Implementation of omega-automata minimization techniques in "Spot", a model checking library.

When "Spot" SAT-based minimization meets incremental SAT solving.

Paris, January 2017

Submitted by:
Alexandre GBAGUIDI AÏSSE
Student in second year or computer engineering
at EPITA
gbagui_a@epita.fr

Supervised by:
Alexandre Duret-Lutz
Assistant Professor at LRDE(Research and
Development Laboratory of EPITA)
adl@lrde.epita.fr

Contents

Part I

Report

Chapter 1

LRDE Presentation

1.1 Line of business

The LRDE (Research and Development Laboratory of EPITA) is focused on fundamental research and development in computer science. Its main areas of expertise are:

- Image processing and pattern recognition
- Automata and verification
- Performance and genericity

Building on its solid scientific production and academic collaborations, the laboratory has industrial contracts, conducts internal research projects and participates in collaborative academic research projects.
Its members also give classes to students at EPITA from the first year of engineering.

1.2 The Laboratory

The LRDE (`https://www.lrde.epita.fr/wiki/Home`) was created in February 1998 to promote the research activity at EPITA and to allow students to be involved into important research projects.

The research activity at LRDE is focusing on subjects related to the school with the aim of getting recognition in the scientific domain through publications and by working together with other research centers.
One particularity of the LRDE is the will to create a bond between traditional teaching given to EPITA students and teaching through research. The point of this is to:

- participate to the production of knowledge in computer science and to promote the image of EPITA in scientific domain.
- develop LRDE student's formation through research and allow them to access a third cycle formation.

1.3 Members

The laboratory is currently composed of thirteen permanent members, including teacher-researchers, engineers and administration.
In addition to permanent staff, the LRDE also hosts PhD students. Currently, there are five of them. During the whole duration of their doctoral studies, they work with two advisor researchers, one of the LRDE and one of another university (joint supervision in partnership).

Each year, the permanent members recruit third year students from EPITA, whom will stay until the end of their studies, following a dedicated study specialisation at EPITA . Hence, the laboratory hosts two generations of students that can grow to a number between ten to fifteen.

1.4 Services

The LRDE is working on four different axis:

1.4.1 Image Processing

Olena

The Olena project (`https://olena.lrde.epita.fr`) consists of a generic image processing library. Its objective is to implement a platform of numerical scientific computations dedicated to image processing, pattern recognition and computer vision. This environment is composed of a generic and efficient library (Milena), a set of tools for shell scripts and a visual programming interface. The project aims at offering an interpreted environment like MatLab or Mathematica. It provides many ready-to-use image data structures (regular 1D, 2D, 3D images, graph-based images, etc.) and algorithms. Milena's algorithms are built upon classical entities from the image processing field (images, points/sites, domains, neighborhoods, etc.).

Each of these parts imply its own difficulties and require the development of new solutions. For example, the library, which require the entirety of low level features on which it relies on to be both efficient and generic — two objectives that are hard to meet at the same time in programmation. Fortunately, the object oriented programming eases this problem if we avoid the classical object modeling with inheritance and polymorphism. Hence, this genericity allows the development of efficient and re-usable code - i.e. developers or practitioners can easily understand, modify, develop and extend new algorithms while retaining the core traits of Milena: genericity and efficiency. The Olena platform uses this paradigm. The project already addressed the problem of the diversity of data and data structures.

Furthermore, the people working on this project were able to put in light the existence of conception models related to generic programmation. Olena is an open source project under General Public License (GPL) version 2.

Climb

The Climb team of the laboratory has chosen to focus on the persistent question of performance and genericity, only from a different point of view.

The purpose of this research is to examine the solutions offered by languages other than C++, dynamic languages notably, and Lisp in particular. C++ has its drawbacks, it is a heavy language with an extremely complex and ambiguous syntax, the template system is actually a completely different language from standard C++ and finally it is a static language. This last point has significant implications on the application, insofar as it imposes a strict chain of Compilation → Development → Run → Debug, making for example rapid prototyping or human-machine interfacing activities difficult. It becomes therefore essential to equip the involved projects with a third language infrastructure that is rather based on scripting languages.

The Climb project aims at investigating the same domain as Olena, but starting from an opposite view. It express the same issues following an axis of dynamic genericity and compares the performance obtained by some Common Lisp compilers with those of equivalent programs written in C or C++.

1.4.2 Finite state machine manipulation

Vcsn

The Vcsn project (`https://vcsn.lrde.epita.fr`) is a finite state machine manipulation platform developed in collaboration with the ENST. Finite state machines, also called automata, are useful for language treatment and task automation. In the past, such platforms, like "FSM", were supposed to work for problems of industrial scale. Hence, for efficiency reasons, they were specialized in letter automata. On the other hand, platforms like "FSA" were based on a more abstract approach. Vcsn tries to answer both of these issues by using techniques of static and generic programmation in C++.

Vcsn can then support the entirety of automata with multiplicity in any kind of semiring. Thanks to generic programming techniques, it is not necessary to code a single algorithm once for each type of automata anymore. A single abstract version is sufficient, and this without loosing efficiency. It is not necessary to handle C++ perfectly to be able to use the platform thanks to an interpreter conceived to highlight all of the system's potential. This environment should allow researchers

to experiment their ideas and beginners to practice with an intuitive interface.

Vcsn is an open source project under GPL license.

1.4.3 Model checking

Spot

Spot (`https://spot.lrde.epita.fr/`) is a library of algorithms for "model checking", which is a way to check that every possible behavior of a system satisy its given properties. Spot allows to express those properties using linear-time temporal logic (LTL). It corresponds to classical propositional calculus (with its "or", "and" and "not" operators) equiped with temporal operators to express things such as "in a future time" or "anytime since now". Spot also supports arbitrary acceptance condition, transition-based acceptance and four different representation formats of ω-automata (HOA, never claims, LBTT, DSTAR). All those terms will be explained in the 'basic concepts' section.

Such formulas seen above (LTL formulas) can be translated to automata (Spot implements different algorithms) , such that verifying that the behavior of a model satisfy a formula can be reduced to operations between two automata (here again Spot implements different algorithms). This approach can be applied to different kind of systems: communication protocoles, electronic circuits, programs...

This project was born in the MoVe team at LIP6, but since 2007 it is mainly developped by the LRDE, with some occasional collaborations with LIP6. It is distributed under a GNU GPL version 3 license.

1.4.4 Speaker recognition

Speaker ID

The Speaker Recognition team is working on Machine Learning solutions applied to Speaker Recognition tasks. They propose statistical representations of speech signal which are more robust to the problem of session and channel variabilities.

A speaker must always be identified, whether he is ill, suffering from sore throats, or his current emotions bring change to his voice. To do this, all the characteristics of a voice that can change depending on any external parameter must be ignored. This is one of the issues the Speaker ID team is facing.

They participated in the evaluation campaign of speaker verification systems organized by NIST (the National Institute of Standards and Technology) which organizes competitions in various fields, both to stimulate research and to define new

standards since the beginning of the project.

The work of LRDE Speaker ID team is conducted in collaboration with the Spoken Language Systems Group of the MIT Computer Science and Artificial Intelligence Laboratory (`http://groups.csail.mit.edu/sls/`).

1.5 The internship in the company's work

This internship took place within the team of model checking. It was essentially focused on the improvement of the SAT-based minimization of ω-automata. It covers one of the many features of the Spot library.

Chapter 2

Spot

Spot was first presented in 2004 [9]. It was purely a library until Spot 1.0 [8], when command-line tools for LTL manipulation and translation of LTL to some generalizations of Büchi Automata have started to be distributed. Today, Spot 2.0 supports more tools with arbitrary acceptance conditions as described in the Hanoi Omega Automata format (HOA) [17] and python bindings usable in interactive environments such as IPython/Jupyter [11].

2.1 Structure

The Spot project can be broken down into several parts, as shown in Figure 2.1. Orange boxes are C/C++ libraries. Red boxes are command-line program. Blue boxes are Python-related.

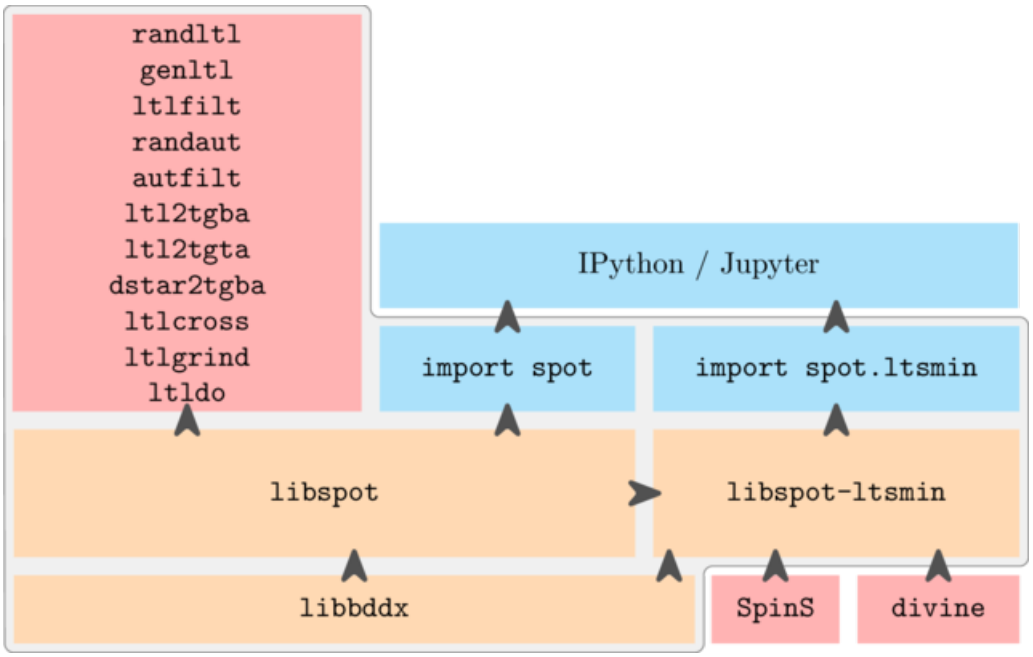

Figure 2.1: Architecture of Spot

Spot is actually split in three libraries:

- libbddx is a customized version of BuDDy for representing Binary Decision Diagrams which we use to label transitions in automata, and to implement a

few algorithms.

- libspot is the main library containing all data structures and algorithms.
- libspot-ltsmin contains code to interface with state-spaces generated as shared libraries by LTSmin.

2.2 Command-line tools

Spot 2.0 installs the following eleven command-line tools, that are designed to be combined as traditional Unix tools.

SPOT 1.0	randltl	Generates random LTL/PSL formulas
	genltl	Generates LTL formulas from scalable patterns
	ltlfilt	Filter, converts, and transforms LTL/PSL formulas
	ltl2tgba	Translates LTL/PSL formulas into generalized Büchi automata[1], or deterministic parity automata (new in 2.0)
	ltl2tgta	Translates LTL/PSL formulas into Testing automata[2]
	ltlcross	Cross-compares LTL/PSL-to-automata translators to find bugs (works with arbitrary acceptance conditions since Spot 2.0)
	ltlgrind	mutates LTL/PSL formulas to help reproduce bugs on smaller ones
	dstar2tgba	converts ltl2dstar automata into Generalized Büchi automata[16]
	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

Figure 2.2: Spot tools description

As you see, the first six tools were introduced in Spot 1.0 [8], and have since received several updates. The other tools were introduced after.

2.3 The Python Interface

Similar tasks can be performed in a more "algorithm-friendly" environment using the Python interface. Combined with the IPython/Jupyter notebook [11] (a web application for interactive programming), this provides a nice environment for experiments, where automata and formulas are automatically displayed.

2.4 Workflow

Working on any project of the LRDE implies to follow some rules. That allows a better integration of each member. Once a patch is ready, any member of the model checking team can re-read the patch and make suggestions.

2.4.1 Coding conventions

As Spot is a free software, uniformity of the code matters a lot. Some coding conventions are used so that the code looks homogeneous. Here are some points:

- UTF-8 is used for non-ASCII characters.
- tabs are not used for indentation in c++ files, only spaces, in order to prevent issues with people assuming different tab widths.
- #include with angle-brackets refers to public Spot headers (i.e those that will be installed, or system headers that are already installed).
- # include with double quotes refer to private headers that are distributed with Spot.
- ... (see `https://gitlab.lrde.epita.fr/spot/spot/blob/master/HACKING` for more details)

2.4.2 Git Versionning Tool

The versionning tool used in Spot is Git. All development branches except 'master' and 'next' follows a particular naming convention: {initials}/{subject of work}). This allows a quick glance to identify who works on which branch and on what.

Concerning commits, large commits introducing a feature are preferred to many small commits covering the same feature. Suppose that a new feature must be implemented and needs 3 key steps. Even if each step is done in many commits during the development, at the end, it's better to squash commits so as to have only 3 large commits representing the 3 key steps.

Also, if at any moment in turns out that a previous work could have been done otherwise, any update must be applied directly to the commit concerned - each commit must actually insert code in its final form.

2.4.3 Adding Tests

Any implementation done must be tested. For the purpose on one hand to avoid regression and on the other hand to ensure the code runs as expected. All tests are located in the 'tests' folders. Must of them are written in Python (using the python bindings) or shell script.

Chapter 3

Basic concepts

Spot essentially manipulates ω-automata which is a variation of finite-state automata (FSA) that runs on infinite, rather than finite. It is important to have a good understanding of automata theory, not only to understand this report but because automata crop up preety much everywhere in computer science. In logic design, natural language processing, system analysis, regular expressions, etc. The next two sections introduce the basics of automata theory and some concepts used in Spot.

3.1 Automata theory

Automata theory [3] is the study of abstract machines and automata, as well as the computational problems that can be solved using them.

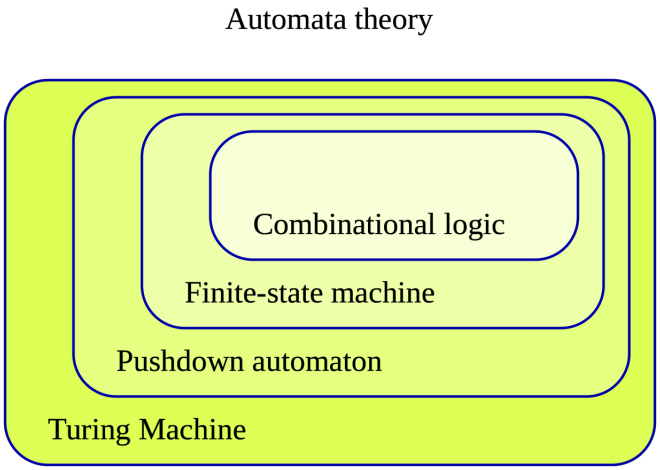

Figure 3.1: Classes of automata [3]

There are different classes of automata. The finite state machine has less computational power than some other models of computation such as the Turing machine [18]. The computational power distinction means there are computational tasks that a Turing machine can do but a finite-state automaton (FSA) can not.

From now on, the acronym FSA will be used instead of final-state automat{a, on}.

Only FSA will be concisely described in this section as (again) ω-automata used in Spot are a variation of FSA.

3.1.1 Automata

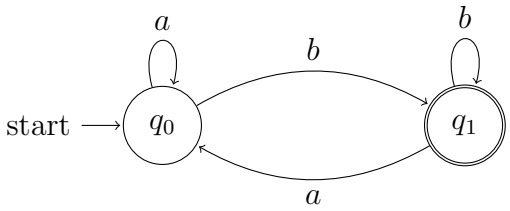

Figure 3.2: A finite-state automaton represented using a graph

The figure (3.2) shows how an automaton looks like. It consists of:

- a set of states (represented in the figure by circles). Among them we can distinguish the *initial state* **q0** (the one pointed by **start**) and an *accepting state* **q1** (the one doubly encircled).
- a transition relation (represented in the figure by arrows). This relation indicates for each state which state could be next according to the next symbol to read.

Before giving a regular definition, some points need clarification.

3.1.2 Alphabet, word, language

An alphabet is a finite set of symbols. It is commonly denoted by Σ. Edges of automata are labeled by those symbols. The alphabet used by the automaton of figure 3.2 is $\{a, b\}$.

A word is a finite sequence of symbols. *a*, *b*, *ab*, *aaaa*, *abbba*, *abababbbbaaabbaba* are some words defined on the alphabet $\{a, b\}$. Automata recognizes words. In Spot, ω-automata recognizes *ω-words*, which are infinite rather than finite.

A language is a set of words defined on the same alphabet. The automata of figure 3.2 accepts or recognizes the set of words ending with *b*. **q1** is the only one accepting state and the only way to come in **q1** is to read b.

3.1.3 Automaton run

Well, an automaton run can be described as follows:

- it starts in the initial state and waits for the first character (of the word) to read.
- At each step, it reads the next symbol (character) and determine the next state using the transition relation.
- It stops when all the character chain has been read. The word is accepted if the automaton is in one of the accepting states.

3.1.4 Determinism (DFA, NFA)

A *deterministic* FSA is a final state machine where for each pair of state and input (symbols) there is one and only one transition to a next state. The figure 3.2 introduced before is actually a *deterministic* FSA or *deterministic* finite-state automaton (DFA).

A *non deterministic* FSA or *Nondeterministic* finite-state automaton (NFA) allows:

- many transitions labeled by the same symbol and outgoing from the same state,
- transitions labeled by the *empty word* ε,
- transitions labeled by more than one symbol.

3.1.5 Finite-state automata (FSA) Definition

More formally, a finite-state automaton is defined by a quintuplet $M = (Q, \Sigma, X, s, F)$ where:

- Q is a set of states,
- Σ is an alphabet,
- X can be either a transition function $\delta : Q \times \Sigma \to Q$ (if the automaton is deterministic) or a transition relation $\Delta \subset (Q \times \Sigma^* \times Q)$ (if it is not deterministic),
- $s \in Q$ is the initial state,
- $F \subseteq Q$ is a set of accepting states.

3.2 About Spot

This section consists essentially of Spot's **concept** web page excerpts[6]. Feel free to have a look on that web page for further details.

3.2.1 Atomic preposition

An *atomic proposition* is a named Boolean variable that represents a simple property that must be true or false. It usually represents some property of a system. They are used to construct temporal logic formulas[13] to specify properties of the system.

3.2.2 Boolean formula

A Boolean formula is formed from *atomic preposition*, the Boolean constants true and false, and standard Boolean operators like and, or, implies, xor, etc.

3.2.3 ω-words

An ω-word as said before is a word of infinite length. In our context, each letter is used to describe the state of a system at a given time, and the sequence of letters shows the evolution of the system as the (discrete) time is incremented.

If the set **AP** of atomic propositions is fixed, an ω-word over **AP** is an infinite sequence of subsets of **AP**. In other words, there are $2^{|\mathbf{AP}|}$ possible letters to choose from, and these letters denote the set of atomic propositions that are true at a given instant.

For instance if **AP**$= \{a, b, c\}$, the infinite sequence $\{a, b\};\{a\};\{a, b\};\{a\};\ \{a, b\};\{a\};\ldots$ is an example of ω-word over **AP**. This particular ω-word can be interpreted as the following scenario: atomic proposition a is always true, b is true at each other instant, and c is always false.

3.2.4 ω-automaton

An ω-automaton is used to represent sets of ω-word.

Those look like the classical NFA in the sense that they also have states and transitions. However ω-automata recognize ω-words instead of finite words. In this context, the notion of final state makes no sense, and is replaced by the notion of acceptance condition: a run of the automaton (i.e., an infinite sequence alternating states and edges in a way that is compatible with the structure of the automaton) is accepting if it satisfies the constraint given by the acceptance condition.

In Spot, ω-automata have their edges labeled by Boolean formulas. An ω-word is accepted by an ω-automaton if there exists an accepting run whose labels (those Boolean formulas) are compatible with the minterms [14] used as letters in the word.

The language of an ω-automaton is the set of ω-words it accepts.

There are many kinds of ω-automata and they mostly differ by their acceptance condition. The different types of acceptance condition, and whether the automata are deterministic or not can affect their expressive power.

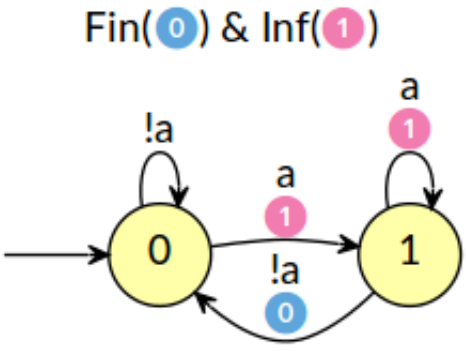

Figure 3.3: ω-automaton representing formula **FGa**

3.2.5 Acceptance condition

An acceptance condition actually consists of two pieces: some acceptance sets, and a formula that tells how to use these acceptance sets.

Acceptance formulas are positive Boolean formula over atoms of the form t, f, $Inf(n)$, or $Fin(n)$, where n is a non-negative integer denoting an acceptance set.

- **t** denotes the true acceptance condition: any run is accepting
- **f** denotes the false acceptance condition: no run is accepting
- **Inf(n)** means that a run is accepting if it visits infinitely often the acceptance set n
- **Fin(n)** means that a run is accepting if it visits finitely often the acceptance set n

The above atoms can be combined using only the operator & and | (also known as $\wedge$ and $\vee$), and parentheses for grouping. Note that there is no negation, but an acceptance condition can be negated swapping t and f, $\wedge$ and $\vee$, and $Fin(n)$ and $Inf(n)$.

The following table gives an overview of how some classical acceptance condition are encoded. The first column gives a name that is more human readable (those names are defined in the HOA [12] format and are also recognized by Spot). The second column give the encoding as a formula. Everything here is case-sensitive.

none	`f`
all	`t`
Buchi	`Inf(0)`
generalized-Buchi 2	`Inf(0)&Inf(1)`
generalized-Buchi 3	`Inf(0)&Inf(1)&Inf(2)`
co-Buchi	`Fin(0)`
generalized-co-Buchi 2	`Fin(0)|Fin(1)`
generalized-co-Buchi 3	`Fin(0)|Fin(1)|Fin(2)`
Rabin 1	`Fin(0) & Inf(1)`
Rabin 2	`(Fin(0) & Inf(1))|(Fin(2) & Inf(3))`
Rabin 3	`(Fin(0) & Inf(1))|(Fin(2) & Inf(3))|(Fin(4) & Inf(5))`
Streett 1	`Fin(0)|Inf(1)`
Streett 2	`(Fin(0)|Inf(1)) & (Fin(2)|Inf(3))`
Streett 3	`(Fin(0)|Inf(1)) & (Fin(2)|Inf(3)) & (Fin(4)|Inf(5))`
generalized-Rabin 3 1 0 2	`(Fin(0) & Inf(1))|Fin(2)|(Fin(3) & (Inf(4)&Inf(5)))`
parity min odd 5	`Fin(0) & (Inf(1)|(Fin(2) & (Inf(3)|Fin(4))))`
parity max even 5	`Inf(4)|(Fin(3) & (Inf(2)|(Fin(1) & Inf(0))))`

Figure 3.4: ω-automata acceptance conditions [6]

3.3 SAT solver

Satisfiability problem is a classic of computer science.

The purpose of SAT solving is to assign each variables of a propositional formula in such a way that the formula evaluates to true. It is the canonical NP-complete problem. SAT solvers are used to solve many practical problems and this is also the case in Spot, they are used to minimize ω-automata.

For more details, **SAT-solving in practice** [15] is a good introduction to SAT solvers.

Chapter 4

Completed work

4.1 Specifications

This section tries to shed light on what was before this internship and why all the achieved work was needed.

4.1.1 Overall goal

The principal purpose of this internship is to improve an algorithm already implemented and used to minimize *deterministic* ω-automaton. The source code associated is the result of two papers:

- **SAT-based Minimization of Deterministic ω-Automata**[5] and
- **Mechanizing the Minimization of Deterministic Generalized Büchi Automata**[4].

Those two papers written by *Souheib baarir* and *Alexandre Duret-Lutz* are themselves a generalization of **Ehlers** SAT' based procedure [10]. Note that the first paper[5] is an extension of the second[4] which is restricted to generalized-Büchi acceptance.

The existing minimization

These previous work introduced a tool that can read any *deterministic* ω-automaton and synthetize (if it exists) an equivalent *deterministic* ω-automaton with a given number of states and arbitrary acceptance condition.

This tool, called SynthetizeDTGBA(R, n, m) works this way, It:

- inputs a complete DTGBA R, two target numbers of state (n) and arbitrary acceptance condition (m),
- produces a DIMACS file [7] with all the above clauses,
- calls a SAT solver to solve this problem,
- builds the resulting DTGBA C if it exists.

Using this tool, two minimization algorithms have been implemented:

Algorithm 1 A naive algorithm that calls SynthetizeDTGBA(R, n, m) in a loop, with a decreasing number of states, and returns the last successfully built automaton.

```
1: procedure ReduceStatesDTGBA(R, m = R.nb_acc_sets())
2: repeat:
3:     n ← R.nb_states()
4:     C ← SynthetizeDTGBA(R, n − 1, m)
5:     if C does not exists then return R
6:     R ← C
```

Algorithm 2 This also calls SynthetizeDTGBA(R, n, m) in a loop, but attempting to find the minimum number of states using a binary search.

```
1: procedure DichotomyDTGBA(R, m = R.nb_acc_sets())
2:     max_states ← R.nb_states() − 1
3:     min_states ← 1
4:     S ← null
5:     while min_states <= max_states do
6:         target ← (max_states + min_states)/2
7:         C ← SynthetizeDTGBA(R, target, m)
8:         if C does not exists then
9:             min_states ← target + 1
10:        else
11:            S ← C
12:            max_states ← R.nb_states() − 1
13:    R ← S
```

Until this internship, Algorithm 1 was used by default. There are no real reasons for that except that the second algorithm was implemented later in a completely different context and has never been benchmarked and compared to the first one.

ω-Automata minimization can be seen in two ways:

- **Reduction of the number of states:** This is typically the algorithm 1 that keeps by default the same number of acceptance sets (m) and decreases n at each SynthetizeDTGBA(R, n, m) call. The algorithm 2 has also the same perception. It knows the minimal automaton is between 1 (obviously, a smaller one does not exists) and $n - 1$ so instead of checking each number of states it performs a binary search with the will to be faster.

- **Rise of the accepting sets number:** This can be interpreted as the converse of a degeneralization: instead of augmenting the number of states to reduce the number of acceptance sets, we augment the number of acceptance sets in an attempt to reduce the number of states.

The figure below (4.1) is a great example from the first paper [4] that shows how smaller an automaton can become if the acceptance sets number is increased. Note that $|\mathcal{F}|$ here is m (the number of accepting sets).

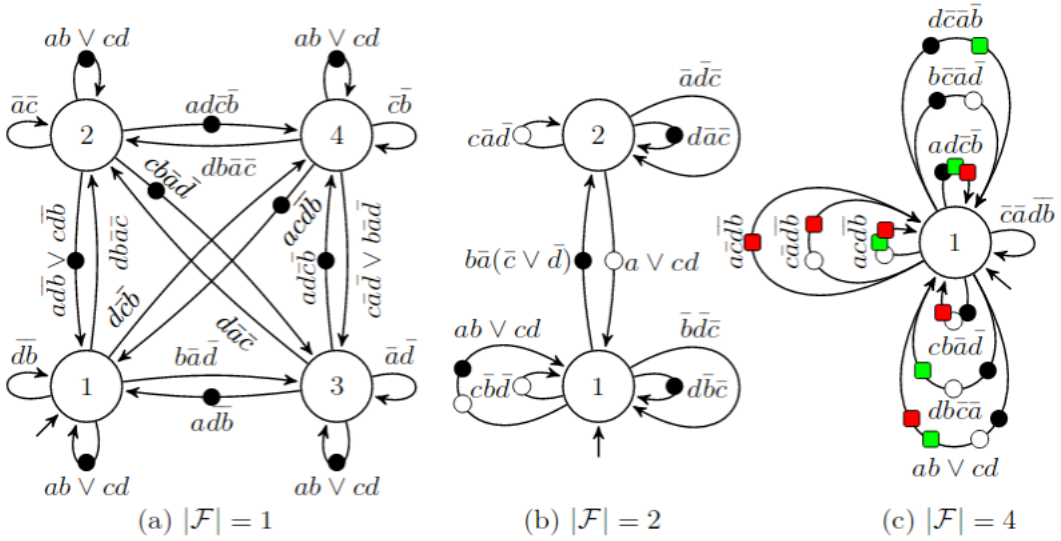

Figure 4.1: Examples of minimal DTGBA recognizing $(GFa \wedge GFb) \vee (GFc \wedge GFd)$

Finding the smallest m such that no smaller equivalent DTGBA with a larger m can be found is still an opened problem.

Tool chain

The figure 4.2 (from the FORTE'14 paper [4]) gives an overview of the processing chains that can be used to turn an LTL formula [13] into minimal DBA, DTBA or DTGBA. The blue area at the top describes:

Listing 4.1: bash command-line to translate a formula using ltl2tgba

```
ltl2tgba -D -x sat-minimize
```

while the purple area at the bottom corresponds to:

Listing 4.2: bash command-line to translate a formula using dstar2tgba

```
dstar2tgba -D -x stat-minimize
```

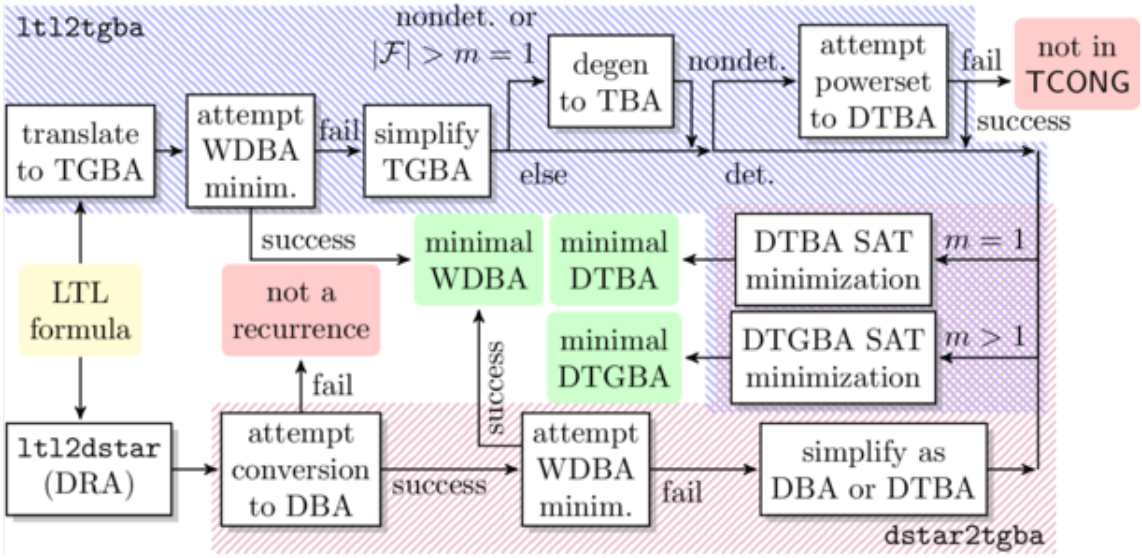

Figure 4.2: Two tool chain used to convert an LTL formula

As the SAT-based minimization only takes DTBA or DTGBA, the input automaton undergoes some transformations. In each tool, a Weak-DBA minimization is attempted. If that succeed, a minimal weak DBA is outputed (looking for transition-based or generalized acceptance will not reduce it further). For further details, please read the FORTE'14 paper [4].

The limits

Consider again the default algorithm (1). At each iteration, this algorithm re-encodes the research of a smaller automaton from scratch. Iterations after iterations (from n to $n-1$, $n-1$ to $n-2$, etc.) the clauses encoded are almost the same. Therefore, it is a shame that nothing is retained, learned at each iteration. This is where incremental sat solving comes in mind. Incremental solving is one of the recent directions in SAT solver research. This is based on an observation that in many applications of SAT solvers, the problems being solved consist of several calls to SAT solver on a sequence of SAT problem. Typically, the problems in the sequence share a large common part, making them highly interrelated. This is exactly our case! The purpose of incremental SAT solving is to recycle the work done in solving a previous problem in the sequence to solve the subsequent problems.

In order to use an incremental approach with SAT solver, it is no more possible to use DIMACS file [7]. Therefore, Spot needs to be linked to a SAT solving library. Let's remember that until now Spot requires a SAT solver to be installed on the same machine and provides a way to set it (through SPOT_SATSOLVER environment variable). obviously, being linked to a SAT solver and make calls to its functions will be more efficient than making Input/Output operations and executing another binary.

The memory conssumption is also another problem. The larger the automaton is, the more variables there are. With some automata, the memory usage could grow over 150 Go which is uncommon for most users. The figure 4.3 helps to see memory usage peaks during a benchmark realized the end of September 2016.

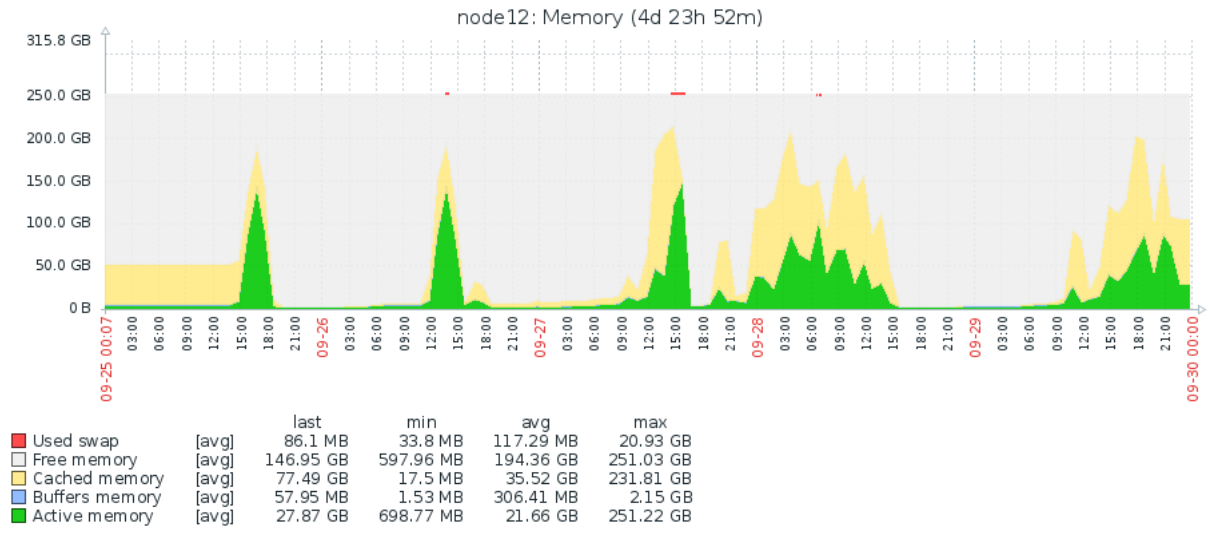

Figure 4.3: A benchmark realized at the beginning of the internship

4.1.2 Detailed explanation of the results to be obtained

4.2 Activity report

4.2.1 Selected areas of study and research

4.2.2 Conduct of studies

4.3 Interpretation and critique of results

Chapter 5

Bibliography and glossary

References

[1] A.Duret-Lutz. "LTL translation improvements in Spot 1.0". In: *Int. J. on Critical Computer-Based Systems* (2014).

[2] A.Duret-Lutz A.E.Ben Salem and F.Kordon. "Model checking using generalized testing automata". In: *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC V1)* (2012).

[3] *Automata Theory*. URL: `https://en.wikipedia.org/wiki/Automata_theory`.

[4] Souheib Baarir and A.Duret-Lutz. "Mechanizing the Minimization of Deterministic Generalized Büchi Automata". In: *FORTE* (2014).

[5] Souheib Baarir and A.Duret-Lutz. "SAT-based Minimization of Deterministic omega-Automata". In: *LPAR* (2015).

[6] *Concepts*. URL: `https://spot.lrde.epita.fr/concepts.html`.

[7] *DIMACS format*. URL: `http://www.satcompetition.org/2009/format-benchmarks2009.html`.

[8] A. Duret-Lutz. "Manipulating LTL formulas using Spot 1.0". In: *ATVA* vol. 8172.13 (Springer, 2013), pp. 442–445.

[9] A. Duret-Lutz and D.Poitrenaud. "SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata". In: *MASCOTS* 04 (2004), pp. 76–83.

[10] Rüdiger Ehlers. *Minimising deterministic Büchi automata precisely using SAT solving*. 2010.

[11] F.Pérez and B.E.Granger. "IPython: a system for interactive scientific computing". In: *Computing in Science and Engineering* (21-29, 2007). URL: `http://ipython.org/`.

[12] *HOA format*. URL: `http://adl.github.io/hoaf/`.

[13] *Linear-time Temporal Logic (LTL)*. URL: `https://spot.lrde.epita.fr/concepts.html#ltl`.

[14] *Minterms*. URL: `https://en.wikipedia.org/wiki/Canonical_normal_form#Minterms`.

[15] *SAT-solving in practice*. URL: http://www.cse.chalmers.se/edu/year/2012/course/TDA956/Papers/satFinal.pdf.

[16] S.Baarir and A.Duret-Lutz. "Mechanizing the minimization of deterministic generalized Büchi automata". In: *FORTE* (Springer, 2014).

[17] A.Duret-Lutz J.Klein J.Kretinskiy D.Müller D.Parker T.Babiak F.Blahoudek and J.Strejcek. "The Hanoi Omega-Automata format". In: *CAV* vol. 9206.15 (Springer, 2015). URL: http://adl.github.io/hoaf/.

[18] *Turing Machine*. URL: https://en.wikipedia.org/wiki/Turing_machine.

Part II

Appendix

Contents (Appendix)

Appendix A

Company documentation

Appendix B

Hardware / Software Documentation

Appendix C

Gross results