Two-automaton accepting run search in Spot

Clément Gillard
(supervisor: Alexandre Duret-Lutz)

Technical Report n^o1805, June 2018
revision e2a7a075

In a previous paper we introduced the two-automaton emptiness check in the Spot library, that would check for the emptiness of the intersection of the languages of two ω-automata without building their product, circumventing a limitation in Spot on the number of acceptance sets of the automata.
This operation, when the intersection is actually not empty, is usually followed by an accepting run search on the product that would return an example of such a word that is in the languages of both automata.
We introduce a new method that computes an accepting run from the data gathered during the two-automaton emptiness check, allowing us to use this new method in algorithms that require the counterexample.

Dans un précédent rapport nous présentions le test de vacuité bi-bande dans la bibliothèque Spot, qui vérifie que l'intersection des langages de deux ω-automates est vide sans construire leur produit, ce qui permet de contourner une limitation de Spot sur le nombre d'ensembles d'acceptations de ces automates.
Cette opération, lorsque l'intersection n'est effectivement pas vide, est généralement suivie d'une recherche de chemin acceptant sur le produit qui expose un mot qui se trouve dans les langages des deux automates.
Nous présentons une nouvelle fonction qui réalise cette recherche de chemin acceptant à partir des données générées par le test de vacuité bi-bande, ce qui nous permet d'utiliser ces nouvelles méthodes dans des algorithmes qui nécessitent la recherche d'un contre-exemple.

Keywords
Spot, emptiness check

Laboratoire de Recherche et Développement de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
clement.gillard@epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Chapter 1

Introduction

Model checking is the field of computer science interested in the exhaustive and automatic verification of the behaviours of a model: from the model M of a system and a set of specifications, we want to check that the system meets all requirements. The automata-theoretic approach to model checking is a classical way to verify a system, that can often be reduced to graph problems, and to which we can apply ω-automata theory, a field of automata theory that considers ω-automata that recognise infinite words, just like a system is designed to execute on infinitely long inputs.

The approach, illustrated in Fig. 1.1, goes as follows: we explore a model M to build an ω-automaton A_M, and we express a specification we want to check as a logical property φ, usually using Linear Temporal Logic (Pnueli, 1977), that we can then negate and translate into an ω-automaton $A_{\neg\varphi}$: A_M accepts all words that represent behaviours of the model, and $A_{\neg\varphi}$ accepts all words that do not satisfy φ. We then check if the languages of these ω-automata have an intersection, that is to say, if there is a word that represents a valid behaviour of the model but does not verify the property. This is usually done by computing the product of the ω-automata, whose language is the intersection of its operands, and then running an *emptiness check* algorithm on that product to check if its language is empty. If it is not, then the property is violated, and we usually want a counterexample of what went wrong: this is done by running an *accepting run search* on the product to extract a single word that exists both in the languages of the model and the negation of property.

Spot 2 (Duret-Lutz et al., 2016) is a `C++` library whose goal is to provide tools to manipulate ω-automata. It implements several algorithms, allows for user-defined automata, provides a Python interface, and comes with several binaries to generate, translate, and process LTL formulae and ω-automata. It also ships with two binaries to compare such tools: `ltlcross` and `autcross`, that take as inputs LTL formulae (respectively ω-automata) and several tools that should translate (respectively process) them equivalently, and semantically compare their outputs. Both `ltlcross` and `autcross`, among other things, compare the ω-automata by running an emptiness check between an automaton and what should be its complement: the intersection of their languages should be empty. `ltlcross` is used by some LTL translators' development teams to test their tool and compare it to others' (Duret-Lutz, 2017, p. 52).

In Spot, ω-automata have a fixed maximum number of acceptance sets they can have. But the number of acceptance sets of the product of two ω-automata is the sum of the numbers of

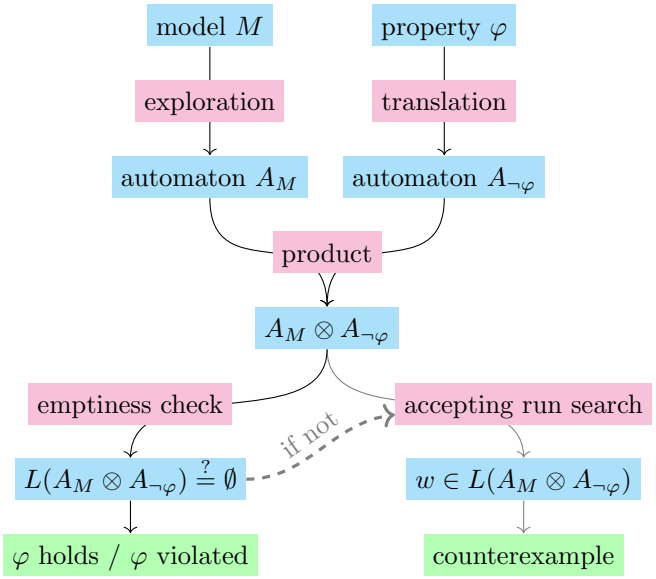

Figure 1.1 – The usual ω-automata approach to model checking

acceptance sets of its operands. That means that even though two ω-automata may be expressible in Spot, their product may not, which prevents the application of a regular emptiness check algorithm on the product, or any other algorithm that is to be run on the product.

In Gillard (2017), we introduced the *two-automaton emptiness check*, which takes two ω-automata and simulates their product while running an emptiness check algorithm, essentially giving the result of a regular product then emptiness check while circumventing the limitation in acceptance sets. This algorithm, while being fully usable, did not change a lot of the workflow of Spot's tools, because of its binary answer: most algorithms do not only need to know if the languages have an intersection, they also need to have a counterexample if there is one.

We are now introducing the *two-automaton accepting run search*, that looks for an accepting run in the data that was generated by the two-automaton emptiness check. With these two algorithms, we can now replace every occurrence of a product followed by an emptiness check and still be able to run an accepting run search when needed, while still not having to compute an actual product of automata.

Chapter 2

Definitions and notations

2.1 Mathematical notations

This section defines various mathematical notations used later in this report.

Definition 2.1 (Power set) *The power set of a set A, denoted $\mathcal{P}(A)$, is the set of all subsets of A, such that we have:*

$$B \subseteq A \iff B \in \mathcal{P}(A)$$

Definition 2.2 (Set of integers) *For two integer numbers a and b, we denote as $[a..b]$ the set of all integers between a and b:*

$$[a..b] = \mathbb{Z} \cap [a, b]$$

Definition 2.3 (Set of strictly positive integers) *For a strictly positive integer number a, we denote as $[a]$ the set $[1..a]$.*

2.2 Notions on ω-automata

This section introduces various definitions on ω-automata that are necessary for the comprehension of this report.

Definition 2.4 (Acceptance set) *An acceptance set is a set of transitions or states. A single transition or state can belong to any number of acceptance sets.*

Transition-based ω-automata (see Definition 2.8), as they are handled in Spot, are ω-automata whose acceptance sets are sets of transitions.

Definition 2.5 (Acceptance mark) *An acceptance mark is a mark associated with an acceptance set. Marking a transition or state denotes its belonging to an acceptance set.*

Acceptance marks allows for a more graphical way to represent acceptance sets, by showing the marks on their associated transitions or states, as can be seen in Fig. 2.1.

Definition 2.6 (Acceptance condition) *An acceptance condition ϕ of an ω-automaton is a Boolean formula which respects the following grammar:*

$$\phi := \top \mid \bot \mid Inf(x) \mid Fin(x) \mid Inf(\overline{x}) \mid Fin(\overline{x}) \mid \phi \wedge \phi \mid \phi \vee \phi \mid (\phi)$$

where x denotes an acceptance set.

As can be seen in Fig. 2.1, for clarity we can use an acceptance mark to denote an acceptance set in the acceptance condition.

Definition 2.7 (Generalised Büchi acceptance condition) *A generalised Büchi acceptance condition is an acceptance condition which does not make use of the Fin operator. It is said to be Fin-less.*

Currently, the two-automaton emptiness check and accepting run search algorithms are only able to work on ω-automata with generalised Büchi acceptance conditions.

Definition 2.8 (Transition-based ω-automaton) *A transition-based ω-automaton is a tuple $A = \langle \Sigma, Q, q_0, \Delta, n, m, \phi \rangle$ where*

- *Σ is a finite alphabet,*
- *Q is a finite set of states,*
- *$q_0 \in Q$ is the initial state,*
- *$\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation,*
- *$n \in \mathbb{N}$ is the number of acceptance sets,*
- *$m : \Delta \mapsto \mathcal{P}([n])$ is a function returning the acceptance marks of a transition,*
- *ϕ is an acceptance condition.*

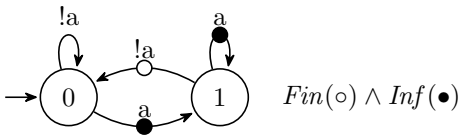

Figure 2.1 – The deterministic ω-automaton of the translation of the LTL formula `FG(a)`, as would be represented in Spot, with its acceptance condition on the right, and the acceptance marks on its transitions.

Definition 2.9 (Run, word, and step) *A run of an ω-automaton is an infinite sequence of consecutive transitions $\rho \in \Delta^\omega$ starting from the initial state of the automaton.*

We say that the run $\rho = (q_0, \ell_0, s_0)(s_0, \ell_1, s_1)(s_1, \ell_2, s_2)\ldots$ recognises the word $w = \ell_0\ell_1\ell_2\ldots \in \Sigma^\omega$.

A single transition $(s, \ell, d) \in \Delta$ of a run is called a step.

Definition 2.10 (Accepting run) *Let $m_{Inf} \subseteq [n]$ be the set of marks that we see infinitely often along an infinite subsequence of transitions σ of a run ρ of an ω-automaton. We write $\sigma \vDash \phi$ to mean σ satisfies ϕ and $\sigma \nvDash \phi$ to mean σ does not satisfy ϕ. The satisfaction of an acceptance condition is interpreted by induction as follows:*

$$
\begin{array}{lcll}
\sigma & \vDash & \top & \\
\sigma & \nvDash & \bot & \\
\sigma & \vDash & \mathit{Inf}(x) & \iff x \in m_{Inf} \\
\sigma & \vDash & \mathit{Fin}(x) & \iff x \notin m_{Inf} \\
\sigma & \vDash & \mathit{Inf}(\bar{x}) & \iff m_{Inf} \cap \{x\} = \emptyset \\
\sigma & \vDash & \mathit{Fin}(\bar{x}) & \iff m_{Inf} \cap \{x\} \neq \emptyset \\
\sigma & \vDash & \phi_1 \wedge \phi_2 & \iff \sigma \vDash \phi_1 \text{ and } \sigma \vDash \phi_2 \\
\sigma & \vDash & \phi_1 \vee \phi_2 & \iff \sigma \vDash \phi_1 \text{ or } \sigma \vDash \phi_2
\end{array}
$$

A run ρ of an automaton is an accepting run if and only if it contains such a σ that satisfies the acceptance condition of the automaton.

Definition 2.11 (Accepting word) *An accepting word is a word recognised by an accepting run.*

Definition 2.12 (Language) *The language of an automaton A, denoted $L(A)$, is the set of all accepting words of A.*

Definition 2.13 (SCC) *A Strongly Connected Component is a set of states of an ω-automaton where every state can be reached from any other state.*

2.3 Types of ω-automata in Spot

Spot implements two kinds of ω-automata:

On-the-fly automata whose states and transitions are only accessible through functions, which means that their exploration leads to code execution that could be reading memory, files, or even generating the automaton as requested. This is useful to represent some models, but is ultimately slower and uses more memory.

Explicit automata whose states and transitions are known in advance and are stored efficiently in memory. This leads to faster exploration and less memory usage.

Explicit automata offer an on-the-fly interface, which means they can be mixed transparently with on-the-fly automata, but that interface is slower that their actual interface, which is why some algorithms are rewritten or adapted to work with the explicit interface.

2.4 ω-automata operations in Spot

This section introduces the operations on ω-automata already implemented in Spot, that we aim to improve in this work.

It must be noted that the emptiness check algorithm introduced by Couvreur (1999) only runs on ω-automata with Fin-less acceptance condition.

2.4.1 Unary operations

`is_empty`: ω-automaton $\mapsto$ Boolean

Return `True` if the language of the automaton is empty, `False` otherwise.

Current implementation: If the automaton's acceptance condition is Fin-less, redirect to an implementation of Couvreur's emptiness check algorithm, else redirect to a generic emptiness check algorithm on an explicited automaton.

`accepting_run`: ω-automaton $\mapsto$ run

Return an accepting run if the language of the automaton is not empty, `nullptr` otherwise.

Current implementation: Run Couvreur's emptiness check algorithm, if the result is non-empty run its accepting run search algorithm, else return `nullptr`. If the automaton's acceptance condition is not Fin-less, abort with an error message.

`accepting_word` : ω-automaton $\mapsto$ word

Return an accepting word if the language of the automaton is not empty, `nullptr` otherwise.

Current implementation: Run `accepting_run` on a *Fin*-less copy of the automaton, and build a word from the result, or `nullptr` if the result is `nullptr`.

2.4.2 Binary operations

`product`: explicit ω-automaton $\times$ explicit ω-automaton $\mapsto$ explicit ω-automaton

Sometimes referred to as *explicit product* for clarity. Compute the synchronised product of the two given explicit automata as an explicit automata.

Requires two explicit automata, as well as a full exploration of the product.

`otf_product`: ω-automaton $\times$ ω-automaton $\mapsto$ on-the-fly ω-automaton

Sometimes referred to as *on-the-fly product* for ease of reading. Produce an on-the-fly automaton whose methods will compute the synchronised product on demand.

Uses the on-the-fly interface, so works with any type of automata, and does not require full exploration of the product.

`intersects`: ω-automaton $\times$ ω-automaton $\mapsto$ Boolean

Return `True` if the languages of the two automata have a non-empty intersection, `False` otherwise.

Current implementation: Perform various checks to ensure the suitability of an explicit product followed by a generic emptiness check of the product, otherwise build an on-the-fly product of *Fin*-less copies of the automata, and invoke `is_empty` on it.

`intersecting_run`: ω-automaton $\times$ ω-automaton $\mapsto$ run

Return an accepting run of the first automaton whose word is also an accepting word of the second automaton, or `nullptr` if the languages of the automata do not have an intersection.

Current implementation: Build an on-the-fly product of *Fin*-less copies of the automata, and invoke `accepting_run` on it.

`intersecting_word`: ω-automaton $\times$ ω-automaton $\mapsto$ word

Return a word that is in the languages of both automata, or `nullptr` if the languages do not have an intersection.

Current implementation: Build an on-the-fly product of *Fin*-less copies of the automata, and invoke `accepting_word` on it.

Chapter 3

Context of the two-automaton accepting run search

The goal of the two-automaton series of algorithms is to remove the limitation on the number of acceptance sets introduced by the product of automata in the `intersects`, `intersecting_run`, and `intersecting_word` methods, by reimplementing their underlying algorithms without building a product in Spot's constructs. This is done by handling pairs of states, iterators, and acceptance marks to simulate the operations that would be applied if they were part of an actual product automaton.

The two-automaton accepting run search completes the two-automaton emptiness check introduced by Gillard (2017), an reimplementation of the on-the-fly emptiness check algorithm introduced by Couvreur (1999). The two-automaton emptiness check was heavily templated to account for various optimisations, based on the implementations of the on-the-fly product and the emptiness check that already existed in Spot.

3.1 Reminders on the two-automaton emptiness check

The two-automaton emptiness check is an implementation of Couvreur's algorithm, which works with generalised Büchi ω-automata, automata whose acceptance condition is a conjunction of *Inf* operators.

This algorithm does a depth-first search through the automaton, looking for SCCs. During the exploration, the states are associated with an *order number*, a unique number representing the order in which states have been discovered, such that states discovered later are given a greater *order*. The algorithm keeps track of what acceptance marks can be seen in an SCC; if at some point an SCC contains marks that, if seen infinitely often, satisfy the acceptance condition of the automata, then it can be concluded that there exists a run capable of reaching that SCC and able to satisfy the acceptance condition, so the language of the automaton is not empty. If the depth-first search finished without finding any accepting SCC, than it can be concluded that there is no accepting run, and that the language of the automaton is empty.

This algorithm is an on-the-fly emptiness check : it does not need to explore the entire automaton to conclude that its language is not empty; however all transitions must be explored to conclude that the language is empty.

3.2 The data available

The two-automaton emptiness check produces several data structures:

- `states`, a hashmap that maps discovered pairs of states ("`product_states`") to their *order*,
- `todo`, a stack of pairs of iterators used for the depth-first search,
- `live`, a stack of the states currently in the search stack, used to mark those states as *dead* when they are known to neither belong to an accepting SCC nor lead to one,
- `acc`, a structure that holds the accepting marks of the current SCC and the order of the first discovered state in it; when the emptiness check ends this contains the minimal order of the states of the accepting SCC and the marks that had to be discovered in order to satisfy the acceptance condition.

We could use `live` or `todo` to build the accepting run: they contain the states that were seen during the depth-first traversal, that we could use as a trail of crumbs from the initial state to the transition that contained the last acceptance mark needed to satisfy the acceptance condition.

However, the transition iterators in `todo` are already pointing to the transition *after* the one we need to take in the accepting run, so using `todo` would require a transition search algorithm; same goes for `live`, which contains no information on the transitions taken. Also, these structures represent the depth-first search, that may not give an efficient run, which is why we decided to not use them and reimplement the run search from scratch.

This allowed following the existing implementation of the accepting run search, that only made use of `states` and `acc`, and used breadth-first searches to build the run.

3.3 Structure of an accepting run

An accepting run is always linked to an automaton.

Since a run is infinite, but runs through a finite automaton, we can use the pumping lemma to conclude that the run must contain a finite sequence of consecutive transitions that repeats itself infinitely in the run, the *cycle*, and a finite, possibly empty sequence of consecutive transitions that lead from the initial state to the first source state of the cycle, the *prefix*. In an accepting run, the cycle corresponds to σ in Definition 2.10.

An accepting run is therefore implemented as two finite sets of *steps*.

3.4 The search algorithm

The accepting run search is done in 3 steps:

1. Search for a path from the initial state to any state of the SCC. We denote as current state and also entry state the first encountered state of the SCC.

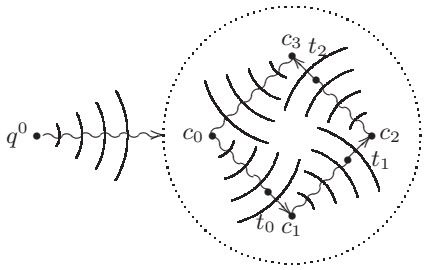

Fig. 5. Computing an accepting run for a TGBA.

Figure 3.1 – Excerpt from Couvreur et al. (2005)

2. From the current state, look for a path to a transition bearing a mark we have not seen yet while staying in the SCC; note the mark as seen, and the destination state of the transition as the current state. Repeat this step until the seen marks satisfy the acceptance condition.
3. From the current state, look for a path going back to the entry state while staying in the SCC.

The transitions of the path found in Step 1 make up the prefix, the transitions of the paths found in Steps 2 and 3 make up the cycle. Figure 3.1 is an illustration of those steps:

- the search from q^0 to c_0 is Step 1,
- the searches from $c0$ to $c1$, $c1$ to $c2$, and $c2$ to $c3$, each respectively seeing the marks t_0, t_1, and t_2, are Step 2,
- the search from $c3$ back to $c0$ is Step 3.

3.5 Existing implementation

As for the two-automaton emptiness check, the two-automaton accepting run search is heavily inspired from the existing single-automaton algorithms. The algorithm we chose to reproduce was one implemented along with the algorithm that inspired the two-automaton emptiness check: an accepting run search that uses back data from an implementation of Couvreur's algorithm. This algorithm, to get shortest paths, uses a breadth-first search for all 3 steps described in Section 3.4, constrained to the states that have been explored during the emptiness check.

The breadth-first search is done with the help of an existing Spot algorithm, `bfs_steps`, which from a starting state, and matching and filtering functions, builds a set of `steps` and appends it to a given set. The matching and filtering functions are user defined, which allow for a very flexible algorithm.

Chapter 4

Implementation of the two-automaton accepting run search

The two-automaton accepting run search is for the biggest part a rewrite of the `bfs_steps` algorithm for the "`product`" data structures of the two-automaton emptiness check. This chapter discusses its implementation and its integration within the two-automaton series of algorithms.

4.1 Spot's data structures

The following structures are used in the two-automaton series of algorithms:

state is a state of an automaton, it provides a field *successors* which is a transition iterator to its first outbound transition, from which one can iterate over all the outbound transitions,

`mark_t` is a set of acceptance marks, implemented as a bitfield, with all usual bitwise operations defined,

step contrarily to Definition 2.9, a step is implemented as a tuple $\langle s, \ell, m \rangle$ where s is the source state, ℓ is the condition, and m is the set of marks seen on the transition; this is because we can retrieve the destination states from the first two, and the marks are to be easily accessible for other algorithms that will not be discussed here.

4.2 Product data structures

The following data structures were implemented for Gillard (2017):

`product_state` is a pair of states, it provides a hash function for use in hash maps and an equality operator,

`product_mark` is a pair of `mark_t`s,

`product_iterator` is a pair of transition iterators, with a field indicating if it still has successors, and a field giving the next successor if there is one.

The members of these structures are called *left* and *right*, for elements respectively from the left and right automata.

Along with these, we also define the `product_steps` data structure, which is a tuple $\langle src, condition, marks \rangle$ where src is a `product_state`, $condition$ is a pair of conditions[1], and $marks$ is a `product_mark`.

4.3 Pseudocode

4.3.1 `product_bfs_steps`

Algorithm 4.1 shows the pseudocode for the `product_bfs_steps` algorithm. This algorithm will run a breadth-first search from a given start to a state indicated by the function *match*, while filtering out unexplored states, states marked as dead, and states indicated by the function *filter*.

It works by building a map *backlinks* which associates each unfiltered state with the first `product_step` that lead to it. From that, it can, once a match is found, reconstruct the run that goes from *start* to the match by getting the step that lead to the match, then the step that lead to the source of that step, and so on until the source is actually *start*; this is what is done lines 16 to 23. However, since the steps retrieve the steps backwards, we need to store them in a stack, and then pop them back to the actual runs ; this is done lines 24 to 31.

If a match is found, we return it, so that the two-automaton accepting run search may start a new breadth-first search from it.

4.3.2 Two-automaton accepting run search

Algorithm 4.2 shows the pseudocode for the actual implementation of the accepting run search. We start by setting up two match functions:

- *matchOrder* which will match on the state whose order is the one in *acc* (Section 3.2), which is the entry state of the accepting SCC we are looking for,
- *matchMarks* which will match on any transition that bears marks that are in a local copy of *acc* but we have not matched with yet, and will modify the local copy of *acc* to exclude those marks for the next match.

We also set up two filter functions:

- *filterNone*, which never filters anything,
- *filterSCC*, which filters out states whose order is lower than the one in *acc*: these states were less deep than the accepting SCC during the depth-first search.

For Step 1 (Section 3.4) we run `product_bfs_steps` from the initial `product_state` of the automata with *matchOrder* and *filterNone*: we are looking for the entry state of the accepting SCC and are restricting the search only to states that were discovered and "live" when the emptiness check stopped. We get back that state, and store it in *substart*. The retrieved runs are stored in the *prefix* part of the given runs.

For Step 2 we run `product_bfs_steps` from *substart* with *matchMarks* and *filterSCC*: we look for any transition bearing an acceptance mark that is still in our local copy of *acc*, that is to

[1] For ease of reading, its field have also been called *left* and *right*.

```
1  Function product_bfs_steps
       Input: product_state start, functions match and filter, lists of product_steps
              steps_ℓ and steps_r already allocated or set to None, states from Section 3.2
       Output: The product_state to which points the matching transition
2      backlinks ← map of product_states to product_steps
3      todo ← queue of product_states
4      todo.push(start)
5      while todo is not empty do
6          src ← todo.top()
7          iter ← product_iterator(src.left.successors, src.right.successors)
8          while iter.HasSuccessors do
9              dst ← product_state(iter.left.destination, iter.right.destination)
10             if dst is in states                          // state is undiscovered
11                 and states[dst] ≠ 0                              // state is dead
12                 and not filter(dst) then
13                 cur_step ← product_step(src, iter.condition, iter.marks)
14                 if match(cur_step, dst) then
15                     tmp_ℓ, tmp_r ← stacks of steps
16                     Loop
17                         if steps_ℓ is not None then
18                             tmp_ℓ.push(cur_step.src.left, cur_step.condition.left,
                                cur_step.marks.left)
19                         if steps_r is not None then
20                             tmp_r.push(cur_step.src.right, cur_step.condition.right,
                                cur_step.marks.right)
21                         if cur_step.src = start then
22                             break
23                         cur_step ← backlinks[cur_step.src]
24                     if steps_ℓ is not None then
25                         while tmp_ℓ is not empty do
26                             steps_ℓ.append(tmp_ℓ.top())
27                             tmp_ℓ.pop()
28                     if steps_r is not None then
29                         while tmp_r is not empty do
30                             steps_r.append(tmp_r.top())
31                             tmp_r.pop()
32                     return dst
33                 if dst not in backlinks then
34                     backlinks[dst] ← cur_step
35             iter ← iter.NextSuccessor
36         todo.pop()
37     return start                              // no match found
```

Algorithm 4.1: Pseudocode for the `product_bfs_steps` algorithm

```
1  Function TwoAutomatonAcceptingRunSearch
       Input: Automata left and right, runs run_ℓ and run_r already allocated or set to None,
              states and a copy of acc from Section 3.2, product_state start the initial
              state
       Output: Allocated runs are filled with corresponding accepting runs
2      if run_ℓ is None
3        and run_r is None then
4          return
5      prefix_ℓ ← if run_ℓ is None then None else run_ℓ.prefix
6      prefix_r ← if run_r is None then None else run_r.prefix
7      cycle_ℓ ← if run_ℓ is None then None else run_ℓ.cycle
8      cycle_r ← if run_r is None then None else run_r.cycle
9      matchOrder ← lambda step, dst: return states[dst] = acc.order
10     matchMarks ← Function
           Input: product_step step, product_state dst
11         if step.marks.left & acc.marks.left
12           or step.marks.right & acc.marks.right then
               // Remove step.marks from acc.marks
13             acc.marks.left ← acc.marks.left & not steps.marks.left
14             acc.marks.right ← acc.marks.right & not steps.marks.right
15             return true
16         return false
17     filterNone ← lambda dst: return false
18     filterSCC ← lambda dst: return states[dst] < acc.order
19     substart ← product_bfs_steps(start, matchOrder, filterNone, prefix_ℓ, prefix_r)
20     while acc.marks.left & acc.marks.right do
21         substart ← product_bfs_steps(substart, matchMarks, filterSCC, cycle_ℓ, cycle_r)
22     if states[substart] ≠ acc.order then
23         product_bfs_steps(substart, matchOrder, filterSCC, cycle_ℓ, cycle_r)
```

Algorithm 4.2: Pseudocode for the two-automaton accepting run search

say, that is in the accepting SCC but has not been matched yet, while staying in the accepting SCC. We store the destination state of that transition back in *substart*, we append the retrieved runs to the *cycle* part of the given runs, and we loop back on that step while there are still unseen marks.

For Step 3 we first check if we have looped back to the entry state of the accepting SCC, if not we run a final `product_bfs_steps` from *substart* with *matchOrder* and *filterSCC*: we want to get back to the entry state, while staying in the accepting SCC. The retrieved runs are appended to *cycle* part of the given runs.

The two-automaton accepting run search is designed such that the caller could choose to only build an accepting run over the left or right automaton.

4.4 Integration with the series of algorithms

Since the accepting run search is only going to be run *after* an emptiness check, and run with some of its data, it makes sense that it would be called on a object built by the emptiness check. We modified the two-automaton emptiness check to return a structure called `two_aut_res` instead of a Boolean. This structure contains pointers to the two automata, the `states` and `acc` structures described in Section 3.2, and a Boolean telling us if the automata were swapped (Gillard, 2017, Sections 5.2 and 5.3). Along with this structure come functions to run the accepting run search and return a run over the left automaton, the right automaton, or a pair of runs over each one. The user cannot access the fields of the `two_aut_res` outside of these methods: this way it only matters to the methods if the automata have been swapped or not.

Instances of this structure are hidden behind a `std::shared_ptr`, which allows for easy Boolean conversion: when there is no intersection, the two-automaton emptiness check returns a null pointer which is easily converted into a Boolean `false`, whereas when there is one, it returns a pointer to a filled `two_aut_res` which is easily converted into a Boolean `true`. This allows us to keep the usability of the old interface that returned Booleans, and ensures that the accepting run search is only run on filled `two_aut_res` (and that the user checked the result of the emptiness check before running the accepting run search, unless they want to dereference a null pointer).

The chaining of algorithms and data structures is illustrated in Fig. 4.1.

4.5 Integration in Spot

The two-automaton emptiness check was already integrated to the `intersects` method in place of the on-the-fly product and emptiness check, but this had little effect since this method is not used a lot: most algorithms, when checking for emptiness, also require an example of a word that is in the languages of both automata. The two-automaton accepting run search was therefore integrated to replace the on-the-fly product and accepting run search of `intersecting_run`, and `intersecting_word` was rebuilt around `intersecting_run`. All those changes were done with the ability to switch back to the old implementation if an environment variable is set to tell us so.

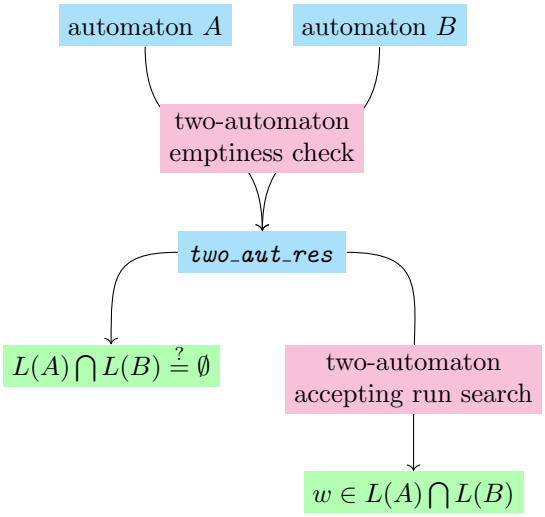

Figure 4.1 – The two-automaton series of algorithm

4.6 Additional changes

Some algorithms used a feature of the product implemented in Spot, which allowed choosing which states would be taken as initial states in each automaton, instead of asking the automata for their initial states. This was ported into the two-automaton emptiness check and the two-automaton accepting run search.

The implementation of the two-automaton accepting run search allowed us to use the two-automaton emptiness check in a lot more algorithms than we were previously able, since most of these algorithms were built around the retrieval of a counterexample in the case of a non-empty intersection. This meant that the two-automaton emptiness check was now executed on a much wider range of automata of various types, built by various tools, and revealed several problems with the implementation, especially in terms of memory management, which are now fixed.

Chapter 5

Benchmarks

5.1 Comparison of two-automaton against existing implementations

5.1.1 Setup

We generated 100 random automata with Spot's `randaut` tool, each with the following properties:

- 500 states,
- a density of transitions of 7.5%
- 16 acceptance sets,
- an acceptance condition being a conjunction of Inf of all those sets,
- 10 atomic propositions to express the conditions over the transitions with, common across all automata.

These automata are of the explicit kind.

We combined these automata into all 5050 different pairs: each automaton is in a pair with each other automaton plus itself. We then computed the time spent during the execution of:

- explicit and on-the-fly products,
- explicit and on-the fly emptiness checks over their respective products,
- two-automaton emptiness check,
- in the case where the product had a non-empty language:
  - explicit and on-the-fly accepting run searches over their respective products,
  - two-automaton accepting run search over the `two_aut_res` got from the two-automaton emptiness check,

for every pair. These computations were done 24 pairs in parallel over 24 cores of the same machine.

Of those 5050 pairs, 2327 of them gave products with empty languages.

5.1.2 Expectations

The following table shows the expected time performance of the existing implementations compared to the two-automaton algorithms.

	explicit	on-the-fly
product and emptiness check when the product is empty	**Similar:** both would build the whole product and explore it completely using the explicit interface of the automata.	**Slower**[1]**:** would perform the same operations using the on-the-fly interface.
product and emptiness check when the product is not empty	**Slower**[1]**:** the whole product would have to be built but not explored.	
product, emptiness check and accepting run search when the product is not empty	**Slower**[1]**:** would perform the same operations with the same interface, and would not get back the time lost building the whole product.	

5.1.3 Results

Size of the automata

Figure 5.1 shows the comparison of the time of computation to the size of the products. We can see that the pairs of automata can be split into three distinct groups:

- the big (more than 100000 states), non-empty products that take little time to process (less than microsecond for the two-automaton emptiness check), less than 100,
- the small (less than 60 states), empty products, 2327 ,
- the big (more than 100000 states), non-empty products that take longer to process, about 2600.

Empty product

Contrary to what we expected, we can see in Fig. 5.2 that the two-automaton emptiness check is about two times slower than the explicit product and emptiness check. This can be attributed to the fact that the explicit product is built once and then worked upon, while the two-automaton emptiness checks always manipulates two automata at the same time. We can also see that shorter computations take longer with the two-automaton emptiness check, but that the time difference reduces with longer computations : this may indicate that we have a constant overhead. This is confirmed by the fact that shorter computations also take longer against the on-the-fly product and emptiness check.

Non-empty product

As expected, we can see in Fig. 5.3 that the explicit product and emptiness check are indeed slower than their on-the-fly counterparts and the two-automaton emptiness check, due to the

[1] Compared to the two-automaton equivalent of the algorithm.

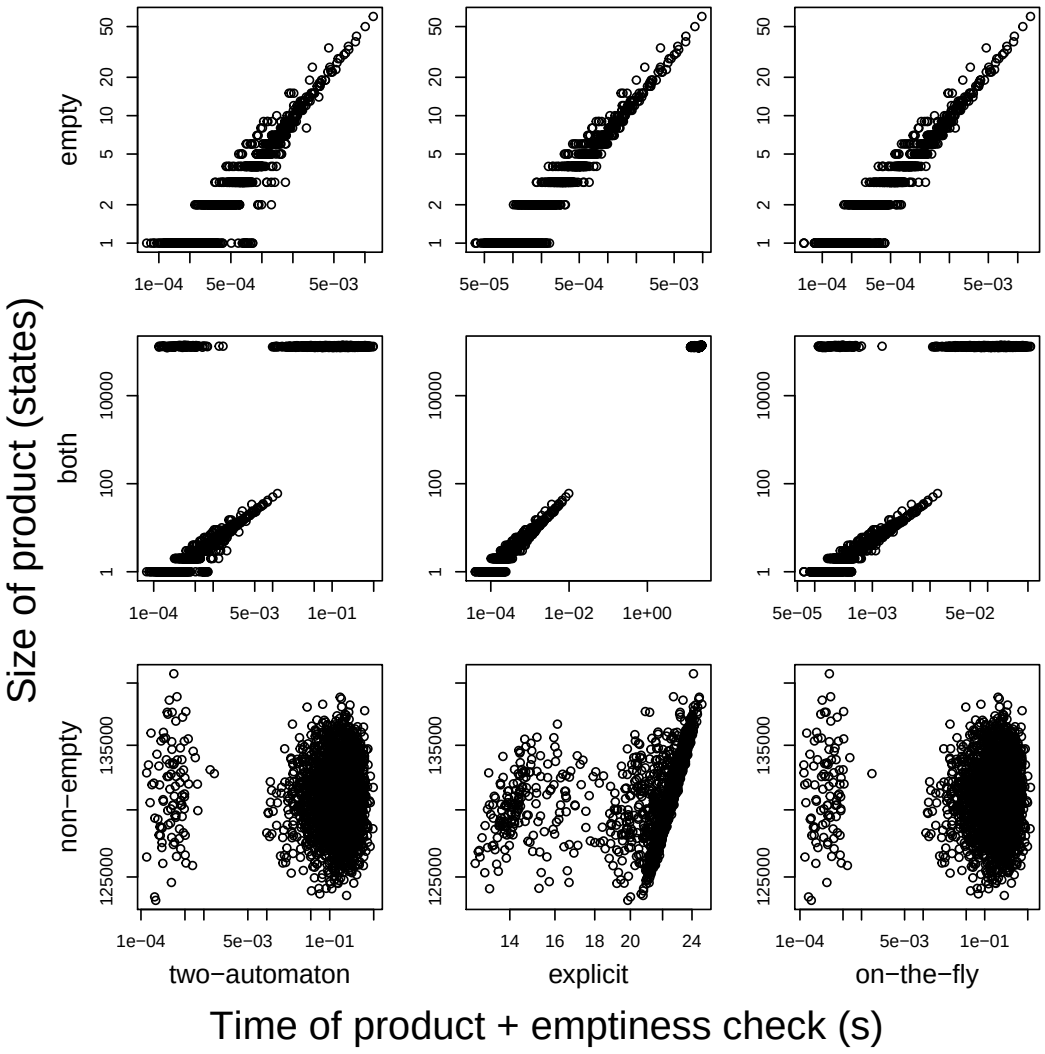

Figure 5.1 – Comparison of the time of products and emptiness check to size of product on explicit automata

useless exploration of the whole product. We can also see that the two-automaton emptiness check is, as predicted, faster than the on-the-fly product by an average of 10%, but it is still beaten on the smaller products, which goes in the direction of a constant overhead.

Finally, in Fig. 5.4 is shown the comparison of the duration of the product, emptiness check and accepting run search in the case of a non-empty product. We can see that as expected the explicit algorithms did not recover from the cost of the exploration of the product, and that on average the two-automaton series of algorithms is faster than the on-the-fly algorithms. However we can also see that some of the advance gotten during the two-automaton emptiness check is lost during the accepting run search, meaning that this implementation is slower than

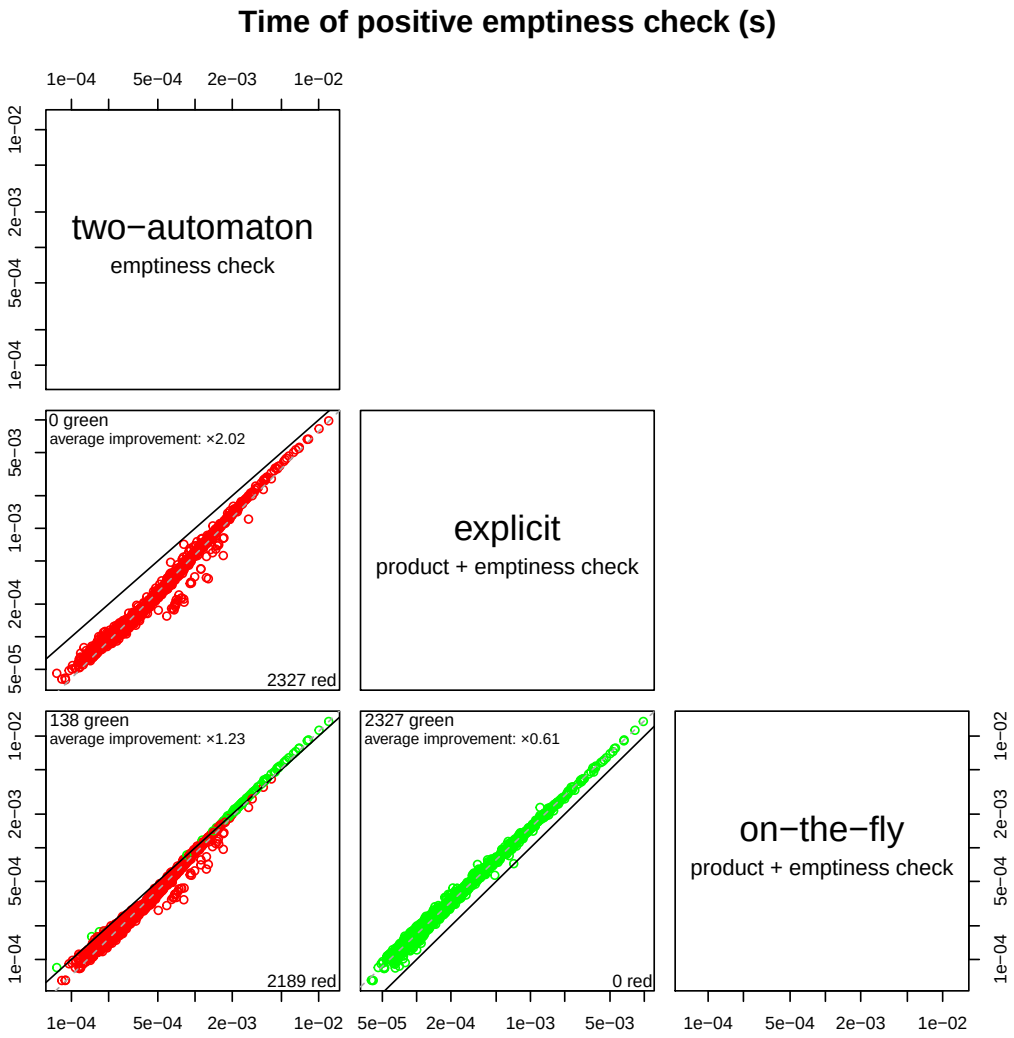

Figure 5.2 – Comparison of products and emptiness checks on explicit automata when the language of the product is empty

the on-the-fly one.

5.2 Use case: `ltlcross`

5.2.1 Setup

We replaced the implementation of the cross check of automata in `ltlcross` from an on-the-fly product to the two-automaton emptiness check. This could not be done previously since the point of this check is to return a counterexample if the product's language is not empty.

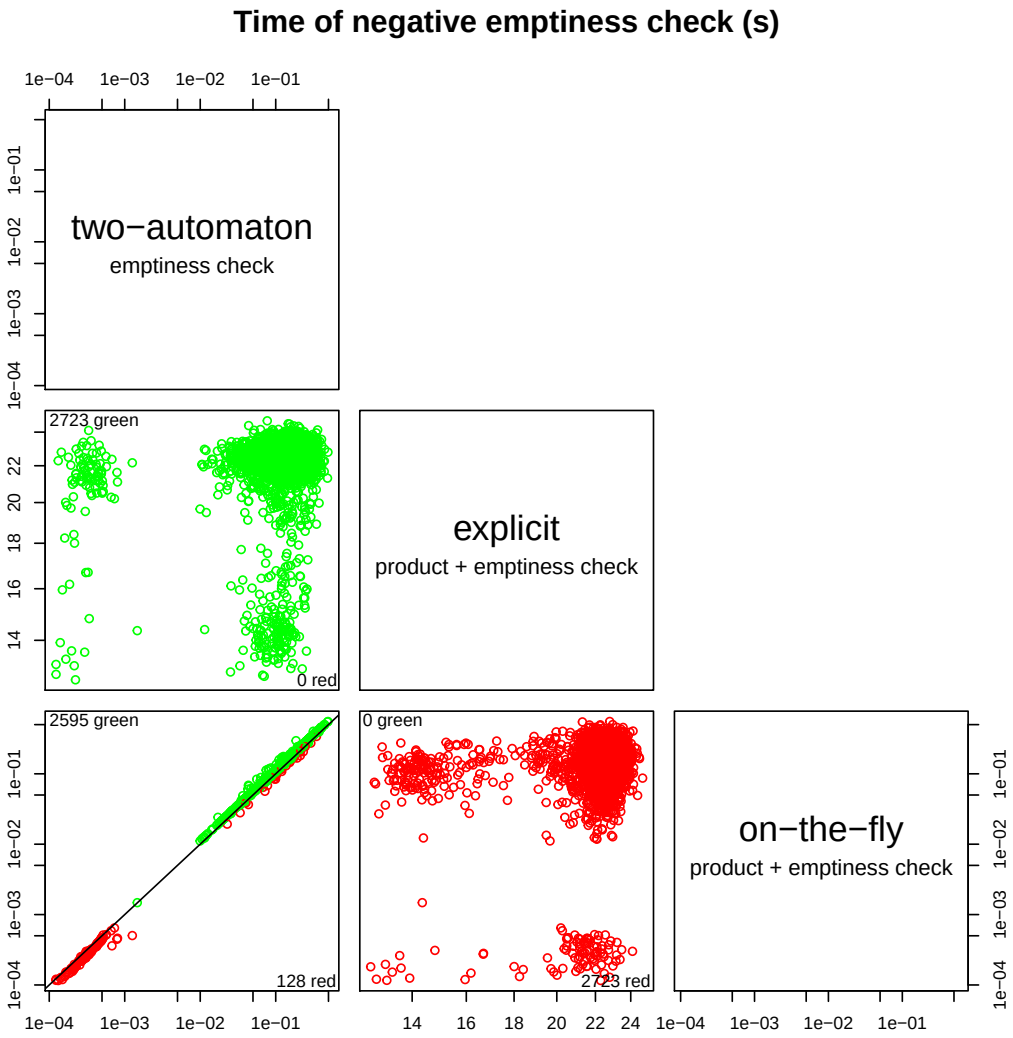

Figure 5.3 – Comparison of products and emptiness checks on explicit automata when the language of the product is not empty

We generated 100 random LTL formulae with Spot's `randltl` tool and ran `ltlcross` with 3 LTL-to-ω-automata translators. The command for the generation was `randltl --tree-size=30 30`. The command for the execution was `ltlcross -T 60 ltl2ba ltl3ba ltl2tgba`.

The execution was done in parallel over 2 cores of the same machine. We measured the time spent during each execution of the emptiness check.

In total, 27 timeouts occurred during the translations, and 849 emptiness checks were run for each implementation.

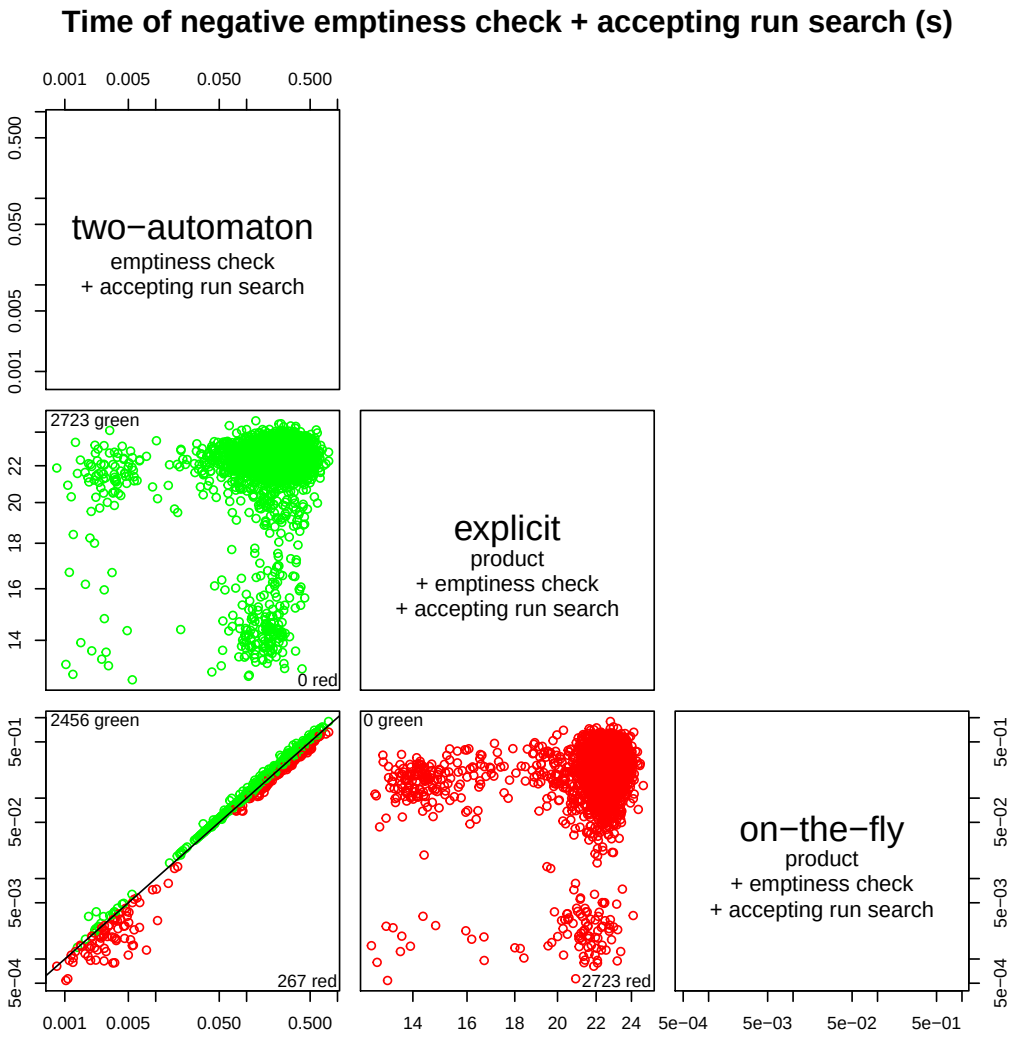

Figure 5.4 – Comparison of product, emptiness check and accepting run search on explicit automata

5.2.2 Results

Figure 5.5 shows the individual times spent computing the emptiness checks. We can see an improvement of the times[2]. We can see an improvement of about 70% on average, which may be attributed to the use of the explicit interface instead of the on-the-fly one.

[2]The single outlier is the formula $(!(X(p11))) \Rightarrow (!(p0))$ and its negation, put through `ltl3ba`, and checked against each other.

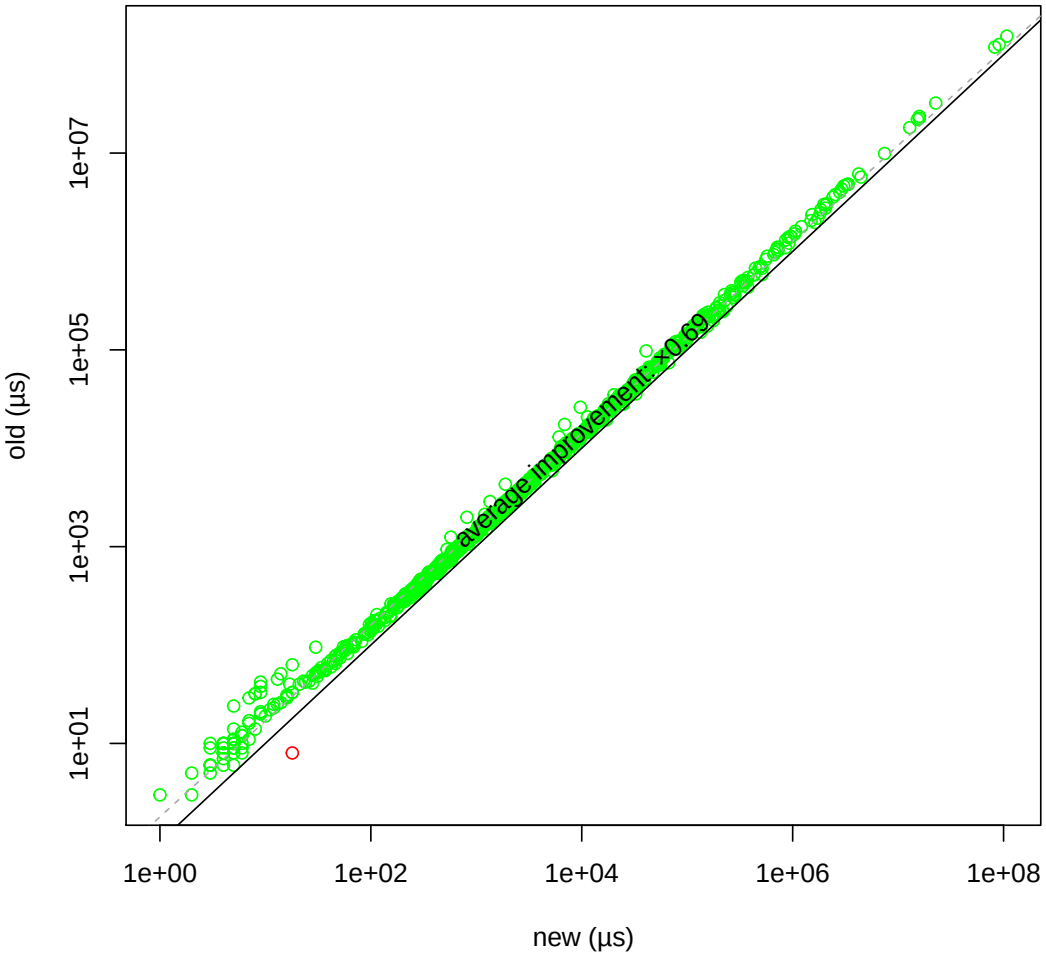

Figure 5.5 – Comparison of old vs. new emptiness check in `ltlcross`

Chapter 6

Conclusion

The two-automaton accepting run search completes the two-automaton emptiness check and allows us to use it in far more places than on its own. We have seen that even though the two-automaton series of algorithms is not more efficient than some of the existing implementations, it is still faster on the cases on which we are more likely to use it. But on top of all, it frees us from the limit of the amount of acceptance sets when doing computations that require a product of automata.

```
info: check_empty P1*N0
info: check_empty P1*N1
info: building Comp(N1)*Comp(P1) requires more acceptance sets than supported
info: building state-space #0/1 of 200 states with seed 0
info: state-space has 4225 edges
```

```
info: check_empty P1*N0
info: check_empty P1*N1
info: check_empty Comp(N1)*Comp(P1)
info: building state-space #0/1 of 200 states with seed 0
info: state-space has 4225 edges
```

Figure 6.1 – A message we will never see again in the output of `ltlcross`!

We may now investigate as to why the two-automaton algorithms are slower than others when we do not expect them to.

We would also like to expand the field of ω-automata accepted by the two-automaton emptiness check to those with generalised Rabin acceptance conditions, as per Bloemen et al. (2017); the current implementation following Couvreur (1999) which only supports generalised Büchi acceptance conditions.

Chapter 7

Bibliography

Bloemen, V., Duret-Lutz, A., and van de Pol, J. (2017). Explicit state model checking with generalized büchi and rabin automata. In *Proceedings of the 24th International SPIN Symposium on Model Checking of Software (SPIN'17)*, pages 50–59. ACM. (page 27)

Couvreur, J.-M. (1999). On-the-fly verification of temporal logic. In Wing, J. M., Woodcock, J., and Davies, J., editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, Toulouse, France. Springer-Verlag. (pages 9, 11, and 27)

Couvreur, J.-M., Duret-Lutz, A., and Poitrenaud, D. (2005). On-the-fly emptiness checks for generalized Büchi automata. In Godefroid, P., editor, *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, volume 3639 of *Lecture Notes in Computer Science*, pages 143–158. Springer. (page 13)

Duret-Lutz, A. (2017). *Contributions to LTL and ω-Automata for Model Checking*. Habilitation thesis, Université Pierre et Marie Curie (Paris 6). (page 5)

Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., and Xu, L. (2016). Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer. (page 5)

Gillard, C. (2017). Two-automaton emptiness check in spot. Technical Report 1706, EPITA Research and Development Laboratory (LRDE). (pages 6, 11, 14, and 18)

Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA. IEEE Computer Society. (page 5)