsf{Inf}(2) \wedge \mathsf{Fin}(3))$

```
$ a='acc="parity max even 3"'
$ autfilt -S \
> --sat-min="$a,colored" dra
```



$\mathsf{Inf}(2) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(0))$

Figure 5.1: Minimization of an automaton output by `ltl2dstar` (the top one), using various acceptance conditions.

example of Figure 5.1 shows how we can also force an arbitrary acceptance condition on the output: here we want parity acceptance with numbers 0, 1, 2 such that a run is accepting if the maximum number seen infinitely often is even. The `colored` option constrains each state to belong to exactly one acceptance set (this is translated into additional constraints for the SAT solver), which is what one expects from a parity automaton.

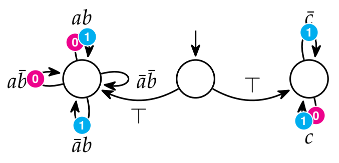
5.4 *Applications*

Let us assume an alphabet of size $|\Sigma|$. To minimize an automaton that uses an n states and m acceptance sets into another automaton that uses n' states and m' acceptance sets, our SAT encoding requires at most $O(n^2 \times n'^2 \times 2^{m+m'})$ variables and $O(n^2 \times n'^3 \times 2^{2m+m'} \times |\Sigma|)$ clauses.

For this reason the minimization procedure is not very practical for automata with a large number of states, or more that 5–6 acceptance sets. However, automata used in model checking are usually quite small, so attempting a SAT-based minimization can be worth a try.

Our main use case however, is to "mine" sub-optimal automata in the output of other tools. Indeed, it is easier to improve an LTL translator when you are provided with a list of formulas for which the output is far from optimal. Our papers contain experiments showing lists of formulas for which existing translators fail to produce reasonably-sized automata: looking at those automata can suggest improvements to the algorithms.

```
$ ltl2tgba '(GFa&GFb) | GFc'
```



```
$ ltl2tgba 'GF(a|c)&GF(b|c)'
```

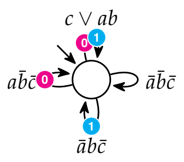


Figure 5.2: The translation offered for $(\mathsf{GF}a \land \mathsf{GF}b) \lor \mathsf{GF}c$ is inferior to that given for the equivalent formula $\mathsf{GF}(a \lor c) \land \mathsf{GF}(b \lor c)$.

As a simple example, we discovered that Spot translates the LTL formula $\varphi = (\mathsf{GF}a \land \mathsf{GF}b) \lor \mathsf{GF}c$ into a 3-state non-deterministic transition-based generalized Büchi automaton with 2 acceptance sets (Fig. 5.2). However, after determinizing this automaton and minimizing it, we discovered that there exists a 1-state deterministic automaton with the same acceptance. The equivalent formula $\mathsf{GF}(a \lor c) \land \mathsf{GF}(b \lor c)$ is translated to the minimal automaton directly.

This suggests that implementing LTL simplification rules that distribute through GF terms would be an improvement. But that is not the only option.

Since this technique works with arbitrary acceptance conditions as input and output, it can also be used to search of sub-optimal automata produced by other algorithms such as acceptance conversions, and other transformations.

6
Model checking of stutter-invariant properties

An ω-regular language is stutter-invariant if it is closed under the operation that duplicates some letter in a word or that removes some duplicate letter [Etessami, 1999].

We say that a formula is stutter-invariant if its language is. Stutter-invariant formula form an important subclass of temporal properties, as they enable model checkers to apply partial-order reduction techniques (e.g., [Clarke et al., 2000, Ch. 10] or [Baier and Katoen, 2008, Ch. 8]) to reduce the set of behaviors they must explore. Such partial-order reductions are implemented by explicit model checkers such as Spin [Holzmann, 2003, Ch. 9], LTSmin [Laarman et al., 2014], or DiVinE [Barnat et al., 2010b], to cite a few. Detecting stutter-invariant properties has also uses beyond partial-order reductions; for instance it is used to optimize the determinization construction implemented in `ltl2dstar` [Klein and Baier, 2007].

In this chapter, we present two results related to stutter invariance: the first is an efficient algorithm for deciding whether a property is stutter-invariant, the second is a generalization of a type of automata known as "testing automata" that can only represent stutter-invariant properties.

6.1 *Stutter-invariance checks*

It is widely known that any LTL formula that does not use the next-step operator X (a.k.a. an LTL\X formula) is stutter-invariant; this check is trivial to implement. Unfortunately there exist formulas using X that are stutter-invariant and whose usage is desirable [Păun and Chechik, 2003]. For instance, the stutter-invariant formula $\mathsf{F}(a \land \mathsf{X}(\neg a \land b))$ specifies that b will be true at a moment where a was just switched off.

Dallien and MacCaull [2006] built a tool that recognizes a stuttering LTL formula if (and only if) it matches one of the patterns of Păun and Chechik [2003]. This syntactical approach is efficient, but incomplete, as not all stutter-invariant formulas follow the recognized patterns.

A more definite procedure was given by Peled and Wilke [1997] as a construction that inputs an LTL formula φ with $|\varphi|$ symbols and n atomic propositions, and outputs an LTL\X formula φ' with $\mathrm{O}(4^n|\varphi|)$

```
$ ltlfilt --remove-x \
> -f 'F(a&X(!a&b))'
```

```
F(a ∧ ((a ∧ (a U(¬a ∧ b)) ∧ ((¬b U ¬a) ∨
(b U ¬a))) ∨ (¬a ∧ (¬a U(a ∧ ¬a ∧ b)) ∧
((¬b U a) ∨ (b U a))) ∨ (b ∧ (b U(¬a ∧ b ∧
¬b)) ∧ ((¬a U ¬b) ∨ (a U ¬b))) ∨ (¬b ∧
(¬b U(¬a ∧ b)) ∧ ((¬a U b) ∨ (a U b))) ∨
(¬a ∧ b ∧ (G¬a ∨ Ga) ∧ (G¬b ∨ Gb))))
```

```
$ ltlfilt --remove-x \
> -f 'F(a&XX(!a&b))'
```

```
F(a ∧ ((a ∧ ((¬b U ¬a) ∨ (b U ¬a)) ∧
(a U(¬a ∧ ((a ∧ (a U(¬a ∧ b)) ∧
((¬b U ¬a) ∨ (b U ¬a))) ∨ (¬a ∧
(¬a U(a ∧ ¬a ∧ b)) ∧ ((¬b U a) ∨
(b U a))) ∨ (b ∧ (b U(¬a ∧ b ∧ ¬b)) ∧
((¬a U ¬b) ∨ (a U ¬b))) ∨ (¬b ∧
(¬b U(¬a ∧ b)) ∧ ((¬a U b) ∨ (a U b))) ∨
(¬a ∧ b ∧ (G¬a ∨ Ga) ∧ (G¬b ∨
Gb)))))) ∨ (¬a ∧ ((¬b U a) ∨ (b U a)) ∧
(¬a U(a ∧ ((a ∧ (a U(¬a ∧ b)) ∧
((¬b U ¬a) ∨ (b U ¬a))) ∨ (¬a ∧
(¬a U(a ∧ ¬a ∧ b)) ∧ ((¬b U a) ∨
(b U a))) ∨ (b ∧ (b U(¬a ∧ b ∧ ¬b)) ∧
((¬a U ¬b) ∨ (a U ¬b))) ∨ (¬b ∧
(¬b U(¬a ∧ b)) ∧ ((¬a U b) ∨ (a U b))) ∨
(¬a ∧ b ∧ (G¬a ∨ Ga) ∧ (G¬b ∨
Gb)))))) ∨ (b ∧ ((¬a U ¬b) ∨ (a U ¬b)) ∧
(b U(¬b ∧ ((a ∧ (a U(¬a ∧ b)) ∧
((¬b U ¬a) ∨ (b U ¬a))) ∨ (¬a ∧
(¬a U(a ∧ ¬a ∧ b)) ∧ ((¬b U a) ∨
(b U a))) ∨ (b ∧ (b U(¬a ∧ b ∧ ¬b)) ∧
((¬a U ¬b) ∨ (a U ¬b))) ∨ (¬b ∧
(¬b U(¬a ∧ b)) ∧ ((¬a U b) ∨ (a U b))) ∨
(¬a ∧ b ∧ (G¬a ∨ Ga) ∧ (G¬b ∨
Gb)))))) ∨ (¬b ∧ ((¬a U b) ∨ (a U b)) ∧
(¬b U(b ∧ ((a ∧ (a U(¬a ∧ b)) ∧
((¬b U ¬a) ∨ (b U ¬a))) ∨ (¬a ∧
(¬a U(a ∧ ¬a ∧ b)) ∧ ((¬b U a) ∨
(b U a))) ∨ (b ∧ (b U(¬a ∧ b ∧ ¬b)) ∧
((¬a U ¬b) ∨ (a U ¬b))) ∨ (¬b ∧
(¬b U(¬a ∧ b)) ∧ ((¬a U b) ∨ (a U b))) ∨
(¬a ∧ b ∧ (G¬a ∨ Ga) ∧ (G¬b ∨
Gb)))))) ∨ ((G¬a ∨ Ga) ∧ (G¬b ∨ Gb) ∧
((a ∧ (a U(¬a ∧ b)) ∧ ((¬b U ¬a) ∨
(b U ¬a))) ∨ (¬a ∧ (¬a U(a ∧ ¬a ∧ b)) ∧
((¬b U a) ∨ (b U a))) ∨ (b ∧ (b U(¬a ∧ b ∧
¬b)) ∧ ((¬a U ¬b) ∨ (a U ¬b))) ∨ (¬b ∧
(¬b U(¬a ∧ b)) ∧ ((¬a U b) ∨ (a U b))) ∨
(¬a ∧ b ∧ (G¬a ∨ Ga) ∧ (G¬b ∨ Gb))))))
```

Example 6.1: Etessami's procedure on two formulas that differ by one X.

symbols, such that φ and φ' are equivalent iff they represent a stutter-invariant property. This construction, which proves that any stutter-invariant formula can be expressed without X, was later improved to $n^{\mathrm{O}(k)}|\varphi|$ symbols, where k is the X-depth of φ, by Etessami [2000] (see Example 6.1). If a disjunctive normal form is desired, Tian and Duan [2009] give a variant with size $\mathrm{O}(n2^n|\varphi|)$. To decide if an LTL formula φ is stutter-invariant, we build φ' using one of these constructions, and then check the equivalence of φ and φ'. This equivalence check can be achieved by translating these formulas into automata.

```
$ ltlfilt --remove-x -f "$f" \
> --equivalent-to "$f" -c
1
$ ltlfilt --remove-x -f "$g" \
> --equivalent-to "$g" -c
0
```

Example 6.2: Continuing the example of Fig 6.1, we can test whether a formula is stutter-invariant by testing whether it is equivalent to its rewriting by Etessami's procedure.

This approach, based on Etessami's procedure, was implemented in old versions of Spot[1] (and can still be performed explicitly, as in Example 6.2). However, two issues motivated us to look into alternative directions:

[1] A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In ATVA'13, vol. 8172 of LNCS, pp. 442–445. Springer, 2013

- As seen in Example 6.1, Etessami's rewriting function (called τ') can generate very large formulas. Testing equivalence via translation of these formulas can therefore be very costly.

- Etessami's procedure only works for LTL, but we wanted to decide the stutter invariance of PSL[2] properties as well. Dax et al. [2009], proposed a rewriting technique that is to PSL what Etessami's procedure is to LTL, however after implementing it we discovered it was incorrect, and no fix has been found so far.

[2] Cf. Sec. 7.2 p. 40.

Instead we took inspiration from the procedure used in `ltl2dstar` [Klein and Baier, 2007], where stutter-invariance is decided at the automaton level. Given an formula φ, they first build an equivalent Büchi automaton A_φ, then they construct an automaton A'_φ that accepts the smallest stutter-invariant language over-approximating the language of φ. The property φ is stutter-invariant iff A_φ and A'_φ have the same language, which can be checked by ensuring that the product $A'_\varphi \otimes A_{\neg\varphi}$ has an empty language.

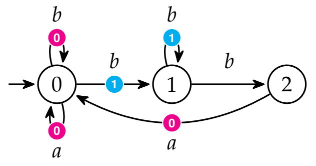
In a paper published at Spin'15[3], we proposed several alternatives to this construction.

[3] T. Michaud and A. Duret-Lutz. Practical stutter-invariance checks for ω-regular languages. In SPIN'15, vol. 9232 of LNCS, pp. 84–101. Springer, 2015

An easy intuition can be provided by seeing the construction of A'_φ as $A'_\varphi = \mathsf{cl}(\mathsf{sl}(A_\varphi))$, where sl (for "self-loopize") transforms the automaton so it accepts all the original words plus those you can obtain by duplicating some letters, and cl (for "closure") transforms the automaton so it additionally accepts words obtained by omitting duplicate letters (Fig 6.3).

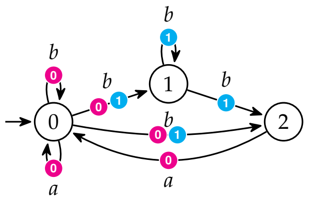
Because $\mathscr{L}(\mathsf{cl}(\mathsf{sl}(A_\varphi)) \subseteq \mathscr{L}(A_\varphi)$ by construction, testing the equivalence of the two automata can be done by testing just that $\mathscr{L}(A_{\neg\varphi} \otimes \mathsf{cl}(\mathsf{sl}(A_\varphi))) = \emptyset$. In this setup, our main result can be expressed as follows:

Theorem S. *Let φ be a property expressed as a ω-automaton A, and assume we know how to obtain $\overline{A}$. Testing φ for stutter-invariance is equivalent to testing the emptiness of any of the following products: $\mathsf{cl}(\mathsf{sl}(A)) \otimes \overline{A}$, $\mathsf{sl}(\mathsf{cl}(A)) \otimes \overline{A}$, $\mathsf{cl}(\mathsf{sl}_2(A)) \otimes \overline{A}$, $\mathsf{sl}_2(\mathsf{cl}(A)) \otimes \overline{A}$, $\mathsf{sl}(A) \otimes \mathsf{sl}(\overline{A})$, $\mathsf{sl}_2(A) \otimes \mathsf{sl}_2(\overline{A})$, or $\mathsf{cl}(A) \otimes \mathsf{cl}(\overline{A})$.*



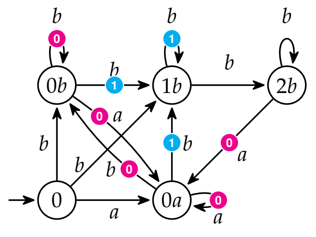
(a) A_1

```
$ autfilt --destut A1
```



(b) $\mathsf{cl}(A_1)$

```
$ autfilt --instut A1
```



(c) $\mathsf{sl}(A_1)$

Figure 6.3: Examples of application of the cl and sl operators.

	$\lvert AP\rvert = 1$	$\lvert AP\rvert = 2$	$\lvert AP\rvert = 3$
$\mathscr{L}(A_{\tau'(\varphi)} \otimes A_{\neg\varphi}) = \emptyset \land \mathscr{L}(A_{\neg\tau'(\varphi)} \otimes A_{\varphi}) = \emptyset$	0.32s	40.62s	>4801s (OOM)
$\mathscr{L}(A_{\neg(\varphi\leftrightarrow\tau'(\varphi))}) = \emptyset$	1.18s	3347.92s	
$\mathscr{L}(\mathsf{cl}(\mathsf{sl}(A_\varphi)) \otimes A_{\neg\varphi}) = \emptyset$	0.61s	1.91s	6.14s
$\mathscr{L}(\mathsf{sl}(\mathsf{cl}(A_\varphi)) \otimes A_{\neg\varphi}) = \emptyset$	0.61s	1.91s	6.10s
$\mathscr{L}(\mathsf{cl}(\mathsf{sl}_2(A_\varphi)) \otimes A_{\neg\varphi}) = \emptyset$	0.61s	1.89s	5.97s
$\mathscr{L}(\mathsf{sl}_2(\mathsf{cl}(A_\varphi)) \otimes A_{\neg\varphi}) = \emptyset$	0.61s	1.91s	5.97s
$\mathscr{L}(\mathsf{sl}(A_\varphi) \otimes \mathsf{sl}(A_{\neg\varphi})) = \emptyset$	0.61s	1.92s	6.18s
$\mathscr{L}(\mathsf{sl}_2(A_\varphi) \otimes \mathsf{sl}_2(A_{\neg\varphi})) = \emptyset$	0.61s	1.90s	5.99s
$\mathscr{L}(\mathsf{cl}(A_\varphi) \otimes \mathsf{cl}(A_{\neg\varphi})) = \emptyset$	0.60s	1.89s	5.94s
number of stutter-invariant formulas found	234	162	112

Table 6.4: Time to classify 500 random LTL formulas that all use the X operator and have the given number of atomic propositions.

In this theorem, sl_2 is an alternative construction that produces automata with the same language as those output by sl.

The surprise of this theorem is in the last three products: it is indeed possible to test stutter-invariance by implementing any one of the three functions cl, sl, sl_2, while previous work is equivalent to a combination of two [Klein and Baier, 2007].

In the context of deciding stutter invariance for LTL formulas, our experimental benchmark in the case of LTL formulas showed all the decision procedures of this theorem are orders-of-magnitude faster than our previous decision procedure based on Etessami's rewriting (Table 6.4). The reason is that all these procedures spend most of their time translating the LTL formulas in automata, so it is better to keep these formulas short.

If we ignore the translation time, which is common to all the checks mentioned in Theorem S, we observe that $\mathscr{L}(\mathsf{cl}(A_\varphi) \otimes \mathsf{cl}(A_{\neg\varphi})) = \emptyset$ is the most efficient in our implementation (see Table 6.6). As a consequence, this is now the default stutter-invariance check in Spot.

```
$ ltlfilt --stutter-inv \
> -f "$f" -f "$g"
F(a & X(!a & b))
```

Example 6.5: Continuing the example of Fig 6.1 and Fig 6.2: the `--stutter-invariant` option filters stutter-invariant formulas using the last check of Theorem S.

6.2 *Generalized Testing Automata*

This section was the subject of A. E. Ben Salem's Ph.D. thesis [2014] which I co-supervised with F. Kordon. However, the presentation of the material, using the notations of Chapter 4 to discuss the

Table 6.6: Cross-comparison of the checks of Theorem S on 40000 random LTL formulas with X. A value v on line (x) and column (y) indicates that there are v cases where check (x) was more than 10% slower than check (y). In other words, a line with many small numbers indicates a check that is usually faster than the others.

									run time	
		(1)	(2)	(3)	(4)	(5)	(6)	(7)	total	median
$\mathscr{L}(\mathsf{cl}(\mathsf{sl}(A_\varphi)) \otimes A_{\neg\varphi}) = \emptyset$	(1)		24615	38158	38593	1999	35200	39660	45.8s	162μs
$\mathscr{L}(\mathsf{sl}(\mathsf{cl}(A_\varphi)) \otimes A_{\neg\varphi}) = \emptyset$	(2)	244		38343	38832	91	34965	39813	34.9s	135μs
$\mathscr{L}(\mathsf{cl}(\mathsf{sl}_2(A_\varphi)) \otimes A_{\neg\varphi}) = \emptyset$	(3)	536	419		7413	67	10297	29495	11.0s	57μs
$\mathscr{L}(\mathsf{sl}_2(\mathsf{cl}(A_\varphi)) \otimes A_{\neg\varphi}) = \emptyset$	(4)	264	163	671		30	10223	28880	10.2s	55μs
$\mathscr{L}(\mathsf{sl}(A_\varphi) \otimes \mathsf{sl}(A_{\neg\varphi})) = \emptyset$	(5)	33410	39112	39746	39909		38403	39977	59.4s	208μs
$\mathscr{L}(\mathsf{sl}_2(A_\varphi) \otimes \mathsf{sl}_2(A_{\neg\varphi})) = \emptyset$	(6)	2689	2564	16896	18621	580		26693	11.7s	64μs
$\mathscr{L}(\mathsf{cl}(A_\varphi) \otimes \mathsf{cl}(A_{\neg\varphi})) = \emptyset$	(7)	16	13	3487	2993	11	2409		7.3s	39μs

acceptance condition, is new.

Testing automata were introduced by Hansen et al. [2002] as an alternative to Büchi automata for the automata theoretic approach to model checking (Fig. 1.1, p. 5). Contrary to Büchi automata which test the values of the different atomic propositions of the system (either ''light is on'' or ''light is off''), testing automata may only detect the changes of those values (''the status of light was changed''). Testing automata should remain in their current state as long as none of the observed properties of the system change. This has two consequences: first, testing-automata naturally capture only stutter-invariant properties, and second, they need two forms of accepting runs. They use a Büchi acceptance condition to accept some runs where atomic propositions are changing infinitely often, and another ''livelock acceptance condition'' to capture runs where no observed atomic proposition change.

Figure 6.7 shows an example of testing automata and presents the two different acceptances. Because the two forms of acceptance have to distinguish runs that stutter continuously from those that do not, the acceptance condition cannot precisely be expressed in terms of the Fin and Inf operators introduced in Chapter 4. However, if we make the stuttering explicit in the automaton and use transition-based acceptance, as in Figure 6.8, then the acceptance condition is similar to a Streett condition with a single pair.

Algorithms for constructing testing automata from LTL formulas (via Büchi automata), for constructing the product between a Kripke structure and a testing automaton, and for testing their emptiness were discussed by Hansen et al. [2002] and Geldenhuys and Hansen [2006]. In the general case, the emptiness check requires two passes: one pass for testing Büchi acceptance, and one pass for testing livelock acceptance. However, to prove the absence of $\emptyset$-labeled cycles, one has to make a dedicated pass on the product between the Kripke structure and the testing automaton.

Our first contribution was to implement testing automata, and reproduce the results observed by Geldenhuys and Hansen [2006]: although testing automata are usually larger than the corresponding Büchi automata, they usually yield smaller products with the system, and counterexamples are found quicker on average.[4] However, proving the absence of counterexample takes more time due to the 2-pass emptiness check. We also compared the testing automaton approach against one using transition-based generalized Büchi automata and found similar results.

However, once the stuttering self-loops are explicitly represented, one can more easily realize that the Fin(¬①) acceptance can be removed from the automaton by adding a new state dedicated to accepting stuttering behaviors, and making non-deterministic jumps into this state (Fig. 6.9).

This construction should look similar to the Fin-removal of Section 4.5, except we introduce non-deterministic jumps before entering a livelock state so that the non-deterministic jumps are not

```
$ ltl2tgta --ta 'a U Gb'
```

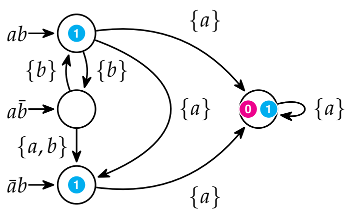


Figure 6.7: A testing automaton for $a \mathsf{U} \mathsf{G} b$. The initial state should be selected according to the initial value of a and b in the system. Transitions are labeled by sets of variables whose value must change in the system. If a and b do not change in the system, the testing automaton stutters in the current state. Infinite runs that stutter continuously in a ①-state are ''livelock accepting''. Runs that do not stutter continuously can also be Büchi-accepted if they visit ⓪-states infinitely often.

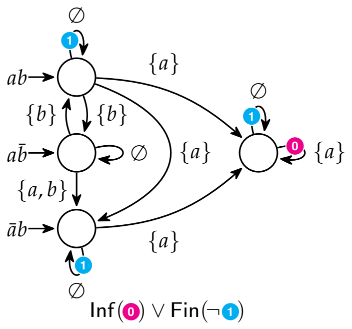


Figure 6.8: If stuttering actions are explicitly represented using $\emptyset$-labeled self-loops, then the livelock acceptance can be specified as visiting finitely many transitions not labeled by ①.

[4] A. E. Ben Salem, A. Duret-Lutz, and F. Kordon. Generalized Büchi automata versus testing automata for model checking. In SUMO'11, vol. 726 of Workshop Proceedings. CEUR, 2011

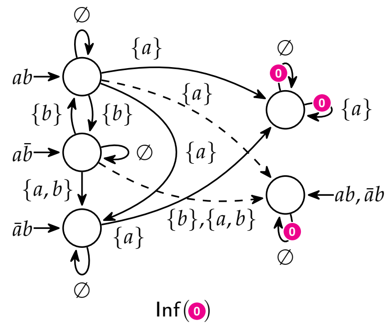


Figure 6.9: A 1-pass testing automaton can be obtained by using non-determinism (dashed lines) to capture stuttering runs into a dedicated state.

labeled by ∅: the only reason for that is to stay compatible with the testing automaton formalism, where stuttering should not move the automaton to another state. Note that states that are both Büchi accepting and livelock accepting can be fixed without introducing a new transition.

In practice, the product between these 1-pass testing automata and a Kripke structure can be seen as a Büchi automaton labeled by changesets, and can be checked for emptiness using any Büchi emptiness check, because emptiness checks do not look at labels. The second pass is no longer needed, since livelock acceptance is not used anymore. However, the extra state and non-deterministic transitions introduced in the automaton will synchronize with the same part of the Kripke structure that a second pass would have to explore; however since everything is explored at once, it requires more memory. Experiments show that 1-pass testing automata are comparable to classical testing automata as far as run time goes [Ben Salem, 2014, Sec. 4.4]. This is therefore a trade-off between ease of implementation (reusing existing emptiness checks for Büchi acceptance) and memory footprint.

We generalized 1-pass testing automata in a very natural way: instead of constructing them from state-based Büchi automata, we generate them from transition-based generalized Büchi automata, keeping the generalized Büchi acceptance. The result, which we call Transition-based Generalized Testing Automata[5] (TGTA), combines the advantages of TGBAs (the conciseness illustrated on Fig. 2.1(d), p. 9) with the use of changeset labels from testing automata. Furthermore, because we create TGTA from TGBA, which are transition-based, we can output smaller automata even when when generalized Büchi is not used. Figure 6.10 shows an example.

Experiments reveal that TGTA are generally better than TGBA, as seen on Figure 6.11. On empty products (i.e., verified properties ●) emptiness checks using TGTAs visit on the average 17.3% fewer transitions than emptiness checks using TGBAs, and 13.4% fewer transitions than those using TAs. For non-empty products (i.e., a counterexample exists ●), the picture is less clear, due to the fact that a different order of transitions can help to find a counterexample

[5] 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: 94–112, 2012

```
$ ltl2tgta 'a U Gb'
```

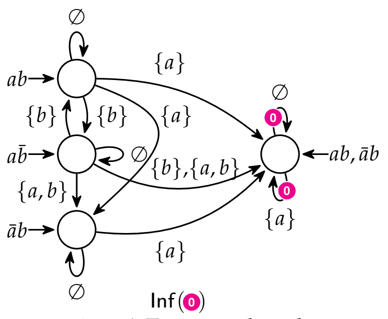


Figure 6.10: A Transition-based Generalized Testing Automaton for $a \mathsf{U} \mathsf{G} b$. Here the acceptance condition use a single Büchi set, but it could use more.

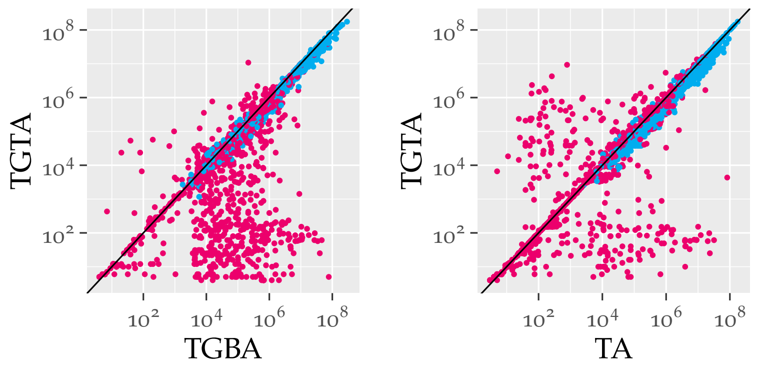


Figure 6.11: Comparison of the TGTA approach against the TA and TGBA approaches on a sample of 2049 pairs (model, formula) where the formula is verified (●) and 2049 pairs (model, formula) where the formula is violated (●). We compare the number of transitions visited by the emptiness checks. TGTA are better for all the dots below the diagonal. Refer to Ben Salem et al. [2012] for the experimental setup.

earlier: the differences can be so important that averaging them would make no sense. Let us just notice than the TGTA approach looses over the TGBA approach in nearly 39% of the non-empty (●) cases.

The above experiment, done in the context of explicit model checking was also successfully transposed to a symbolic setting and produced similar results.[6] In symbolic model checking, all variants of testing automata offer the nice guarantee that one can always stutter on any state. This can be used by saturation techniques [Ciardo et al., 2003, Thierry-Mieg et al., 2009] to speed up the construction of the stuttering part of the product.

[6] 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 TACAS'14, vol. 8413 of LNCS, pp. 440–454. Springer, 2014

7 Conclusion

7.1 Summary and Perspectives

In this document, I have presented a selected subset of the work I achieved since 2007. Some topics have been left out for reasons that will be discussed in Section 7.2.

Most of the contributions can probably be described as "practical improvements to a well-known theory". This comes from the fact that my motivation is to implement a toolbox (Spot) of efficient and reusable algorithms that are useful to researchers and teachers in the domain of ω-automata and model checking.

Let us quickly summarize and present perspectives for each chapter.

Chapter 2 presented all the techniques that have been implemented to make `ltl2tgba` one of the best tools to translate LTL formulas into small automata with few nondeterministic choices. While we do not believe there is a lot of room for improvement in the core algorithm used for translation, we think that the pre- and post-processings can be improved. Firstly, section 5.4 already hinted that better LTL simplifications could be found. Secondly, the simulation-based automata-reduction algorithm currently implemented only use "direct" or "backward" simulation-relations, but there exist coarser relations that can also be used for quotienting [Mayr and Clemente, 2013]. Most of these coarser relations have only be studied in the context of Büchi automata, or maybe generalized Büchi automata, but it would be interesting to extend those to generalized acceptance. Lastly, we should keep in mind that most LTL translators so far have tried to produce smaller automata or more deterministic automata, but we have shown (Section 2.9) that it makes sense to tune the generation of these automata according to how they will be used (e.g., the location of accepting states can affect model checkers using emptiness checks based on a nested-DFS). Working on translations from LTL to automata with different acceptance conditions (not generalized Büchi) is another interesting perspective, especially to produce deterministic automata, which is desirable for probabilistic model checking.

Chapter 3 presented some new ideas to parallelize emptiness checks for generalized Büchi automata. One is to decompose the property automaton in sub-automata of different strengths that can each be checked using an emptiness check algorithm tailored to this particular strength. This decomposition could in fact be pushed further in automata with many SCCs: there is no reason to restrict ourselves to one automaton per strength. A second idea was to use the union-find data structure to implement an SCC-based emptiness check: such a data structure is well-suited to be shared between multiple threads that explore the automaton. This work was the subject of Étienne Renault's Ph.D. thesis.

Chapter 4 discussed our adoption of automata with generalized acceptance condition, so that Spot would be usable with third-party tools that produce automata with various acceptance conditions. Doing so, we established a common format for exchanging ω-automata. We have shown how we can make some acceptance transformation in order to realize certain operations (like emptiness checks) for which Spot does not yet support any acceptance conditions. Working on an emptiness check that works for general acceptance is definitely on our agenda; in fact this is a topic we have started to work on with colleagues from the Dresden University of Technology, and from the Masaryk University of Brno.

Chapter 5 briefly presented a SAT-based technique to minimize deterministic ω-automata. This expensive procedure can be used to obtain minimal automata when that is desirable, and to discover areas where LTL translation and automata-simplification algorithms can be improved. We have also found this to be a useful tool to explore various generalized acceptance conditions, since we can now take a deterministic automaton with any acceptance and attempt to synthesize an equivalent automaton with another given acceptance condition. As these lines are written, an intern from EPITA is working towards making this minimization process more efficient using a better SAT-solver integration, some incremental SAT-solving techniques, and some new encoding ideas.

Finally in Chapter 6 we have presented some efficient ways to detect stutter invariant properties, and some results on using a generalization of what is known as testing automata. While the latter work (the subject of A. E. Ben Salem's Ph.D. thesis) was done before we started working with generalized acceptance, it is now much easier to present using these notions, as has been done in Section 6.2. In fact, a complete generalization of testing automata would be to implement them as ω-automata with arbitrary acceptance condition, but using changeset-labeled edges. It should be pointed out that the precise reason why the emptiness check using (generalized) testing automata is often better than using (generalized) Büchi automata is still unclear, and deserves further study.

7.2 *Omitted Contributions*

Four topics have been omitted from this document. The first two have not yet been published.

PSL, the Property Specification Language [Eisner and Fisman, 2006], is a huge industrial standard [PSL] that contains a linear fragment that is more expressive than LTL: it adds operators that allow using ω-regular expressions to denote finite prefixes that should be followed by some temporal properties. Spot supports some these PSL operators and the LTL translator has been extended to deal with the PSL operators and ω-regular expressions in some non-trivial ways, and some specific PSL simplifications have been added. The supported PSL fragment is comparable to the Linear Dynamic Logic introduced by Vardi [2011]. Although the techniques we have implemented have been presented in two invited talks[1], we have yet to write an article about it.

[1] At the Masaryk University of Brno in 2012, and at the Dresden University of Technology in 2015

As mentioned on Section 4.6, Spot implements a determinization procedure that inputs transition-based Büchi automata and outputs automata with transition-based parity acceptance. The algorithm is based on the construction of Redziejowski [2012], augmented with some optimizations from `ltl2dstar` [Klein and Baier, 2006, 2007], as well as a few of our own. The most important improvement is to take the strongly connected components of the input automaton into account during the determinization. This would deserve at least a tool paper.

The next two topics have been published, but were omitted from this document for size reason.

In 2011, we studied two new "hybrid" model-checking approaches[2][3], i.e., some techniques that are midway between explicit model checking (where the product $K_M \otimes A_{\neg\varphi}$ is represented as a graph) and symbolic model checking (where the product is represented symbolically, usually with decision diagrams). The two new techniques we presented, SOP (Symbolic Observation Product) and SLAP (Self-Loop Aggregation Product), both represent the product as a graph of aggregates: the nodes are symbolic sets of states of the original automata. Our experiment have shown that these techniques often outperform other existing hybrid or fully symbolic approaches.

[2] 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 ATVA'11, vol. 6996 of LNCS, pp. 336–350. Springer, 2011b

[3] 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, 2011a. URL `http://arxiv.org/abs/1106.5700`. Extended version of our ATVA'11 paper, presenting two new techniques instead of one

Finally, we recently published a study of several provisos for Partial-Order Reductions.[4] POR techniques are used during emptiness check to reduce the subset of K_M that needs to be explored. They do so by locally ignoring some outgoing transitions whose corresponding action in the model M is guaranteed to still be enabled later on. Ignoring such transitions can only be done if we guarantee that the corresponding action may not be continuously expanded along a cycle. A typical way to satisfy this constraint is to ensure that on each cycle, at least one state is fully expanded (i.e., none of its outgoing transitions are ignored). We reviewed several existing provisos to achieve this goal, and suggested new ones; discussing a total of 46 provisos.

[4] A. Duret-Lutz, F. Kordon, D. Poitrenaud, and E. Renault. Heuristics for checking liveness properties with partial order reductions. In ATVA'16, vol. 9938 of LNCS, pp. 340–356. Springer, 2016a

8
Bibliography

P. Arcaini, A. Gargantini, and E. Riccobene. Online testing of LTL properties for Java code. In HVC'13, vol. 8244 of LNCS, pp. 95–111. Springer, 2013.

A. Arnold. Deterministic and non ambiguous rational omega-languages. In Proc. of Automata on Infinite Words, École de Printemps d'Informatique Théorique, vol. 192 of LNCS, pp. 18–27. Springer, 1984.

S. Baarir and A. Duret-Lutz. Mechanizing the minimization of deterministic generalized Büchi automata. In FORTE'14, vol. 8461 of LNCS, pp. 266–283. Springer, 2014.

S. Baarir and A. Duret-Lutz. SAT-based minimization of deterministic ω-automata. In LPAR'15, vol. 9450 of LNCS, pp. 79–87. Springer, 2015.

T. Babiak, M. Křetínský, V. Řehák, and J. Strejček. LTL to Büchi automata translation: Fast and more deterministic. In TACAS'12, vol. 7214 of LNCS, pp. 95–109. Springer, 2012.

T. Babiak, T. Badie, A. Duret-Lutz, M. Křetínský, and J. Strejček. Compositional approach to suspension and other improvements to LTL translation. In SPIN'13, vol. 7976 of LNCS, pp. 81–98. Springer, 2013a.

T. Babiak, F. Blahoudek, M. Křetínský, and J. Strejček. Effective translation of LTL to deterministic Rabin automata: Beyond the (F, G)-fragment. In ATVA'13, vol. 8172 of LNCS, pp. 24–39. Springer, 2013b.

T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Křetínský, D. Müller, D. Parker, and J. Strejček. The Hanoi Omega-Automata format. In CAV'15, vol. 9206 of LNCS, pp. 479–486. Springer, 2015. See also `http://adl.github.io/hoaf/`.

C. Baier and J.-P. Katoen. Principles of Model Checking. The MIT Press, 2008.

C. Baier, S. Kiefer, J. Klein, S. Klüppelholz, D. Müller, and J. Worrell. Markov chains and unambiguous Büchi automata. Technical Report 1605.00950, ArXiv, 2016a. URL `http://arxiv.org/abs/1605.00950`. Extended version of their CAV'16 paper, with appendices.

C. Baier, S. Kiefer, J. Klein, S. Klüppelholz, D. Müller, and J. Worrell. Markov chains and unambiguous Büchi automata. In CAV'16, vol. 9779 of LNCS, pp. 23–42. Springer, 2016b.

J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In ASE'03, pp. 106–115. IEEE Computer Society, 2003.

J. Barnat, L. Brim, and J. Chaloupka. From distributed memory cycle detection to parallel LTL model checking. In FMICS'05, vol. 133 of ENTCS, pp. 21–39, 2005.

J. Barnat, L. Brim, and P. Ročkai. A time-optimal on-the-fly parallel algorithm for model checking of weak LTL properties. In ICFEM'09, vol. 5885 of LNCS, pp. 407–425. Springer, 2009.

J. Barnat, L. Brim, and P. Ročkai. Scalable shared memory LTL model checking. STTT, 12(2):139–153, 2010a.

J. Barnat, L. Brim, and P. Ročkai. Parallel partial order reduction with topological sort proviso. In SEFM'10, pp. 222–231. IEEE Computer Society Press, 2010b.

J. Barnat, P. Bauch, N. Beneš, L. Brim, J. Beran, and T. Kratochvíla. Analysing sanity of requirements for avionics systems. Formal Aspects of Computing, 28(1):45–63, 2016.

P. Bauch. Automating Software Development with Explicit Model Checking. PhD thesis, Masaryk University, 2015.

A. E. Ben Salem. Improving the Model Checking of Stutter-Invariant LTL Properties. PhD thesis, Université Pierre et Marie Curie - Paris VI, Paris, France, 2014.

A. E. Ben Salem, A. Duret-Lutz, and F. Kordon. Generalized Büchi automata versus testing automata for model checking. In SUMO'11, vol. 726 of Workshop Proceedings. CEUR, 2011.

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), 7400:94–112, 2012.

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 TACAS'14, vol. 8413 of LNCS, pp. 440–454. Springer, 2014.

M. Benedikt, R. Lenhardt, and J. Worrell. LTL model checking of interval markov chains. In 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13), vol. 7795 of LNCS, pp. 32–46. Springer, 2013.

F. Blahoudek, M. Křetínský, and J. Strejček. Comparison of LTL to deterministic Rabin automata translators. In LPAR'13, vol. 8312 of LNCS, pp. 164–172. Springer, 2013.

F. Blahoudek, A. Duret-Lutz, M. Křetínský, and J. Strejček. Is there a best Büchi automaton for explicit model checking? In SPIN'14, pp. 68–76. ACM, 2014.

F. Blahoudek, A. Duret-Lutz, V. Rujbr, and J. Strejček. On refinement of Büchi automata for explicit model checking. In SPIN'15, vol. 9232 of LNCS, pp. 66–83. Springer, 2015.

F. Blahoudek, M. Heizmann, S. Schewe, J. Strejček, and M.-H. Tsai. Complementing semi-deterministic Büchi automata. In TACAS'16, vol. 9636 of LNCS, pp. 770–787. Springer, 2016.

N. Bousquet and C. Löding. Equivalence and inclusion problem for strongly unambiguous Büchi automata. In LATA'10, vol. 6031 of LNCS, pp. 118–129. Springer, 2010.

L. Brim, I. Černá, P. Krcal, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In FSTTCS'01, pp. 96–107. Springer, 2001.

L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed LTL model-checking. In FMCAD'04, vol. 3312 of LNCS, pp. 352–366. Springer, 2004.

J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In APLAS'12, vol. 7705 of LNCS, pp. 350–367. Springer, 2012.

D. Bucur. Temporal monitors for TinyOS. In RV'12, vol. 7687 of LNCS, pp. 96–109. Springer, 2012.

I. Černá and R. Pelánek. Relating hierarchy of temporal properties to model checking. In MFCS'03, vol. 2747 of LNCS, pp. 318–327. Springer, 2003.

I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In SPIN'03, vol. 2648 of LNCS, pp. 49–73. Springer, 2003.

K. Chatterjee, A. Gaiser, and J. Křetínský. Automata with generalized Rabin pairs for probabilistic model checking and LTL synthesis. In CAV'13, vol. 8044 of LNCS, pp. 559–575. Springer, 2013.

G. Ciardo, R. M. Marmorstein, and R. Siminiceanu. Saturation unbound. In TACAS'03, vol. 2619 of LNCS, pp. 379–393. Springer, 2003.

J. Cichoń, A. Czubak, and A. Jasiński. Minimal Büchi automata for certain classes of LTL formulas. In DEPCOS'09, pp. 17–24. IEEE Computer Society, 2009.

E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking. The MIT Press, 2000.

J.-M. Couvreur. On-the-fly verification of temporal logic. In FM'99, vol. 1708 of LNCS, pp. 253–271. Springer, 1999.

J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In SPIN'05, vol. 3639 of LNCS, pp. 143–158. Springer, 2005.

J. Dallien and W. MacCaull. Automated recognition of stutter-invariant LTL formulas. Atlantic Electronic Journal of Mathematics, (1):56–74, 2006.

M. d'Amorim and G. Roşu. Efficient monitoring of ω-languages. In CAV'05, vol. 3576 of LNCS, pp. 364–378. Springer, 2005.

C. Dax, J. Eisinger, and F. Klaedtke. Mechanizing the powerset construction for restricted classes of ω-automata. In ATVA'07, vol. 4762 of LNCS. Springer, 2007.

C. Dax, F. Klaedtke, and S. Leue. Specification languages for stutter-invariant regular properties. In CAV'09, vol. 5799 of LNCS, pp. 244–254. Springer, 2009.

E. W. Dijkstra. EWD 376: Finding the maximum strong components in a directed graph. `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF`, 1973.

E. W. Dijkstra. Finding the maximal strong components in a directed graph. In A Discipline of Programming, chapter 25, pp. 192–200. Prentice-Hall, 1976.

A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In ATVA'13, vol. 8172 of LNCS, pp. 442–445. Springer, 2013.

A. Duret-Lutz. LTL translation improvements in Spot 1.0. Int. J. on Critical Computer-Based Systems, 5 (1/2):31–54, 2014.

A. Duret-Lutz and D. Poitrenaud. SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In MASCOTS'04, pp. 76–83. IEEE Computer Society Press, 2004.

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, 2011a. URL `http://arxiv.org/abs/1106.5700`. Extended version of our ATVA'11 paper, presenting two new techniques instead of one.

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 ATVA'11, vol. 6996 of LNCS, pp. 336–350. Springer, 2011b.

A. Duret-Lutz, F. Kordon, D. Poitrenaud, and E. Renault. Heuristics for checking liveness properties with partial order reductions. In ATVA'16, vol. 9938 of LNCS, pp. 340–356. Springer, 2016a.

A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In ATVA'16, vol. 9938 of LNCS, pp. 122–129. Springer, 2016b.

S. Dutta and M. Y. Vardi. Assertion-based flow monitoring of SystemC models. In MEMOCODE'14, pp. 145–154. IEEE Computer Society, 2014.

M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In FMSP'98, pp. 7–15. ACM Press, 1998.

R. Ehlers. Minimising deterministic Büchi automata precisely using SAT solving. In SAT'10, vol. 6175 of LNCS, pp. 326–332. Springer, 2010.

R. Ehlers and B. Finkbeiner. On the virtue of patience: minimizing Büchi automata. In SPIN'10, vol. 6349 of LNCS, pp. 129–145. Springer, 2010.

C. Eisner and D. Fisman. A Practical Introduction to PSL. Series on Integrated Circuits and Systems. Springer, 2006.

E. A. Emerson and C.-L. Lei. Modalities for model checking: branching time strikes back. In POPL'85, pp. 84–96. ACM, 1985.

E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. Science of Computer Programming, 8(3):275–306, 1987.

K. Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In CAV'99, vol. 1633 of LNCS, pp. 236–248. Springer, 1999.

K. Etessami. A note on a question of Peled and Wilke regarding stutter-invariant LTL. Information Processing Letters, 75(6):261–263, 2000.

K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In Concur'00, vol. 1877 of LNCS, pp. 153–167. Springer, 2000.

S. Evangelista, L. Petrucci, and S. Youcef. Parallel nested depth-first searches for LTL model checking. In ATVA'11, vol. 6996 of LNCS, pp. 381–396. Springer, 2011.

S. Evangelista, A. Laarman, L. Petrucci, and J. van de Pol. Improved multi-core nested depth-first search. In ATVA'12, vol. 7561 of LNCS, pp. 269–283. Springer, 2012.

B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In LICS'05, pp. 321–330, 2005.

Ł. Fronc and A. Duret-Lutz. LTL model checking with Neco. In ATVA'13, vol. 8172 of LNCS, pp. 451–454. Springer, 2013. Code moved to `https://github.com/Lvyn/neco-net-compiler`.

H. N. Gabow. Path-based depth-first search for strong and biconnected components. Information Processing Letters, 74(3-4):107–114, 2000.

P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In CAV'01, vol. 2102 of LNCS, pp. 53–65. Springer, 2001.

J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In SPIN'06, vol. 3925 of LNCS, pp. 53–70. Springer, 2006.

J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In TACAS'04, vol. 2988 of LNCS, pp. 205–219. Springer, 2004.

J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. Theoretical Computer Science, 345(1):60–82, 2005. Conference paper selected for journal publication.

D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In FORTE'02, vol. 2529 of LNCS, pp. 308–326. Springer, 2002.

E. M. Hahn, Y. Li, S. Schewe, A. Turrini, and L. Zhang. iscasMc: A Web-Based Probabilistic Model Checker. In FM'14, vol. 8442 of LNCS, pp. 312–317. Springer, 2014.

H. Hansen, W. Penczek, and A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. In FMICS'02, vol. 66(2) of ENTCS. Elsevier, 2002.

K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi. Towards manipulation planning with temporal logic specifications. In ICRA'15, pp. 346–352. IEEE Computer Society, 2015.

G. J. Holzmann. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, 2003.

G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. In Proc. of the 2nd Spin Workshop, vol. 32 of DIMACS. American Mathematical Society, 1996.

G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification techniques. IEEE Transaction on Software Engineering, 37(6):845–857, 2011.

K. Klai and D. Poitrenaud. MC-SOG: An LTL model checker based on symbolic observation graphs. In Proc. of Petri Nets'08, vol. 5062 of LNCS, pp. 288–306. Springer, 2009.

J. Klein and C. Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. Theoretical Computer Science, 363(2):182–195, 2006.

J. Klein and C. Baier. On-the-fly stuttering in the construction of deterministic ω-automata. In CIAA'07, vol. 4783 of LNCS, pp. 51–61. Springer, 2007.

Z. Komárková and J. Křetínský. Rabinizer 3: Safraless translation of LTL to small deterministic automata. In ATVA'14, vol. 8837 of LNCS, pp. 235–241. Springer, 2014.

J. Křetínský and J. Esparza. Deterministic automata for the (F,G)-fragment of LTL. In CAV'12, vol. 7358 of LNCS, pp. 7–22. Springer, 2012.

S. C. Krishnan, A. Puri, and R. K. Brayton. Deterministic ω-automata vis-a-vis deterministic Büchi automata. In ISAAC'94, vol. 834 of LNCS, pp. 378–386. Springer, 1994.

R. P. Kurshan. Complementing deterministic Büchi automata in polynomial time. J. Comput. Syst. Sci., 35 (1):59–71, 1987.

A. Laarman and J. van de Pol. Variations on multi-core nested depth-first search. In PDMC'11, pp. 13–28, 2011.

A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs. Multi-core nested depth-first search. In ATVA'11, vol. 6996 of LNCS, pp. 321–335. Springer, 2011.

A. Laarman, E. Pater, J. van de Pol, and H. Hansen. Guard-based partial-order reduction. STTT, pp. 1–22, 2014.

B. Le Saëc and I. Litovsky. On the minimization problem for ω-automata. In MFCS'94, vol. 841 of LNCS, pp. 504–514. Springer, 1994.

C. Lemieux, D. Park, and I. Beschastnikh. General LTL specification mining. In ASE'15, pp. 81–92. IEEE Computer Society, 2015.

R. Lenhardt. Tulip: Model checking probabilistic systems using expectation maximisation algorithm. In QEST'13, pp. 155–159. Springer, 2013.

J. Li, L. Zhang, G. Pu, M. Y. Vardi, and J. He. LTL satisfiability checking revisited. In TIME'13, pp. 91–98. IEEE Computer Society, 2013.

C. Löding. Efficient minimization of deterministic weak ω-automata. Information Processing Letters, 79 (3):105–109, 2001.

Z. Manna and A. Pnueli. A hierarchy of temporal properties. In PODC'90, pp. 377–410. ACM, 1990.

S. Maoz and J. O. Ringert. GR(1) synthesis for LTL specification patterns. In FSE'15, pp. 96–106. ACM, 2015.

R. Mayr and L. Clemente. Advanced automata minimization. In POPL'13, pp. 63–74. ACM New York, 2013.

T. Michaud and A. Duret-Lutz. Practical stutter-invariance checks for ω-regular languages. In SPIN'15, vol. 9232 of LNCS, pp. 84–101. Springer, 2015.

M. Michel. Algèbre de machines et logique temporelle. In STACS'84, vol. 166 of LNCS, pp. 287–298, 1984.

S. Mochizuki, M. Shimakawa, S. Hagihara, and N. Yonezaki. Fast translation from LTL to büchi automata via non-transition-based automata. In ICFEM'14, vol. 8829 of LNCS, pp. 364–379. Springer, 2014. The tool is now at `https://sites.google.com/site/yonezakilab/tools`.

V. Molnár, D. Darvas, A. Vörös, and T. Bartha. Saturation-based incremental ltl model checking with inductive proofs. In TACAS'15, vol. 9035 of LNCS, pp. 643–657. Springer, 2015.

D. O. Păun and M. Chechik. On closure under stuttering. Formal Aspects of Computing, 14(4):342–368, 2003.

R. Pelánek. BEEM: benchmarks for explicit model checkers. In Proc. of the 14th international SPIN conference on Model checking software, LNCS, pp. 263–267. Springer, 2007.

D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. Information Processing Letters, 63(5):243–246, 1997.

F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. Computing in Science and Engineering, 9(3):21–29, 2007. See also `http://ipython.org`.

PSL. Standard for property specification language (PSL). IEC 62531 Ed. 1 (2007-11) (IEEE Std 1850-2005), 2007.

R. Redziejowski. An improved construction of deterministic omega-automaton using derivatives. Fundamenta Informaticae, 119(3-4):393–496, 2012.

E. Renault. Contribution aux tests de vacuité pour le model checking explicite. PhD thesis, Université Pierre et Marie Curie - Paris VI, 2014.

E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In LPAR'13, vol. 8312 of LNCS, pp. 668–682. Springer, 2013a.

E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Strength-based decomposition of the property Büchi automaton for faster model checking. In TACAS'13, vol. 7795 of LNCS, pp. 580–593. Springer, 2013b.

E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Parallel explicit model checking for generalized Büchi automata. In TACAS'15, vol. 9035 of LNCS, pp. 613–627. Springer, 2015.

E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Variations on parallel explicit model checking for generalized Büchi automata. Int. J. on Software Tools for Technology Transfer (STTT), 2016. DOI 10.1007/s10009-016-0422-5.

K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In SPIN'07, vol. 4595 of LNCS, pp. 149–167. Springer, 2007.

S. Safra. Complexity of Automata on Infinite Objects. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, 1989.

S. Safra and M. Y. Vardi. On ω-automata and temporal logic. In STOC'89, pp. 127–137. ACM, 1989.

S. Schewe. Beyond hyper-minimisation—minimising DBAs and DPAs is NP-complete. In FSTTCS'10, vol. 8 of Leibniz International Proceedings in Informatics (LIPIcs), pp. 400–411. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

V. Schuppan and L. Darmawan. Evaluating LTL satisfiability solvers. In Automated Technology for Verification and Analysis, vol. 6996 of LNCS, pp. 397–413. Springer, 2011. URL `http://dx.doi.org/10.1007/978-3-642-24372-1_28`.

D. R. Schäfer, T. Bach, M. A. Tariq, and K. Rothermel. Increasing availability of workflows executing in a pervasive environment. In SCC'14, pp. 717–724. IEEE Computer Society, 2014.

R. Sebastiani and S. Tonetta. "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In CHARME'03, vol. 2860 of LNCS, pp. 126–140. Springer, 2003.

L. Shan, Z. Qin, S. Li, R. Zhang, and X. Yang. Conversion algorithm of Linear-time Temporal Logic to Büchi automata. Journal of Software, 9(4):970–976, 2014.

S. Sickert, J. Esparza, S. Jaax, and J. Křetínský. Limit-Deterministic Büchi Automata for Linear Temporal Logic, vol. 9780 of LNCS, pp. 312–332. Springer, 2016.

F. Somenzi and R. Bloem. Efficient Büchi automata for LTL formulæ. In CAV'00, vol. 1855 of LNCS, pp. 247–263. Springer, 2000.

M. Staats and M. P. E. Heimdahl. Partial Translation Verification for Untrusted Code-Generators, vol. 5256 of LNCS, pp. 226–237. Springer, 2008.

D. Tabakov and M. Y. Vardi. Optimized temporal monitors for SystemC. In RV'10, vol. 6418 of LNCS, pp. 436–451. Springer, 2010.

D. Tabakov, K. Y. Rozier, and M. Y. Vardi. Optimized temporal monitors for SystemC. Formal Methods in System Design, 41(3):236–268, 2012.

R. Tarjan. Depth-first search and linear graph algorithms. In Conference records of the 12th Annual IEEE Symposium on Switching and Automata Theory, pp. 114–121. IEEE, 1971. Later republished [Tarjan, 1972].

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

R. E. Tarjan. Efficiency of a good but not linear set union algorithm. Journal of the ACM (JACM), 22(2): 215–225, 1975.

H. Tauriainen. Automata and Linear Temporal Logic: Translation with Transition-based Acceptance. PhD thesis, Helsinki University of Technology, Espoo, Finland, 2006.

H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. STTT, 4(1):57–70, 2002.

Y. Thierry-Mieg. Symbolic model-checking using ITS-Tools. In TACAS'15, pp. 231–237. Springer, 2015.

Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical set decision diagrams and regular models. In TACAS'09, vol. 5505 of LNCS, pp. 1–15. Springer, 2009.

C. Tian and Z. Duan. A note on stutter-invariant PLTL. Information Processing Letters, 109(13):663–667, 2009.

M.-H. Tsai, Y.-K. Tsay, and Y.-S. Hwang. GOAL for Games, Omega-Automata, and Logics. In CAV'13, pp. 883–889. Springer, 2013.

M. Y. Vardi. Automata-theoretic model checking revisited. In VMCAI'07, vol. 4349 of LNCS. Springer, 2007. Invited paper.

M. Y. Vardi. The rise and fall of LTL. Invited talk to the 2nd International Symposium on Games, Automata, Logics and Formal Verification (GANDALF'11), 2011.

T. Varghese. Parity and Generalized Büchi Automata — determinisation and complementation. PhD thesis, University of Liverpool, 2014.

A
Selected uses of Spot

At the time of writing Google Scholar reports more than 126 citations of the paper in which we originally presented Spot in 2004[1]. This paper has remained the main paper to cite about Spot for 9 years, until we published some tool papers about Spot 1.0 in 2013[2] (cited 21 times) and Spot 2.0 in 2016[3]. Not all these 147 citations are actual uses of Spot. In this appendix, we review a sample of the some work that have been built using Spot in some way.

[1] A. Duret-Lutz and D. Poitrenaud. SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In MASCOTS'04, pp. 76–83. IEEE Computer Society Press, 2004

[2] A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In ATVA'13, vol. 8172 of LNCS, pp. 442–445. Springer, 2013

[3] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In ATVA'16, vol. 9938 of LNCS, pp. 122–129. Springer, 2016b

A.1 Spot as an LTL-to-BA translator

Spot is used as an off-the-shelf translator from LTL to Büchi automata, or to generalized Büchi automata in context such as:

- creation of automata for monitoring C code [Staats and Heimdahl, 2008],
- generation of initial BA before SAT-based minimization [Ehlers and Finkbeiner, 2010],
- generation of automata before conversion to monitor [Tabakov and Vardi, 2010, Tabakov et al., 2012] before Spot could do it, or direct conversion to monitor [Bucur, 2012, Arcaini et al., 2013, Dutta and Vardi, 2014] now that it can,
- generation of DFA from LTL formulas that are guarantee properties [He et al., 2015],
- generation of BA before conversion to deterministic Rabin automata [Blahoudek et al., 2013, Sickert et al., 2016]
- generation of a GBA in a probabilistic model checker [Hahn et al., 2014] (they hope for a determinisic GBA, but will determinize the result to Rabin if not).

Spot has been used in several translation benchmarks, either by authors of other translators [Babiak et al., 2012, Mochizuki et al., 2014, Shan et al., 2014, e.g.,], or by authors of benchmarks [Rozier and Vardi, 2007, Cichoń et al., 2009].

Some of those benchmarks are difficult to interpret because they do not always mention the version of Spot and the options used. Before Spot 1.0, there were no public command-line utility to translate LTL formulas into automata: Spot had a program called `ltl2tgba` in its test suite, which is what people used for benchmarking, but this badly named binary was a crude front-end to most of the algorithms available in Spot, and it required a careful selection of several cryptic options to produce an optimized output. Today this testing program has been renamed, and there is a public tool called `ltl2tgba` that only does translation and has all optimizations turned on by default (favoring quality over execution time).

The benchmark of Rozier and Vardi [2007] and of Cichoń et al. [2009] have played some important role in the history of Spot, as we used them (after publication) to improve its performance. Some benchmarks released alongside Spot 0.9.1 show that the time needed to translate 100 formulas from Cichoń et al. [2009] was divided by 2.83 between version 0.8.3 and version 0.9.1.[4]

[4] `https://www.lrde.epita.fr/dload/spot/bench-0.9.1.pdf`

Finally, Rozier and Vardi [2007] presents Spot 0.3 as the only tool of their benchmark that ‘‘can be considered an industrial quality tool’’ and consider Spot as ‘‘the best explicit LTL translator in our experiments’’. Nonetheless using explicit translators (i.e., translator that creates automatas represented as graphs) for satisfiability checking is less efficient than using symbolic translators, so Spot is not the ideal tool to use for this task. Further work by Li et al. [2013] has built upon those two quotes to justify comparing a new LTL satisfiability solver against Spot; however, there exists more challenging tools that are actually dedicated to LTL satisfiability [Schuppan and Darmawan, 2011].

A.2 Spot as a research/development toolbox

Tools distributed with Spot have many features that are convenient building-blocks for experiments. Here are some examples of uses:
- Generation of random LTL formulas [Molnár et al., 2015],
- Filtering of unique BAs up to one isomorphism [Blahoudek et al., 2016],
- Simplification of BAs [Blahoudek et al., 2016],
- Syntax conversion of LTL formulas [Maoz and Ringert, 2015],
- The `ltlcross` tool was used by the authors of `ltl3ba`, `ltl3dra`, `ltl2dstar`, and `Rabinizer3` to test recent version of their translators.[5]

[5] Personal communications with those tools' authors.

A.3 Spot as a library to build model checkers

As Spot offers the bare minimum needed to build a model checker over a custom state-space, it was used in a couple of tools for this purpose.

MC-SOG [Klai and Poitrenaud, 2009] is a model-checker based on symbolic observation graphs (SOG), in which each state symbolically encodes a set of states of the model. MC-SOG uses Spot for the dashed area of Figure 1.1 (p. 5), and implements the rest.

Neco, a compiler for turning Petri nets into native shared libraries that allow fast on-the-fly exploration of the state-space, was similarly linked with Spot to provide model-checking capabilities.[6]

[6] Ł. Fronc and A. Duret-Lutz. LTL model checking with Neco. In ATVA'13, vol. 8172 of LNCS, pp. 451–454. Springer, 2013. Code moved to `https://github.com/Lvyn/neco-net-compiler`

As already mentioned in Section 7.2 we generalized the SOG idea to Self-Loop Aggregation Product[7] (SLAP) and Symbolic Observation Product[8] (SOP). A model checking using these techniques has been implemented in ITS-Tools [Thierry-Mieg, 2015]. This time ITS-Tools provides an on-the-fly interface for building those hybrid products on-the-fly, and Spot is used over this interface to perform the emptiness check.

[7] 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 ATVA'11, vol. 6996 of LNCS, pp. 336–350. Springer, 2011b

[8] 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, 2011a. URL `http://arxiv.org/abs/1106.5700`. Extended version of our ATVA'11 paper, presenting two new techniques instead of one

A.4 Spot as an LTL or ω-automaton library

Brotherston et al. [2012] used Spot in a theorem prover called Cyclist in which proof trees may be cyclic (hence the name) and need to satisfy some transition-based Büchi acceptance. Spot 0.8 was used since it was able to provide emptiness-checks for this type of acceptance, and also because it implemented a complementation algorithm (via a transition-based version of Safra's determinization [Safra, 1989]). The current version of Cyclist now uses Spot 2 and its new transition-based determinization.

In the context of executing a workflow over multiple replicas for failure recovery, Schäfer et al. [2014] used the LTL translation routine of Spot to synthesize structurally different replicas from an identical LTL specification of the workflow. They modified the translation routine of Spot to include another parameter needed for their work.

Lemieux et al. [2015] use Spot just to parse LTL formulas and explore their syntactic tree.

Bauch [2015] and Barnat et al. [2016] used Spot as a C++ library for translating LTL formulas into automaton in a tool called Looney to check the sanity of real-world sets of requirements. They report that the translator they used before was inefficient on large formulas, and that switching to Spot "accelerated the process by several orders of magnitude on larger sets of formulae".

Very recently, Blahoudek et al. constructed two tools using Spot:

- `ltl3hoa`[9] translates LTL formulas into small non-deterministic automata using arbitrary acceptance conditions (as suggested by the LTL formula). It reuses the LTL parsing and simplification routines of Spot, as well as all the automata simplification routines; this way they focus only on their translation algorithm.
- `seminator`[10] converts ω-automata into semi-deterministic automata. They reuse the automaton parser of Spot, and its automata simplification routines. Again, they only needed to focus on the determinization-algorithms they wanted to implement.

[9] The code is at `https://github.com/jurajmajor/ltl3hoa` and they submitted an article to LICS'17.

[10] The code is at `https://github.com/mklokocka/seminator` and they submitted an article to LPAR-21.

A.5 Spot as a teaching environment

We use Spot at EPITA in an introductory lecture on model checking. Using the Python bindings in the IPython/Jupyter notebook, a web application for interactive programming [Pérez and Granger, 2007], the students can experiment with LTL and automata, and they can even build a small model checker by assembling the operations of Figure 1.1 (p. 5) in a few lines of Python code. Examples are given in Figure 2 of our ATVA'16 tool paper.[11]

[11] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In ATVA'16, vol. 9938 of LNCS, pp. 122–129. Springer, 2016b