Counterexample searches in Spot

Clément Gillard
(supervisor: Alexandre Duret-Lutz)

Technical Report n^o1903, January 2019
revision 49692f5e

Spot is an ω-automata manipulation library who aims to help doing ω-automata-theoretic approach to model checking or develop tools for ω-automata transformation. As such, it provides many algorithms, with many different implementations depending of the specificities of each ω-automaton.
Of those algorithms, emptiness check algorithms and counterexample search algorithms are often used, with various goals, and since their results are linked, they are often used together. However, some implementations of emptiness check algorithms lack a similar counterexample search implementation which is able to work in the same way on the same automata.
We introduce two implementations of counterexample computation, which complete two already existing implementations of emptiness check algorithms by walking in their footsteps and reusing some of the data they gathered to efficiently compute counterexamples.

Spot est une bibliothèque de manipulation d'ω-automates qui tend à aider la vérification de modèles par ω-automates et le développement d'outils de transformation d'ω-automates. Elle fournit donc de multiples algorithmes avec de multiples implémentations qui fonctionnent sur une grande variété d'ω-automates.
Parmi ces algorithmes, les tests de vacuité et les recherches de contrexemple sont souvent utilisés pour de diverses raisons. Comme leurs résultats sont liés, ils sont souvent utilisés ensemble. Cependant, il manque à Spot les implémentations de recherche de contrexemple correspondant à certaines implémentations de tests de vacuité qui soient capables de travailler de la même manière sur les mêmes automates.
Nous présentons deux implémentations de calcul de contrexemple, qui viennent compléter deux implémentations de tests de vacuité déjà existantes, qui suivent leur sillage et se servent des données dejà accumulées pour construire efficacement des contrexemples.

Keywords
Spot, emptiness check, counterexample search

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@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Chapter 1

Introduction

Model checking is the field of computer science that looks into the exhaustive and automatic proof of the specifications of a system. From a formalisation of the system, called a model, and a set of specifications, we want to prove that the system meets all the requirements. The automata-theoretic approach to model checking is one of the ways to perform these verifications, in which we solve the problem with graph theory and ω-automata theory, a field of automata theory that considers automata which execute on infinitely long inputs.

The ω-automata-theoretic approach to model checking comes from turning the model checking problem into an ω-automaton problem (Fig. 1.1). The goal of model checking is to prove that a model M always satisfies a property φ (Eq. (1.1)). By turning the model and the property into ω-automata A_M, whose words are sequences of valid inputs in the model, and A_φ, whose accepting words are all the words satisfying the property, we now have turned the problem to only have to check that the language of the former is included in the language of the latter (Eq. (1.2)). This is equivalent to checking if the language of A_M has an intersection with the complement of the language of A_φ (Eq. (1.3)). The complement of the language of an automaton is the language of the complement of the automaton (Eq. (1.4)), but complementing an automaton is an expensive operation; we would rather compute the automaton of the negation of the property $A_{\neg\varphi}$ which is equivalent to $\overline{A_\varphi}$: they match all the words which do not satisfy φ (Eq. (1.5)). The intersection of the languages of two automata is actually the language of the synchronised product of the automata (Eq. (1.6)), so we only have to check the language of the product $A_M \otimes A_{\neg\varphi}$.

$$
\begin{aligned}
& M \models \varphi && (1.1)\\
\iff\ & L\left(A_M\right) \subseteq L\left(A_\varphi\right) && (1.2)\\
\iff\ & L\left(A_M\right) \cap \overline{L\left(A_\varphi\right)} = \emptyset && (1.3)\\
\iff\ & L\left(A_M\right) \cap L\left(\overline{A_\varphi}\right) = \emptyset && (1.4)\\
\iff\ & L\left(A_M\right) \cap L\left(A_{\neg\varphi}\right) = \emptyset && (1.5)\\
\iff\ & L\left(A_M \otimes A_{\neg\varphi}\right) = \emptyset && (1.6)
\end{aligned}
$$

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

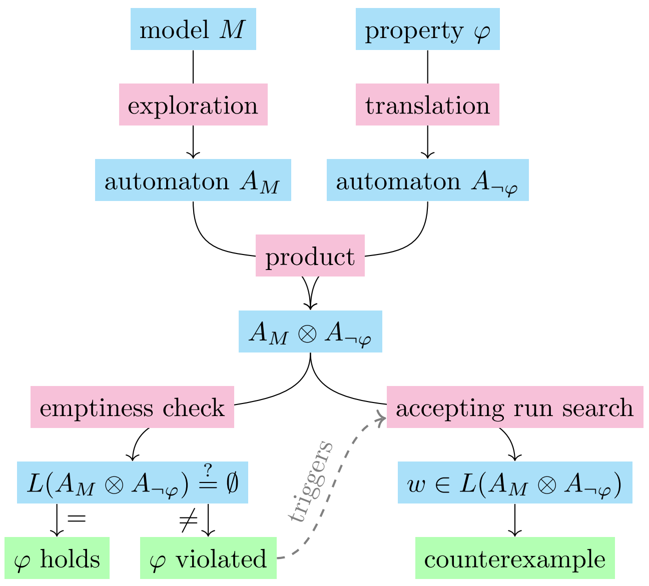

Figure 1.2 – The algorithmic chain of the ω-automata approach to model checking

We have reduced the problem of checking if all the possible inputs given to a system leave it in a state where a property is satisfied, to checking if the language of a single automaton is empty.

The chaining of algorithms to deploy for the ω-automata-theoretic approach to model checking is illustrated in Fig. 1.2. It works by first turning the model M into an ω-automaton A_M, with a tool like LTSmin (Kant et al., 2015) which plugs in with DiVinE (Baranová et al., 2017) or SpinS (Holzmann, 1997), and a property φ, usually written in a logic like the LTL, Linear Temporal Logic (Pnueli, 1977), into the automaton of its negation $A_{\neg\varphi}$. We then compute the product of the automata.
From there, we have several options: either run an emptiness check algorithm (Couvreur, 1999) or an accepting run search (Couvreur et al., 2005; Gastin and Moro, 2007; Gastin et al., 2004; Hansen and Geldenhuys, 2008; Hansen and Kervinen, 2006) on the product automaton. Emptiness checks answer the question "Is the language of this automaton empty?", where, if the answer is "Yes", is sufficient to prove that the property φ always holds in the model M. However, if the answer is "No", we only know that there exists at least one word in the intersection of the languages of A_M and $A_{\neg\varphi}$, we do not have such a word. Accepting run searches actually look for an example of an accepting word, which in the case of model checking corresponds to a counterexample to the proof we are trying to make. Even though they process the automaton in similar ways, emptiness checks are usually lighter algorithms than accepting run searches, but they answer a broader question; to avoid redundancy and optimise performance, a common implementation of accepting run searches is to first do an emptiness check, and when sure that there exists an accepting run, search for it, reusing data gathered by the emptiness check. This way, if the language is empty, only the emptiness check is run, and if not, the accepting run search is more efficient than it would be if it had been run first. The chaining of those algorithms is illustrated in Fig. 1.2.

Another case where emptiness checks and accepting run searches are used is to check the

$$\begin{aligned}
& A_1 \equiv A_2 \\
\iff & L(A_1) = L(A_2) && (1.7)\\
\iff & L(A_1) \subseteq L(A_2) \quad \text{and } L(A_1) \supseteq L(A_2) && (1.8)\\
\iff & L(A_1) \cap \overline{L(A_2)} = \emptyset \text{ and } \overline{L(A_1)} \cap L(A_2) = \emptyset && (1.9)\\
\iff & L(A_1) \cap L\left(\overline{A_2}\right) = \emptyset \text{ and } L\left(\overline{A_1}\right) \cap L(A_2) = \emptyset && (1.10)\\
\iff & L\left(A_1 \otimes \overline{A_2}\right) = \emptyset \quad \text{and } L\left(\overline{A_1} \otimes A_2\right) = \emptyset && (1.11)
\end{aligned}$$

Figure 1.3 – How to check for the equivalence of ω-automata

equivalence of automata (Fig. 1.3). Automata are equivalent if their languages are equal (Eq. (1.7)), i.e. if they accept exactly the same words. The equality of language can be tested by checking inclusion both ways (Eq. (1.8)). As we have seen before, inclusion checks can be replaced by intersection checks with complements of languages of automata (Eq. (1.9)), which are equivalent to languages of complement of automata (Eq. (1.10)), and intersection checks can be replaced with emptiness checks of products (Eq. (1.11)).

Imagine that we are developing a new tool for automata transformation, or translation from a model or formula. We want to check that this tool actually produces equivalent automata to other, more thoroughly tested tools. We start by generating the automata A_1 and A_2 from the tools. We then compute the complement $\overline{A_1}$ to one of the tool's automata then compute the product of that complement with the other automata. We then check that the product is empty by running an emptiness check on it; but if we get that the language is not empty (which means the starting automata are not equivalent), we need to run an accepting run search to help us debug our tool.

*
* *

Spot 2 (Duret-Lutz et al., 2016) is a library whose goal is to provide tools for LTL and ω-automata manipulation. It is developed in C++ and ships with a Python interface and several binaries to generate, translate, and process formulæ and ω-automata. Spot manipulates multiple kinds of automata, with different types of acceptance conditions, and different representations; it must provide generic implementations of manipulation algorithms, as well as implementations optimised for only a particular type of representation, or acceptance condition... But, when algorithms come linked, like with emptiness checks and accepting run searches, you run into the problem of having to develop two algorithms for the same usage. This explains why some emptiness check implementations do not have their accepting run search counterpart in Spot.

Table 1.1 shows the implementations of emptiness check and accepting run search algorithms in Spot which we will talk about in this paper. Blue ticks indicate the algorithm was presented in Gillard (2017), purple ticks, presented in Gillard (2018), while red ticks and text indicate what will be treated in this report. Black ticks indicate algorithms which were developed prior to these reports and whose implementations will not be discussed here.

"On-the-fly" ω-automata are an implementation of ω-automata of Spot, where states and transitions are only accessible through methods, which means that their exploration relies primarily on code. This is useful to generate an automaton as it is being requested, or reading a large au-

Name	Type of automata	Emptiness check	Accepting run search
Couvreur New:	Generalised Büchi		
	– on-the-fly	✔	✔
	– explicit	✔	✔
Product EC&CE[1]:	Generalised Büchi		
	– on-the-fly	✔ (2017)	✔ (2018)
	– explicit	✔ (2017)	✔ (2018) (benchmarks: 2019)
Generic EC:	Any		
	– explicit	✔	✔ (2019)

Table 1.1 – Emptiness check and Accepting run search algorithms in Spot

tomaton bits at a time from files to lower RAM usage. This is used to represent some models, but is a slower implementation which requires more memory per state and transition.

"Explicit" ω-automata, on the other hand, are an implementation of ω-automata of Spot where states and transitions are known in advance and stored efficiently in memory. This is a faster implementation, but requires the whole automaton to be stored at once in memory, which does not fit some usages. Explicit ω-automata offer an on-the-fly interface, such that they can be mixed transparently with on-the-fly ω-automata but that interface is slower than the actual explicit interface, which is why some algorithms are reworked to work with the explicit interface.

"Couvreur New" is the name given to a recent implementation of the algorithm introduced in Couvreur (1999); it works on both on-the-fly and explicit automata, but only works with generalised Büchi acceptance conditions (see Definition 2.16). The generic emptiness check is a new algorithm which is able to determine emptiness with any acceptance condition, but only on explicit automata.

This report will present once again the product counterexample search of Gillard (2018), with updated benchmarks, and the implementation of a counterexample search based on the generic emptiness check.

[1]Product Emptiness Check and CounterExample search, formerly called "two-automaton emptiness check" and "two-automaton accepting run search".

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 integers 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 2.7) are ω-automata whose acceptance sets are sets of transitions. This is how they are handled by Spot.

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:*

$$\phi \coloneqq \top \mid \bot \mid \mathit{Inf}(x) \mid \mathit{Fin}(x) \mid \phi \land \phi \mid \phi \lor \phi \mid (\phi)$$

where x denotes an acceptance set.

As can be seen in Fig. 2.1, we may directly an acceptance mark in place of a set in the acceptance condition for ease of reading.

Definition 2.7 (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,*
- *$m \colon \Delta \mapsto \mathcal{P}([n])$ is a function returning the acceptance marks of a transition,*
- *ϕ is an acceptance condition.*

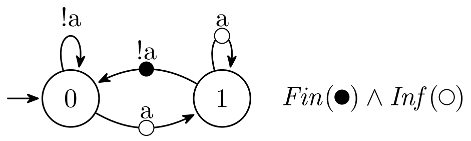

Figure 2.1 – The deterministic transition-based ω-automaton of the translation of the LTL formula `FG(a)`, with its acceptance condition on the right, and the acceptance marks on its transitions.

Definition 2.8 (State-based ω-automaton) *A state-based ω-automaton is defined in the same way as a transition-based ω-automaton (see Definition 2.7), except for m, which is defined as $m \colon Q \mapsto \mathcal{P}([n])$ a function returning the acceptance marks of a state.*

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 a run ρ of an ω-automaton. We write $\rho \vDash \phi$ to mean ρ satisfies ϕ and $\rho \nvDash \phi$ to mean ρ does not satisfy ϕ. The satisfaction of an acceptance condition is interpreted by induction as follows:*

$$\begin{array}{lcllcl}
\rho & \vDash & \top & & & \\
\rho & \nvDash & \bot & & & \\
\rho & \vDash & Inf(x) & \iff & x \in m_{Inf} \\
\rho & \vDash & Fin(x) & \iff & x \notin m_{Inf} \\
\rho & \vDash & \phi_1 \land \phi_2 & \iff & \rho \vDash \phi_1 \text{ and } \rho \vDash \phi_2 \\
\rho & \vDash & \phi_1 \lor \phi_2 & \iff & \rho \vDash \phi_1 \text{ or } \rho \vDash \phi_2
\end{array}$$

A run ρ of an automaton is an accepting run if and only if it 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.*

Definition 2.14 (Büchi acceptance condition) *A Büchi acceptance condition is an acceptance condition with a single Inf operator, over a single acceptance set.*

Definition 2.15 (Büchi automaton) *A Büchi automaton is a state-based ω-automaton with a Büchi acceptance condition.*

Definition 2.16 (Generalised Büchi acceptance condition) *A generalised Büchi acceptance condition is an acceptance condition which does not make use of the Fin operator.*

Büchi and generalised Büchi acceptance conditions are examples of Fin-less acceptances conditions, which do not make use of the Fin operator.

Chapter 3

Generalities on accepting run searches

3.1 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*.

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

3.2 Algorithms for accepting run searches

Several papers show various ways of computing an efficient accepting run (Gastin and Moro, 2007; Gastin et al., 2004; Hansen and Geldenhuys, 2008; Hansen and Kervinen, 2006). However, many are actually computing them without any knowledge; they act as both emptiness check and accepting run search algorithms by gathering data necessary for both (Gastin et al., 2004; Hansen and Geldenhuys, 2008; Hansen and Kervinen, 2006). This does not match the model of Spot, since most of the time we are going to check against empty products, and do not want to gather the data necessary to build back an accepting run.

Moreover, we must consider that Spot does not use bitstate hashing, a method introduced by Morris (1968) which allows for a more memory efficient way of keeping track of visited states, but requires heavier algorithms if references to states are needed.
This simplifies greatly the algorithm: we need to keep hold of very little data on our path to an accepting cycle, since we are always assured to be able to retrieve them back easily. This also excludes Gastin and Moro (2007) which presents an algorithm for bitstate hashing. But we may not have to look this far ahead.

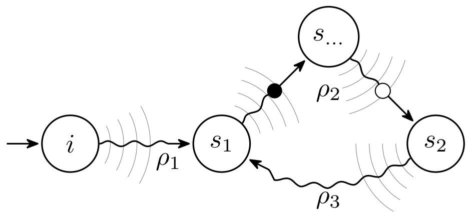

Figure 3.1 – Accepting run search on a transition-based generalised Büchi automaton

3.3 The search algorithm

"If states are stored in an hash table as usual, one can recover the trace of the counter-example using a BFS algorithm [. . .]. It then suffices to apply this BFS from the initial state i to s_1 to generate ρ_1, then to apply it from s_1 to s_2 to generate ρ_2 and finally to apply it once more from s_2 to s_1 to generate ρ_3."

According to this quote from Gastin and Moro (2007), if bitstate hashing is not mandatory, provided we have the states s_1 and s_2, then a simple breadth-first search is the most efficient way of retrieving the accepting run.

But what are those states s_1 and s_2? s_1 is described as the first encountered state of a loop of states containing s_2, and s_2 is an accepting state. By chance, we do have these states: emptiness check algorithms work by finding an accepting SCC, therefore we can find an entry point s_1, and we know which states belong to the SCC, hence we can find s_2.

Gastin and Moro also define three paths: ρ_1 from i to s_1, ρ_2 from s_1 to s_2, and ρ_3 from s_2 back to s_1. ρ_1 is actually the prefix of the run while ρ_2 and ρ_3 make up the cycle (see Section 3.1). This definition holds for Büchi ω-automata but we may extend it to generalised Büchi ω-automata by defining s_2 as the destination state of the last transition needed to satisfy the acceptance condition, and shaping ρ_2 to go through several marked transitions. This is illustrated in Fig. 3.1.

The accepting run search algorithm has three steps:

1. Search for a path from the initial state to any state of the accepting SCC. We denote as current state and also entry state the first encountered state of the SCC. This is ρ_1.

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. The concatenation of all those paths gives ρ_2.

3. From the current state, look for a path going back to the entry state while staying in the SCC. This is ρ_3.

Chapter 4

Product counterexample search

Note: this chapter is a rewriting and extension of Chapters 3 to 5 of Gillard (2018). The main differences lie in the replacement of the name of the algorithm family, from "two-automaton" to "product", and the updated benchmarks (in Section 4.7).

The goal of the "product" algorithms is to remove the limitation on the number of acceptance sets introduced by the product of automata in the checks for the intersection of the languages of two automata, 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 product accepting run search completes the product emptiness check introduced by Gillard (2017), a reimplementation of the on-the-fly emptiness check algorithm introduced by Couvreur (1999). The product 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.

4.1 Reminders on the product emptiness check

The product 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 (see Definition 2.16).

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 which 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, then 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.

4.2 The data available

The product 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.

4.3 Existing implementation of accepting run searches

As for the product emptiness check, the product 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 product 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 3.3, 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.

4.4 Implementation of the product accepting run search

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

4.4.1 Spot's data structures

The following structures are used in the product 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.

4.4.2 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`.

4.4.3 Pseudocode

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 are retrieved backwards, we need to store them in a stack, and then pop them back to get the actual runs; this is done lines 24 to 31.

```
Function product_bfs_steps
    Input: product_state start, functions match and filter, lists of product_steps
        $steps_\ell$ and $steps_r$ already allocated or set to None, states from Section 4.2
    Output: The product_state to which points the matching transition
    backlinks ← map of product_state to product_step
    todo ← queue of product_state
    todo.push(start)
    while todo is not empty do
        src ← todo.top()
        iter ← product_iterator(src.left.successors, src.right.successors)
        while iter.HasSuccessors do
            dst ← product_state(iter.left.destination, iter.right.destination)
            if dst is in states                        // state is undiscovered
              and states[dst] $\neq 0$                   // state is dead
              and not filter(dst) then
                cur_step ← product_step(src, iter.condition, iter.marks)
                if match(cur_step, dst) then
                    $tmp_\ell$, $tmp_r$ ← stacks of steps
                    Loop
                        if $steps_\ell$ is not None then
                            $tmp_\ell$.push(cur_step.src.left, cur_step.condition.left,
                              cur_step.marks.left)
                        if $steps_r$ is not None then
                            $tmp_r$.push(cur_step.src.right, cur_step.condition.right,
                              cur_step.marks.right)
                        if cur_step.src = start then
                            break
                        cur_step ← backlinks[cur_step.src]
                    if $steps_\ell$ is not None then
                        while $tmp_\ell$ is not empty do
                            $steps_\ell$.append($tmp_\ell$.top())
                            $tmp_\ell$.pop()
                    if $steps_r$ is not None then
                        while $tmp_r$ is not empty do
                            $steps_r$.append($tmp_r$.top())
                            $tmp_r$.pop()
                    return dst
                if dst not in backlinks then
                    backlinks[dst] ← cur_step
            iter ← iter.NextSuccessor
        todo.pop()
    return start                       // no match found
```

Algorithm 4.1: Pseudocode for the product_bfs_steps algorithm

```
Function ProductAcceptingRunSearch
    Input: Automata left and right, runs $run_\ell$ and $run_r$ already allocated or set to None,
           states and a copy of acc from Section 4.2, product_state start the initial
           state
    Output: Allocated runs are filled with corresponding accepting runs
    if $run_\ell$ is None
      and $run_r$ is None then
        return
    $prefix_\ell \leftarrow$ if $run_\ell$ is None then None else $run_\ell$.prefix
    $prefix_r \leftarrow$ if $run_r$ is None then None else $run_r$.prefix
    $cycle_\ell \leftarrow$ if $run_\ell$ is None then None else $run_\ell$.cycle
    $cycle_r \leftarrow$ if $run_r$ is None then None else $run_r$.cycle
    matchOrder $\leftarrow$ lambda step, dst: return states[dst] = acc.order
    matchMarks $\leftarrow$ Function
        Input: product_step step, product_state dst
        if step.marks.left & acc.marks.left
          or step.marks.right & acc.marks.right then
            // Remove step.marks from acc.marks
            acc.marks.left $\leftarrow$ acc.marks.left & not steps.marks.left
            acc.marks.right $\leftarrow$ acc.marks.right & not steps.marks.right
            return true
        return false
    filterNone $\leftarrow$ lambda dst: return false
    filterSCC $\leftarrow$ lambda dst: return states[dst] < acc.order
    substart $\leftarrow$ product_bfs_steps(start, matchOrder, filterNone, $prefix_\ell$, $prefix_r$)
    while acc.marks.left & acc.marks.right do
        substart $\leftarrow$ product_bfs_steps(substart, matchMarks, filterSCC, $cycle_\ell$, $cycle_r$)
    if states[substart] $\neq$ acc.order then
        product_bfs_steps(substart, matchOrder, filterSCC, $cycle_\ell$, $cycle_r$)
```

Algorithm 4.2: Pseudocode for the product accepting run search

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

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 4.2), 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 (from Section 3.3) 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 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 product accepting run search is designed such that the caller could choose to only build an accepting run over the left or right automaton.

4.5 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 an object built by the emptiness check. We modified the product emptiness check to return a structure called `product_emptiness_res` instead of a Boolean. This structure contains pointers to the two automata, the `states` and `acc` structures described in Section 4.2, and a Boolean telling us if the automata were swapped (Gillard, 2017, Sections 5.2 and 5.3). Along with this structure comes functions to run the accepting run search and return a run over the left automaton, the right automaton, or a pair of runs

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

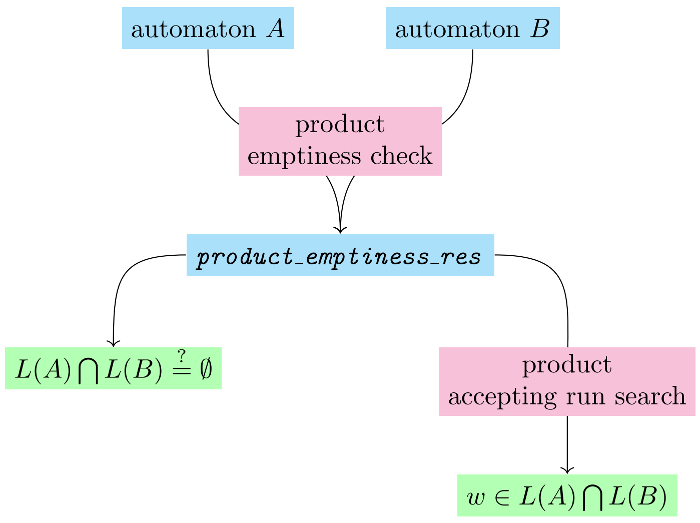
automaton A
automaton B
product emptiness check
product_emptiness_res
$L(A) \bigcap L(B) \overset{?}{=} \emptyset$
product accepting run search
$w \in L(A) \bigcap L(B)$

Figure 4.1 – The product algorithms

over each one. The user cannot access the fields of the `product_emptiness_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 product 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 `product_emptiness_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 `product_emptiness_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.

4.6 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 product emptiness check and the product accepting run search.

The implementation of the product accepting run search allowed us to use the product 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 product 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.

4.7 Benchmarks

Note: this section, although bearing the same structure, is not taken from Gillard (2018). The differences are a few minor changes in the implementation, which had no effect on the performances measured here, and a change on the benching method removing a huge cache bias against the product algorithms, which had a great effect on the performances.

4.7.1 Comparison of product algorithms against existing implementations

Benchmark 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 all these automata into 5050 different pairs: each automaton is in a pair with each other automaton plus itself; of those pairs, 2327 gave products with empty languages. We then computed the time spent during the execution of:

- **For Thread 1:**
  - explicit product,
  - explicit emptiness check on that product,
  - if the product is not empty, explicit accepting run search on that product,
- **For Thread 2:**
  - on-the-fly product,
  - on-the-fly emptiness check on that product,
  - if the product is not empty, on-the-fly accepting run search on that product,
- **For Thread 3:**
  - product emptiness check,
  - if the product is not empty, product accepting run search on the `product_emptiness_res`,

for every pair. These threads were executed in parallel on a dedicated machine with more than three physical cores.

	Explicit	On-the-fly
product and emptiness check when the product is empty	**Similar:** both algorithms would build the whole product and explore it completely using the explicit interface of the ω-automata	**Faster:** the product algorithms would perform the same operations as the on-the-fly algorithms, but would use the faster explicit interface to do so
product and emptiness check when the product is not empty	**Faster:** the explicit product would have to build the whole product, which is not required by the explicit emptiness check, whereas the product emptiness check would only build the required parts of the product	
product, emptiness check and accepting run search when the product is not empty	**Faster:** the product algorithms would perform the same operations on the same interface as the explicit algorithms, but the explicit product would have already lost time building the whole product	

Table 4.1 – Expectations on the new algorithms compared to the existing explicit and on-the-fly implementations

Expectations

Table 4.1 shows the expected time performance of the existing implementations compared to the new algorithms.

Results

Size of automata: Figure 4.2 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:

- in orange, the 100 pairs where an automaton was paired with itself, which all formed non-empty products and compute, for the biggest part, in less than a millisecond,
- in plum, the empty products, which are very small (less than 60 states) and compute in less than 10 milliseconds, there are 2327,
- in brown, the non-empty products, which are quite big (over 120 thousand states) which compute in more than a millisecond, there are 2623.

We can see that there is a huge bias in our dataset towards small empty products and big non-empty products. This is due to the method used to generate automata, which will either

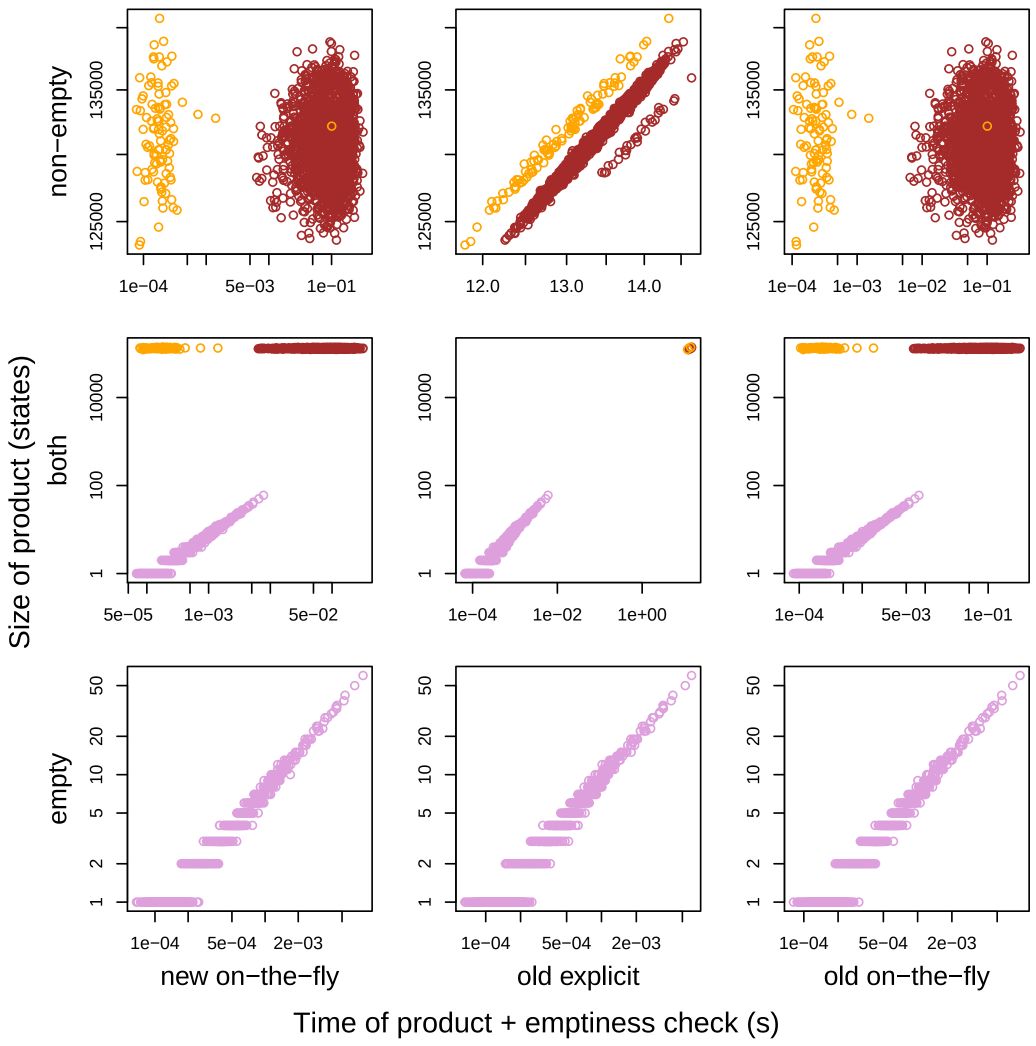

Figure 4.2 – Comparison of the time of products and emptiness check to size of product on explicit automata (all axes logarithmic)

give “compatible” automata whose transitions are going to merge, which will give a big product with huge chances of finding an accepting word, or “incompatible” automata whose transitions cancel each other out, giving a small, non-accepting product.

Empty products: Like we expected, we can see in Fig. 4.3 that the product emptiness check compares well with the already existing implementations, with an average time ratio[2] of 115% compared to the old on-the-fly implementation, and 94% to the explicit implementation. How-

[2]Computed as the mean, for each pair, of the time spent computing with the new algorithm divided by the time with the old algorithm.

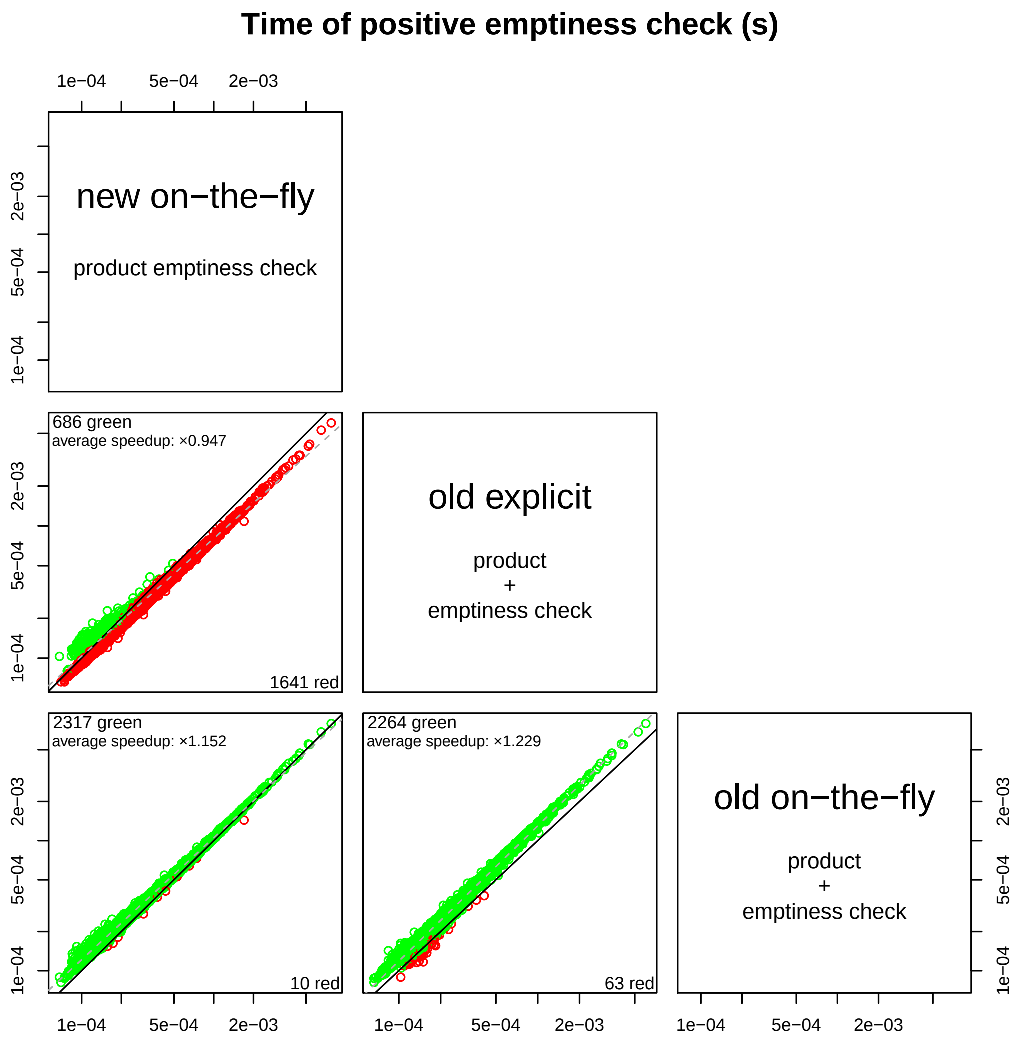

Figure 4.3 – Comparison of products and emptiness checks on explicit automata when the language of the product is empty (all axes logarithmic)

ever, even though they are quite close, we can see that the explicit algorithms still perform a bit better. This can be attributed to the fact that the explicit product is built once and then worked upon, while the product emptiness checks always manipulates two automata at the same time.

Non-empty product: As expected, we can see in Fig. 4.4 that the explicit product and emptiness check are indeed slower than the two on-the-fly emptiness checks, due to the useless exploration of the whole product. We can also see that the product emptiness check is, as predicted, faster than the on-the-fly product by an average of 4%.

Non-empty products: Finally, in Fig. 4.5 is shown the comparison of the duration of the prod-

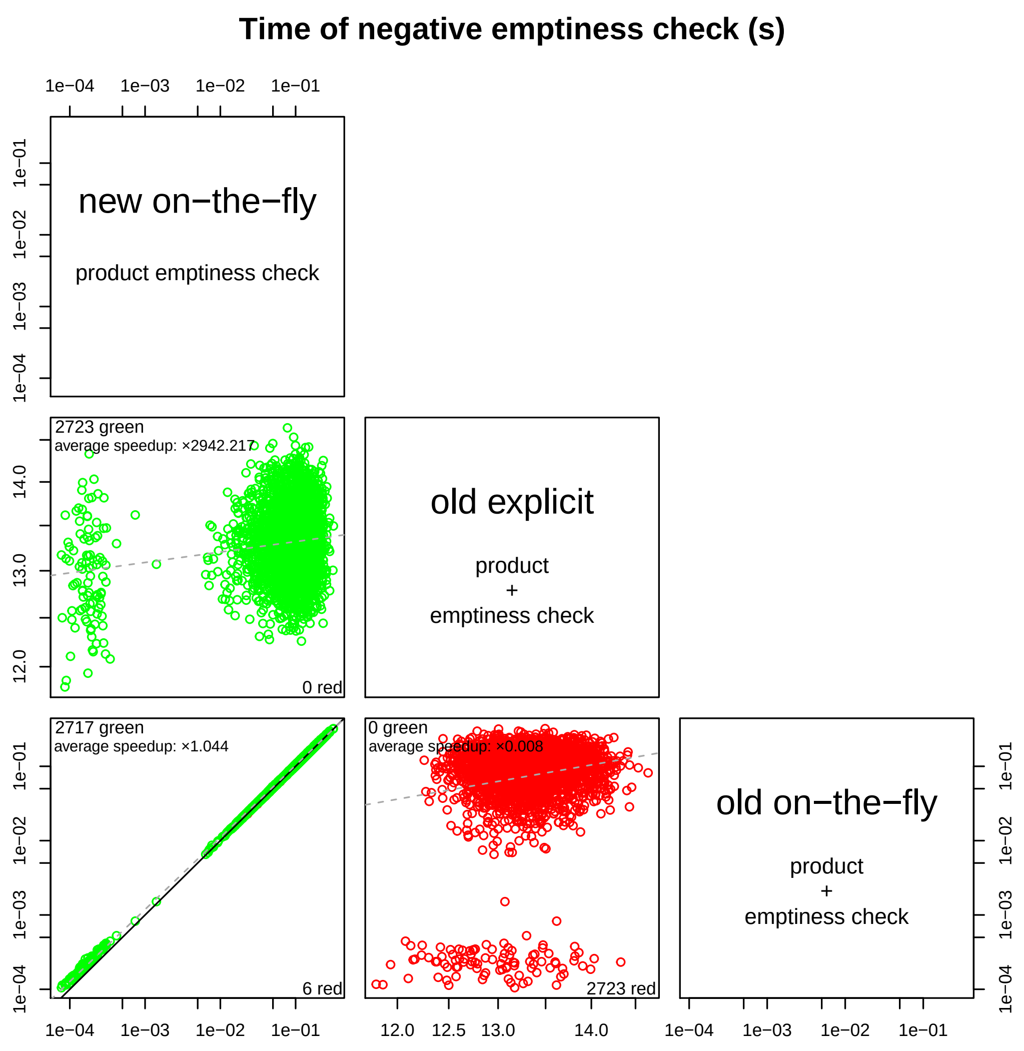

Figure 4.4 – Comparison of products and emptiness checks on explicit automata when the language of the product is not empty (all axes logarithmic)

uct, 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 new on-the-fly implementation is faster than the old one by an average of 2%. However, we can also see that the computation times of the product accepting run search are much more spread than those of the on-the-fly one, resulting in more products being processed more quickly by the on-the-fly algorithms.

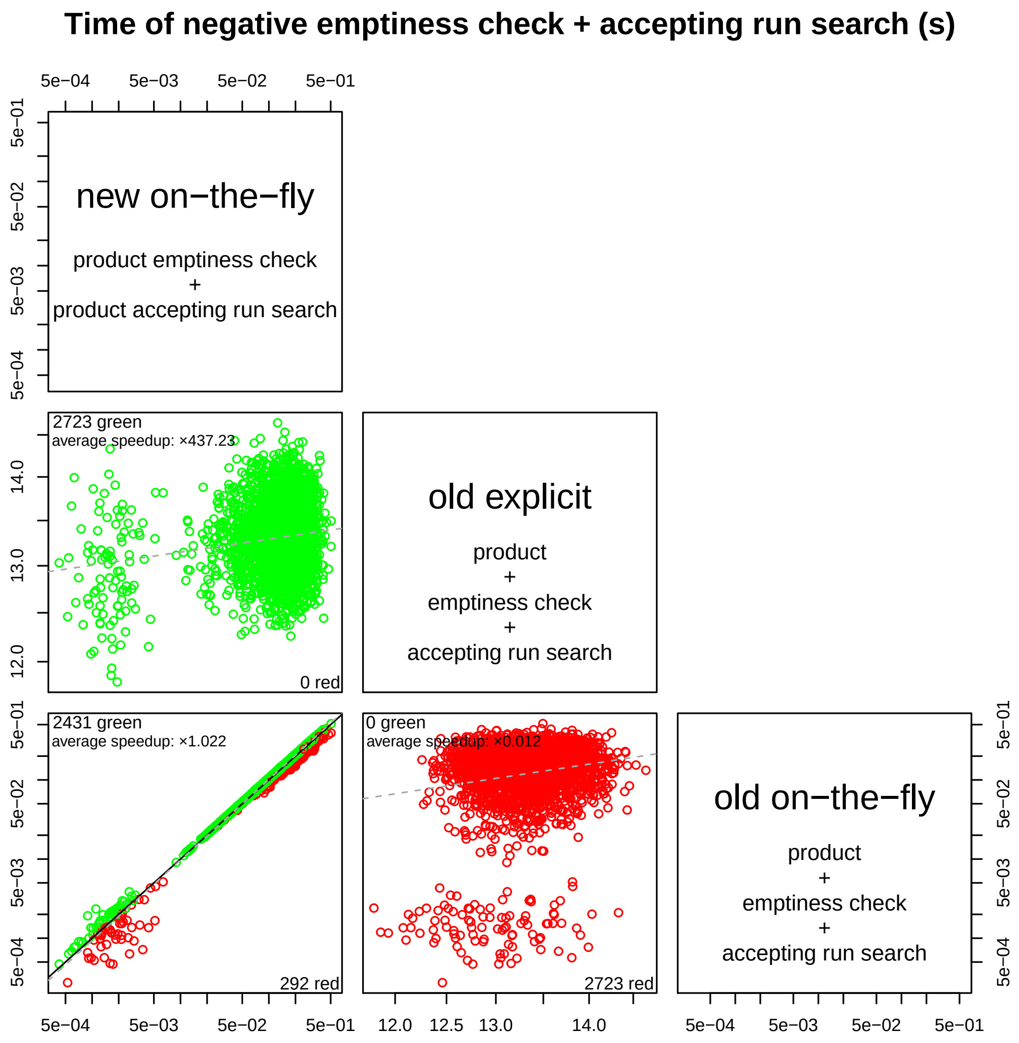

Figure 4.5 – Comparison of product, emptiness check and accepting run search on explicit automata (all axes logarithmic)

4.7.2 Use case: `ltlcross`

`ltlcross` is a tool shipped with Spot which compares LTL-to-ω-automata translators. You give it a set of commands, and a set of LTL formulæ, and it will check for the equivalence of the automata produced, while measuring several parameters like time spent translating or size of the automata produced. `ltlcross` only manipulates explicit automata.

Benchmark setup

We generated 100 random LTL formulæ with Spot's `randltl` tool with the command `randltl --tree-size=30 30`, and ran `ltlcross` with three LTL-to-ω-automata translators with the command `ltlcross -T 60 ltl2ba ltl3ba ltl2tgba`.

We then replaced the implementation of the cross checking of automata in `ltlcross` from an on-the-fly product and emptiness check to the product emptiness check, and ran the same command again.

In each case, we measured the time spent during the execution of the product and emptiness check.

Overall, 26 timeouts occurred during the translations, and 852 products and emptiness checks were computed.

Results

Figure 4.6 shows the individual times spent computing the products and emptiness checks. We can see an improvement of the performances of 50% on average, which can be atributed to the use of the explicit interface of the automaton, instead of the on-the-fly interface.

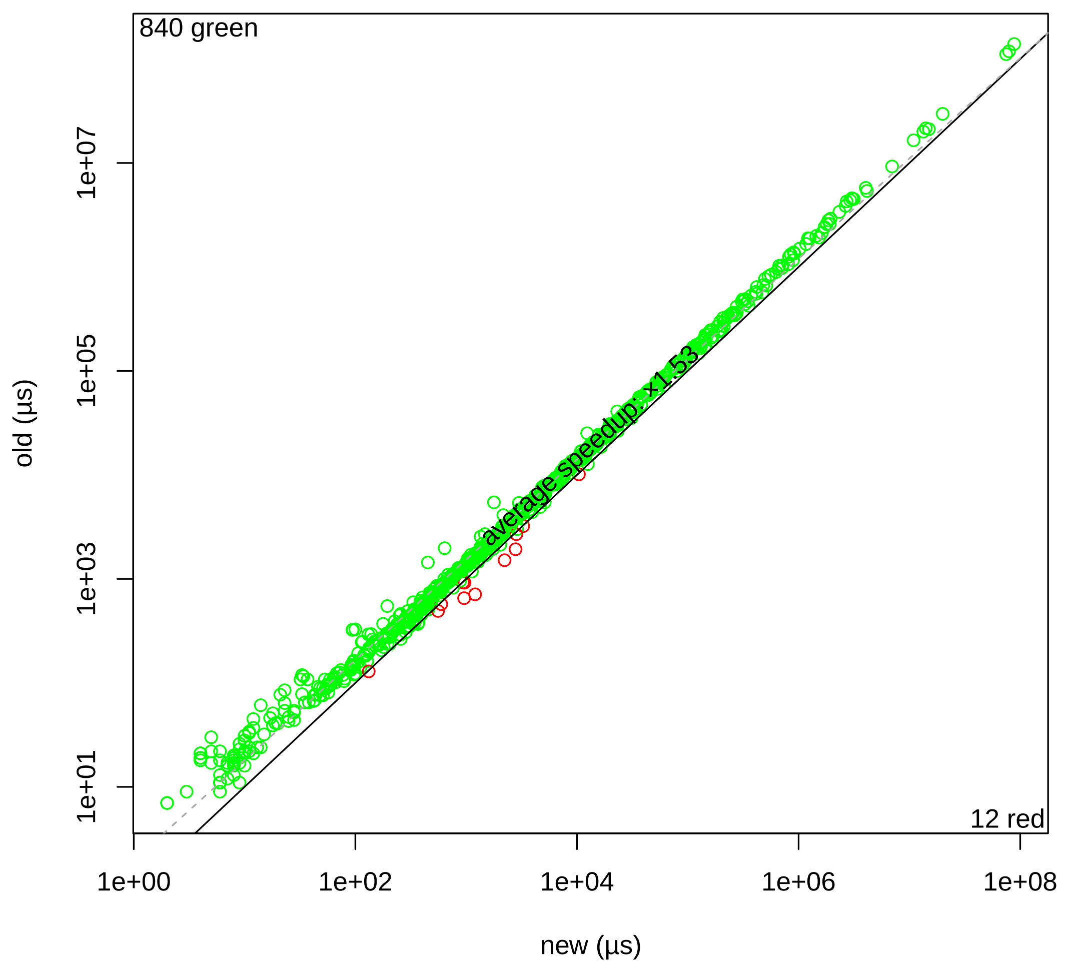

Figure 4.6 – Comparison of old vs. new product and emptiness check in `ltlcross`

Chapter 5

Generic counterexample search

Some emptiness check algorithms, like Couvreur's, are quite simple and efficient, but may be restricted to a certain type of ω-automata. The generic emptiness check is a new algorithm, which works with any acceptance condition, at the cost of a way higher complexity, as was shown by Emerson and Lei (1987).

5.1 Acceptance condition evaluation

Whenever we can conclude on whether we see an acceptance mark infinitely or finitely often, we may replace its occurrences in the acceptance condition by Boolean constants, which may in turn simplify the acceptance condition.

mark appears in...	**mark** is seen infinitely often	**mark** is seen finitely often
Inf(**mark**)	replace by $\top$	replace by $\bot$
Fin(**mark**)	replace by $\bot$	replace by $\top$

5.2 Introduction to the generic emptiness check

The first step to the generic emptiness check is an SCC analysis of the automaton, which will classify SCCs into three categories:

trivially accepting by considering the marks found in the SCC as seen infinitely often and the marks not found as seen finitely often, the acceptance condition evaluates to $\top$,

trivially rejecting by considering the marks which cannot be avoided in the SCC as seen infinitely often, the acceptance condition evaluates to $\bot$,

non trivial no conclusion can be made on the acceptance of the SCC without further analysis.

If an accepting SCC is found, then we can conclude that there exists an accepting run in the automaton. If not, we start by dropping all the trivially rejecting SCCs, and proceed to the rest of the analysis.

We consider each non trivial SCC individually as if it were a smaller automaton. After restricting the acceptance condition to the acceptance sets seen in the SCC we look at it in more details.

If it contains a mark such that we have to not see it to satisfy the acceptance condition, like in the form $\ldots \wedge Fin(\text{mark}) \wedge \ldots$, then we *cut* it from the automaton: we remove all transitions that bear this mark, such that we cannot see it in any way; we may then simplify the acceptance condition and restart from the first step by considering the cut SCC as an automaton.

In the third step, we look at the lowest priority operator of the acceptance condition. If it is a logical Or, we take each operand as an acceptance condition and try to run the second step; if any of them turns out to be accepting, we have found an accepting SCC.

The fourth step is to take any mark that is inside a Fin operator, cut it from the automaton, and see how it affects the acceptance condition with step one.

The last step is to assume the mark from step four is seen infinitely often, and start back from step two.

This algorithm works recursively, by working on smaller and smaller SCCs, and cutting the automaton and its acceptance condition, until it encounters a trivially accepting SCC.

5.3 Implementation details of the generic emptiness check

The generic emptiness check is actually divided into four parts:

`scc_info`, which analyses the SCCs,

`check_for_scc`, which runs the steps two to five,

`scc_split_check`, which cuts the SCCs by building transition filters and giving them to `scc_info`, and runs the first step on the got SCCs before giving them to `check_for_scc`,

`check_main`, which runs the first step on the automaton before giving the SCCs individually to `check_for_scc`.

Since all the information on how an SCC was considered accepting is gathered here, we decided to implement the accepting run search as a method of `scc_info`.

5.4 Implementation of the generic accepting run search

Like the product accepting run search, the generic accepting run search is heavily inspired from the `bfs_steps` algorithm of Spot. But since we need more control over the exploration than is possible with `bfs_steps`, we decided once again to reimplement a breadth-first search algorithm more suited to our needs.

5.4.1 Spot's data structures

The structures used include the ones described in Section 4.4.1, to which we add:

state identifier is a number uniquely identifying a state in an explicit automaton, from which we can get back all usual information about the state, and even an actual state, by passing it through explicit automata's method,

edge is a structure representing a transition in an explicit automaton,

SCC is a structure holding, among other things:

- the set of its states (by state identifier),
- the set of all marks that can be found in it,
- the edge filter that was used for its analysis.

5.4.2 Additional data structures

The following data structures are implemented for the generic accepting run search:

`exp_step` which is similar to Spot's step (see Section 4.4.1), except the source state is replaced by a state identifier.

5.4.3 Pseudocode

Algorithm 5.1 shows the pseudocode for the `explicit_bfs_steps` algorithm. Like Algorithm 4.1, it runs a breadth-first search from a given stating state to a matching state, determined by a given function *match*, while not exploring transitions as told by another function *filter*. This algorithm, while heavily inspired by `bfs_steps`, makes use of the explicit interface of automata, and allows filtering of transitions, which gives us the control we need for a correct accepting run search.

Algorithm 5.2 shows the pseudocode for the implementation of the generic accepting run search. It first checks if a prefix is needed by testing if the initial state of the automaton is not already in the SCC. If it is not, we set up the functions *matchSCC*, which will only return true if it encounters a state that is inside the accepting SCC, and *filterNone*, which does not filter anything. These functions are passed to `explicit_bfs_steps` to get us the prefix and the entry state of the accepting loop, *SCCstart*.

We then set up the functions *matchMarks*, which will iterate over the marks we need to see and then get back to *SCCstart*, and *filterSCC*, which will ensure we stay in the SCC and do not traverse any transition that would have been filtered out by the analysis, which we pass to `explicit_bfs_steps` for successive breadth-first searches to compute the accepting cycle.

```
Function explicit_bfs_steps
    Input: Automaton $A$, the state identifier start, functions match and filter, list of
           steps steps
    Output: The state identifier to which points the matching transition
    backlinks $\leftarrow$ map of state identifier to exp_step
    todo $\leftarrow$ queue of state identifiers
    todo.push(start)
    while todo is not empty do
        src $\leftarrow$ todo.pop()
        for edge $t \in A$.SuccessorsOf(src) do
            if not filter(t) then
                cur_step $\leftarrow$ exp_step(src, t.condition, t.marks)
                if match(cur_step, t.destination) then
                    path $\leftarrow$ stack of steps
                    Loop
                        path.push(A.AsState(cur_step.src), cur_step.condition,
                          cur_step.marks)
                        if cur_step.src = start then
                            break
                        cur_step $\leftarrow$ backlinks[cur_step.src]
                    while path is not empty do
                        steps.append(path.pop())
                    return t.destination
                if t.destination $\notin$ backlinks then
                    backlinks[t.destination] $\leftarrow$ cur_step
    return -1;
```

Algorithm 5.1: Pseudocode for the `explicit_bfs_steps` algorithm

```
Function GenericAcceptingRunSearch
    Input: Automata $A$, already allocated run $r$, trivially accepting SCC scc
    Output: $r$ is filled with an accepting run over $A$
    if $A$.initial $\in$ scc.states then
        substart $\leftarrow$ $A$.initial
    else
        matchSCC $\leftarrow$ lambda step, dst: return dst $\in$ scc.states
        filterNone $\leftarrow$ lambda edge: return false
        substart $\leftarrow$ explicit_bfs_steps($A$, $A$.initial, matchSCC, filterNone, $r$.prefix)
    SCCstart $\leftarrow$ substart
    to_see $\leftarrow$ scc.seen_marks
    matchMarks $\leftarrow$ Function
        Input: exp_step st, state identifier dst
        if to_see $= \emptyset$ then
            return dst = SCCstart
        if st.marks & to_see then
            // Remove st.marks from to_see
            to_see $\leftarrow$ to_see & not st.marks
            return true
        return false
    filterSCC $\leftarrow$ Function
        Input: edge $t$
        if $t$.destination $\notin$ scc.states then
            return true
        if scc.edge_filter $\neq$ None then
            return scc.edge_filter($t$)
        return false
    do
        substart $\leftarrow$ explicit_bfs_steps($A$, substart, matchMarks, filterSCC, $r$.cycle)
    while to_see $\neq \emptyset$ and substart $\neq$ SCCstart
```

Algorithm 5.2: Pseudocode for the generic accepting run search

Chapter 6

Conclusion

The product and generic accepting run search come to complete the panorama of algorithms that Spot provides. They both have their specificities, their usage, their needs, but came together as solutions to a common problem. There may however still be room for performance improvement in the product algorithms.

Some work needs to be done with the benching, to reduce the disparity of the sizes of the products used to test the product emptiness check and accepting run search, and to see how the generic emptiness check and accepting run search compare to other methods, such as converting the automaton to have a certain acceptance condition and then running algorithms specific to it.

It would be interesting to see if we could combine some of the mechanics behind each algorithm to not have such a huge tradeoff between genericity, capacity, and performances.

Chapter 7

Bibliography

Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., and Štill, V. (2017). Model Checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis (ATVA 2017)*. (page 5)

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 5, 7, 13, and 28)

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 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 6)

Emerson, E. A. and Lei, C.-L. (1987). Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306. (page 28)

Gastin, P. and Moro, P. (2007). Minimal Counterexample Generation for SPIN. In *Model Checking Software*. Berlin, Heidelberg. (pages 5, 11, and 12)

Gastin, P., Moro, P., and Zeitoun, M. (2004). Minimization of counterexamples in SPIN. In Graf, S. and Mounier, L., editors, *Proceedings of the 11th International SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *Lecture Notes in Computer Science*, pages 92–108. (pages 5 and 11)

Gillard, C. (2017). Two-automaton emptiness check in spot. Technical Report 1706, EPITA Research and Development Laboratory (LRDE). (pages 6, 13, 15, and 18)

Gillard, C. (2018). Two-automaton accepting run search in spot. Technical Report 1805, EPITA Research and Development Laboratory (LRDE). (pages 6, 7, 13, and 20)

Hansen, H. and Geldenhuys, J. (2008). Cheap and Small Counterexamples. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, Cape Town, South Africa. (pages 5 and 11)

Hansen, H. and Kervinen, A. (2006). Minimal Counterexamples in $O(n \log n)$ Memory and $O(n^2)$ Time. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, Turku, Finland. (pages 5 and 11)

Holzmann, G. J. (1997). The Model Checker SPIN. *IEEE Transactions on Software Engineering*, (5). (page 5)

Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., and van Dijk, T. (2015). LTSmin: High-Performance Language-Independent Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems*. (page 5)

Morris, R. (1968). Scatter storage techniques. *Communications of the ACM*, (1). (page 11)

Pnueli, A. (1977). The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA. (page 5)