Two-automaton emptiness check in Spot

Clément Gillard
(supervisor: Alexandre Duret-Lutz)

Technical Report n^o1708, June 2017
revision d63ad103

Spot is library that handles ω-automata whose acceptance conditions are expressed with at most 32 acceptance sets. Since the acceptance condition of the product of two automata has to use the sum of their sets, we cannot produce a product whose operands use more than 32 sets in total.
A typical operation on automata is to compute $L(A \times B) \neq \emptyset$ to know if $L(A)$ intersects $L(B)$. When it is implemented as $empty(product(A, B))$, the computation of the product throttles the amount of acceptance sets A and B can use.
We propose a new function $empty(A, B)$ which does the emptiness check of $A \times B$ without actually building an automaton and hence without any limit on the acceptance conditions. The **ltlcross** tool can now compare automata using a total of more than 32 acceptance sets.

Spot est une bibliothèque qui manipule des ω-automates dont les conditions d'acceptation sont exprimées avec au plus 32 ensembles d'acceptation. Puisque la condition d'acceptation du produit de deux automates doit utiliser la somme de leurs ensembles, on ne peut pas construire des produits dont les opérandes utilisent plus de 32 ensembles au total.
Une opération typique sur les automates est de calculer $L(A\times B) \neq \emptyset$ pour décider si $L(A)$ intersecte $L(B)$. Lorsqu'elle est implémentée par $empty(product(A, B))$, le calcul du produit limite le nombre d'ensembles d'acceptation que A et B peuvent utiliser.
On propose une nouvelle fonction $empty(A, B)$ qui réalise le test de vacuité de $A \times B$ sans construire un automate et donc sans limite sur les conditions d'acceptation. L'outil **ltlcross** peut maintenant comparer des automates pour un total supérieur à 32 ensembles d'acceptation.

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

1 Introduction **5**

2 Definitions and notations **7**
2.1 Notations on ω-automata 7
2.2 Strongly Connected Components 8
2.3 On-the-fly and Explicit automata 9
2.4 Kripke Structures 9

3 Implementation **10**
3.1 Couvreur's emptiness check 10
3.2 Problems with the existing implementation 10
3.3 Product elements 11
3.4 Data structures of the algorithm 12

4 The algorithm **13**
4.1 Pseudocode 13
4.2 Example 13

5 Improvements **23**
5.1 Explicit automata 23
5.2 Kripke structures 24
5.3 Strength of automata 24
5.4 Summary 25

6 Benchmarks **26**

7 Conclusion **29**

8 Bibliography **30**

Chapter 1

Introduction

Spot 2.0, presented in Duret-Lutz et al. (2016), is a C++ library which handles ω-automata, automata that run on infinitely long words whose acceptance is defined by an *acceptance condition* and *acceptance sets*. For implementation reasons, an automaton in Spot cannot have more than 32 acceptance sets.

The automata-theoretic approach to model checking is a classical way to do the formal verification of a system: from a model M and a property φ, we want to check that M validates φ. We start by expressing them as ω-automata and get A_M, the automaton representing M, and $A_{\neg\varphi}$, the automaton accepting only the words that do not verify φ. We then need to check if their languages intersect: since the intersection of their languages is the language of their product, we compute the synchronized product of A_M and $A_{\neg\varphi}$. Finally, we do an *emptiness check* over this product to test if its language is empty: if it is not, there exists an intersection between the languages of A_M and $A_{\neg\varphi}$, so there exists at least one word that is accepted by M but does not satisfy φ. This chain of algorithms is illustrated in Figure 1.1.

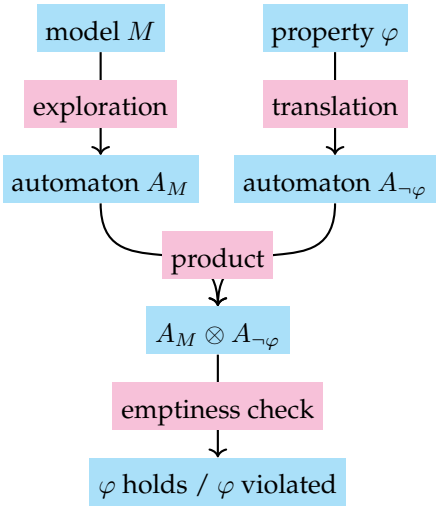


Figure 1.1 – The model checking toolchain

The check for the intersection of the languages of two automata is not only used for model checking, but also to check if two automata are equivalent. Spot's **ltlcross** tool, which benchmarks and compares LTL-to-automata translators, is constantly checking equivalences. Spot therefore provides the following implementation of this algorithm, both for model checking and its own tools.

$$empty(product(A, B))$$

This implementation is problematic since the product of A and B uses the sum of the acceptance sets of both automata: it means that we cannot compute the product if A and B use more than 32 acceptance sets in total. To remove this limitation, we implemented the *two-automaton emptiness check*, which computes the emptiness check of the product of two automata without computing the product itself. This work is a fusion of the existing implementations of the synchronized product and the single-automaton emptiness check, along with the various possible optimizations of these algorithms.

Chapter 2

Definitions and notations

This chapter defines various notations about the notions addressed in this report.

2.1 Notations on ω-automata

Definition 2.1 (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.*

The ω-automata manipulated in Spot are transition-based, which means their acceptance sets are sets of transitions.

Definition 2.2 (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.*

This allows for a more graphical way to represent acceptance sets, by showing the marks on their associated transitions, as shown in Figure 2.1.

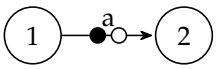


Figure 2.1 – Acceptance marks on a transition

Definition 2.3 (Acceptance condition) *An acceptance condition ϕ is a Boolean formula. The two-automaton emptiness check only handles Fin-less conditions, which means the acceptance condition must respect the grammar*

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

where x denotes an acceptance set.

$\mathit{Inf}(x)$ is true if we see infinitely often a transition of the set x, false otherwise.

We will denote acceptance sets by a symbol similar to their mark on the automaton, e.g. $\bullet$ or $\circ$.

Definition 2.4 (Transition-based ω-automaton) *A transition-based ω-automaton is a tuple $A = \langle Q, \Sigma, Q_0, \Delta, n, m, \phi \rangle$ where*

- Q *is a finite set of states,*
- Σ *is an alphabet,*
- $Q_0 \in Q$ *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,*
- $m : \Delta \mapsto 2^{[n]}$ *is a function returning the acceptance marks of a transition,*
- ϕ *is an acceptance condition.*

Definition 2.5 (Word) *Let Σ be an alphabet. A word is defined as $w \in \Sigma^\omega$, where ω is the smallest infinite ordinal.*

Automata take words as inputs, and either accept or reject them. The ω-automata-theoretic approach to model checking often uses Boolean conditions as letters to label the transitions.

Definition 2.6 (Run) *A run of an automaton is a sequence of transitions $\rho = (s_0, l_0, d_0)(s_1, l_1, d_1)\ldots$ where $s_0 = Q_0$ and for all $i \geq 0$ we have $d_i = s_{i+1}$ and $(s_i, l_i, d_1) \in \Delta$. We say that ρ recognizes the word $l_0 l_1 l_2 \ldots \in \Sigma^\omega$.*

Definition 2.7 (Accepting run) *$\Delta_i \subseteq 2^{[n]}$ is the set of marks that we see infinitely often along a infinite subsequence of transitions ρ_i of a run ρ. We take $\rho_i \vDash \phi$ to mean ρ_i satisfies ϕ and $\rho_i \nvDash \phi$ to mean ρ_i does not satisfy ϕ. The satisfaction of an acceptance condition is interpreted by induction as follows:*

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

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

Definition 2.8 (Accepting word) *An accepting word is a word recognized by an accepting run.*

Definition 2.9 (Language) *The language of an automaton A, denoted $L(A)$, is the set of all accepting words of A.*

The two-automaton emptiness check is used to check if the languages of two automata intersect: if it is negative, then there exists an intersection.

2.2 Strongly Connected Components

Definition 2.10 (SCC) *A Strongly Connected Component is a subautomaton of an ω-automaton where every state can be reached from every other state.*

Upon entering an SCC, a run can then stay in the same SCC.

Since the acceptance condition only contains sets that we want to see infinitely often, if an SCC contains a mark that satisfies it, then we know this SCC is accepting, and any run leading to it is an accepting run. Since finding an accepting run means there is an accepting word and that the language of the automaton is not empty, SCC-based emptiness-check algorithms look for accepting SCCs.

Definition 2.11 (Weak SCC) *An SCC is weak if all its transitions belong to the same acceptance sets.*

ω-automata can be classified according to their SCCs:

Definition 2.12 (Weak automaton) *An ω-automaton is weak if all its SCCs are weak.*

Definition 2.13 (Terminal automaton) *An ω-automaton is terminal if it is weak, its accepting SCCs are complete, and no accepting transition leads to a non-accepting SCC.*

Definition 2.14 (Strong automaton) *An ω-automaton is strong if it is neither weak nor terminal.*

2.3 On-the-fly and Explicit automata

Spot manipulates two types of ω-automata:

On-the-fly automata, or `twa`, uses virtual method calls to get and allocate data about itself on-the-fly. This allows for big procedural automata to be generated as they are traversed. This is especially useful in model checking, where the models we use may be too big to store at once in memory.

Explicit automata, or `twagraph`, is a subclass of `twa`. Its states and transitions are already known, so their handling is easier (e.g. states can be addressed with a number, instead of allocating and manipulating structures). It offers a more efficient explicit interface, but requires the whole automata to be loaded in memory at once. As it is a subclass, one could use the on-the-fly interface of an explicit automaton, but it would be less efficient.

2.4 Kripke Structures

Definition 2.15 (Kripke structure) *A Kripke structure is an automaton reading infinite words with conditions expressed on its states. It can be represented as a state-based ω-automata with acceptance condition $\top$ and no acceptance sets.*

Definition 2.16 (Fair Kripke structure) *A Fair Kripke structure is a Kripke structure with acceptance sets over its states and an acceptance condition. It can be represented as a state-based ω-automata.*

Kripke structures are often used in model checking to represent the state-space of the model to verify.

Since Spot manipulates transition-based ω-automata, Kripke and Fair Kripke structures are represented by pushing all conditions and marks from a state to its outgoing transitions, giving an equivalent transition-based ω-automata.

Chapter 3

Implementation

The two-automaton emptiness check is an algorithm based on two algorithms that were already implemented in Spot: the on-the-fly computation of the product of automata, and the on-the-fly emptiness check algorithm presented by Couvreur (1999). The latter has had multiple implementations and improvements in Spot; the one this work is based on was written in late 2016 and relies on templates to accommodate for its various optimizations: this allows for faster execution at the cost of an heavier binary, it is roughly 4 times faster than the previous implementation.

3.1 Couvreur's emptiness check

The algorithm presented by Couvreur (1999) is an SCC-based on-the-fly emptiness check. It looks for accepting SCCs to determine if the language of the given automaton is empty; as such, it does not need to explore the whole automaton to give a negative answer.

Couvreur's algorithm is an improvement of the SCC lookup algorithm by Tarjan (1972) with ideas taken from Dijkstra (1973) (Couvreur et al., 2005; Renault et al., 2013). It is considered as more efficient with the generalized Büchi ω-automata Spot manipulates (Gaiser and Schwoon, 2009; Schwoon and Esparza, 2005).

This algorithm works by doing a depth-first traversal of the automaton. For each new state we see, we create it its own SCC and store the marks seen in the transition from the last SCC. If we discover a transition leading to a state we already saw, it means we uncovered a loop, so all the states we have seen since belong to the same SCC, and the marks seen in that SCC also include the ones seen between the SCCs. If the marks seen in an SCC satisfy the acceptance condition, then the SCC is accepting and the emptiness check is negative. This algorithm does not need to store the transitions, just check them during the depth-first traversal, and store their marks. When backtracking from an SCC, we mark it as *dead* as we explored all its successors without finding an accepting SCC; if we encounter a dead SCC during our continuation of the traversal, we know not to explore it. States are identified by a unique number called their *order*. An SCC is linked to the order of the first state discovered in that SCC, called its *root*.

3.2 Problems with the existing implementation

The main goal of the implementation of the two-automaton emptiness check was to replace the following line in `twa::intersects`, a method meant to compute if the language of two

automata intersect:

```
return !otf_product(a, b)->is_empty();
```

This implementation, apart from the limit on acceptance sets seen in the introduction, has two problems:

`otf_product` this function builds an on-the-fly automaton from the on-the-fly interface of the two input automata. This is inefficient if we have explicit automata as inputs, as this does not differentiate types of automata, and uses the on-the-fly interface for both[1].

`is_empty` this redirects to an implementation of Couvreur's emptiness check algorithm, which dispatches to an optimized version for explicit automata. But since `otf_product` returns an on-the-fly automaton, it cannot use those optimizations.

We therefore needed a fully dispatched algorithm, that would use the explicit interface whenever an explicit automaton was given as either inputs.

3.3 Product elements

The synchronized product of two automata requires for each element of the product to correspond to two similar elements from each factor automaton. Since we have to compute the product without building an automaton, we cannot rely on storing them in the usual data structures used in Spot's automata; we have to implement our own structures. Most of these structures are pairs, or act as such.

`product_state` A pair of states. Provides an equality operator and a hash function for use in hash maps.

`product_mark` A pair of sets of marks[2]. Provides the union between two `product_marks`, as the `|=` operator, which does the union of the sets: with a, b, c, d sets of marks, $(a, b) \cup (c, d) = (a \cup c, b \cup d)$.

`product_iterator` A pair of transition iterators. Implements almost all operations of the usual transition iterators of Spot; very similar to the `twa_succ_iterator_product` of `otf_product`, except for:

- the `first` method, which sets the iterator to its first valid transition. To alleviate the code, we merged it in the constructor, so that my iterators are already on the first valid transition on instantiation.
- the `cond` method, which returns the condition to take the transition. The conditions of the factors' transitions are checked when looking for a valid transition in the `next` method which iterates to the next transition, but are not required for the emptiness check.

SCC structure A structure containing an integer for the order of the root, and a `product_mark` for the marks found in that SCC.

[1] Spot also implements `product` which builds the product of two explicit automata as an explicit automaton. This may explore more transitions and states than Couvreur's algorithm needs, so it is not suited as part of `twa::intersects`.

[2] In Spot, a `mark_t` is a set of marks; this structure is named in the same way.

3.4 Data structures of the algorithm

The two-automaton emptiness check implements Couvreur's emptiness check with an on the fly discovery of the synchronized product of two automata. Couvreur uses the following data structures:

- *Num*, an integer, to assign to each new SCC its order, starting from 1 and increasing after each assignment.
- *Hash*, a hash map from `product_state` to integer, which maps every discovered state to its order. We will use it to check if we already discovered a state, and also to mark states as part of a *dead* SCC by giving them an order of 0.
- *Root*, a stack of SCC structures, to store the discovered SCCs.
- *Arc*, a stack of `product_mark`s, to store the marks of the transitions between the SCCs of *Root*.

Along with these, our algorithm also uses the following data structures:

- *Todo*, a stack of pairs of a `product_state` and a `product_iterator`, for the depth-first traversal.
- *Live*, a stack of `product_state`s. It contains all the *live* states, states discovered during the depth-first traversal, that we will need to mark as *dead* once we backtrack from their SCC.

Chapter 4

The algorithm

4.1 Pseudocode

The two-automaton emptiness check is mostly a rewrite and improvement of Couvreur's algorithm. The pseudocode in Algorithm 4.1 (page 14) and Algorithm 4.3 (page 15) is an iterative approach of the recursive one in Couvreur (1999); the biggest change is in the introduction of Algorithm 4.2 (page 14) which monitors the exploration of the two automata.

4.2 Example

To illustrate the algorithm, let us consider the two automata in Figure 4.1 and apply the the two-automaton emptiness check step by step. The target product is in Figure 4.2 (page 16). The order in which the transitions are discovered is the alphabetical order.

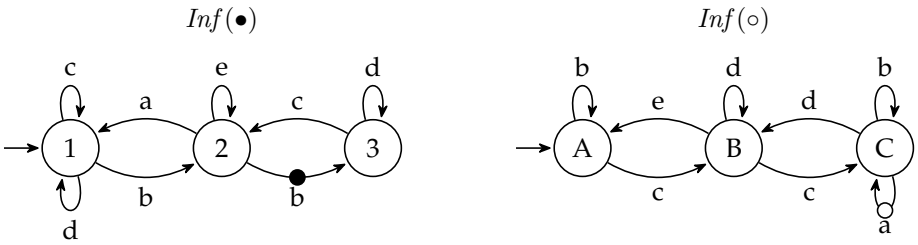


Figure 4.1 – The factors of this example

```
1  Function NewState
       Input: States s_A and s_B
       Output: A pair where the first field is a Boolean indicating if the product_state
               was already seen, and the second field is the order of the new state
2      p = ProductState(s_A, s_B)
3      (b, o) = Hash.get(p)
4      if not b then
5          o = Num
6          Hash.put(p, Num)
7          Root.push(Num, EMPTYPRODUCTMARK)
8          iter = ProductIterator(s_a.successors, s_b.successors)
9          Todo.push(p, iter)
10         Live.push(p)
11         Num = Num + 1
12     end
13     return (b, o)
14 end
```

Algorithm 4.1: Pseudocode for the state creation, used in Algorithm 4.3

```
1  Function NextSuccessor
       Input: Product iterator I
       Result: I is set to the next transition, or is marked as having no successors
2      (l, r) = I
3      while r has successors do
4          if l has successors then
5              move l to next transition
6          else
7              move l to first transition
8              move r to next transition
9          end
10         conjunction = l.label ∧ r.label
11         if conjunction is not equivalent to False then
12             I = (l, r)
13             return
14         end
15     end
16     mark I as having no more successors
17 end
```

Algorithm 4.2: Pseudocode for the transition exploration, used in Algorithm 4.3

```
1  Function TwoAutomatonEmptinessCheck
       Input: Automata A and B
       Output: A Boolean indicating if the languages of A and B intersect
2      Num = 1
3      NewState(A.Q_0, B.Q_0)
4      Arc.push(EMPTYPRODUCTMARK)
5      while Todo is not empty do
6          (curr_state, iterator) = Todo.top()
7          if iterator has no more successors then          // end of DFS, backtrack
8              Todo.pop()
9              (o_root, marks) = Root.top()
10             (b, o_curr) = Hash.get(curr_state)
11             if o_root = o_curr then        // curr_state is the root of an SCC
12                 repeat                   // mark all states in the SCC as dead
13                     s = Live.pop()
14                     Hash.put(s, 0)
15                 until curr_state = s
16                 Root.pop()
17                 Arc.pop()
18             end
19         end
20         (s_A, s_B) = iterator.destination
21         (b, o_new) = NewState(s_A, s_B)
22         if not b then
23             Arc.push(iterator.marks)
24         else if o_new ≠ 0 then                          // this state is not dead
25             (o_root, marks_root) = Root.pop()
26             marks = iterator.marks ∪ marks_root
27             while o_new < o_root do
28                 marks_arc = Arc.pop()
29                 (o_root, marks_root) = Root.pop()
30                 marks = marks ∪ marks_root ∪ marks_arc
31             end
32             Root.push(o_root, marks)
33             if A.φ.satisfy(marks) and B.φ.satisfy(marks) then
34                 return False
35             end
36         end
37         NextSuccessor(iterator)
38     end
39     return True                  // no accepting SCC was found
40 end
```

Algorithm 4.3: Pseudocode for the two-automaton emptiness check

$$Inf(\bullet) \wedge Inf(\circ)$$

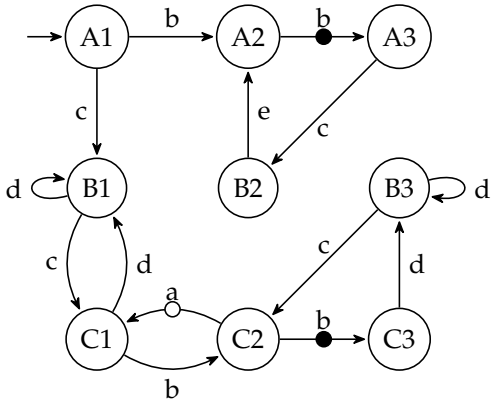
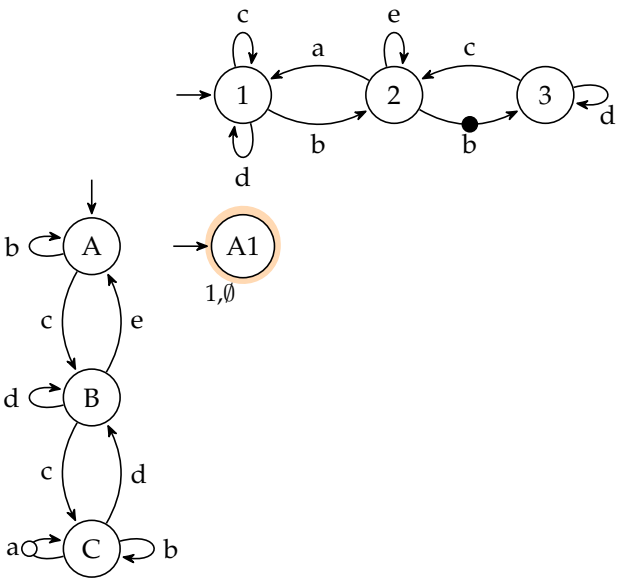


Figure 4.2 – The product computed during this example

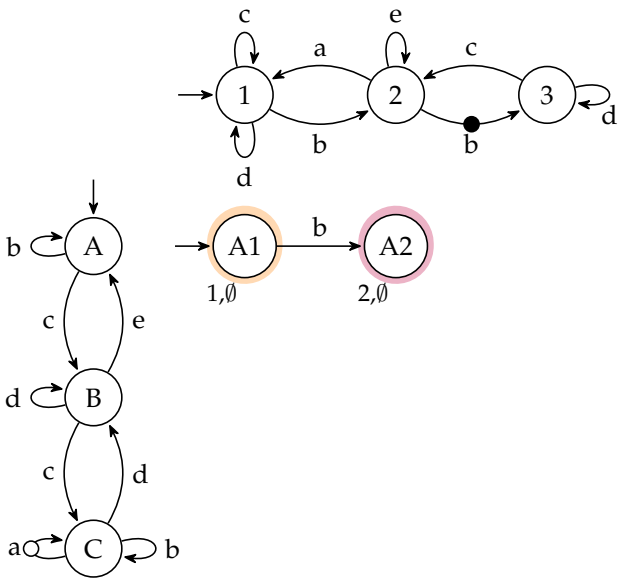


$Num = 2$

Root	*Arc*		*Todo*		*Live*	*Hash*
1	∅	∅	1→2	A→A	A1	A1: 1

$$Inf(\bullet) \wedge Inf(\circ)$$

We start by fetching the initial state, making it its own SCC. We set *Todo* to the first pair of transitions that is valid, here leading to $A2$ by reading a b.

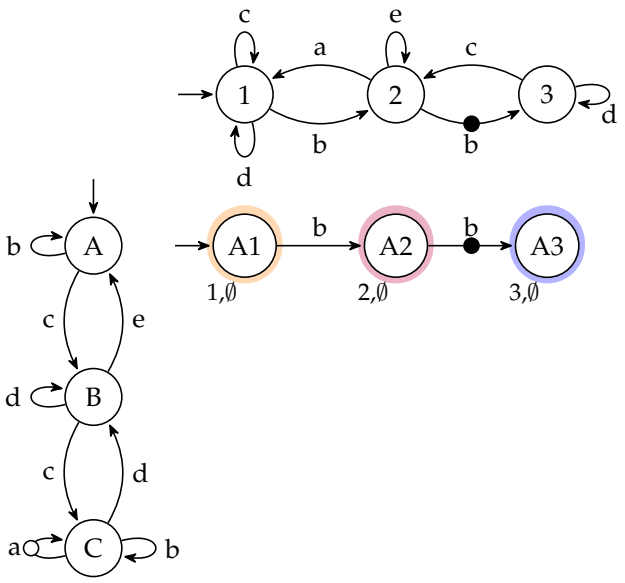


Num = 3

Root	*Arc*		*Todo*		*Live*	*Hash*
1	∅	∅	1→1	A→B	A1	A1: 1
2	∅	∅	2→3	A→A	A2	A2: 2

$$Inf(\bullet) \wedge Inf(\circ)$$

We take the transition, and make the top iterators of *Todo* point to the next valid transition. We discover $A2$: we add make it an SCC, update *Root* and *Live*, and push a pair of iterators on *Todo*.



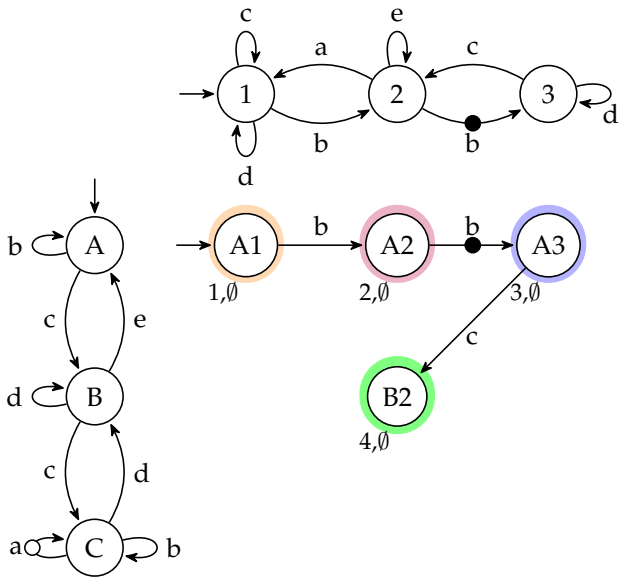
Num = 4

Root	*Arc*		*Todo*		*Live*	*Hash*
1	∅	∅	1→1	A→B	A1	A1: 1
2	∅	∅	2→×	A→×	A2	A2: 2
3	•	∅	3→2	A→B	A3	A3: 3

$$Inf(\bullet) \wedge Inf(\circ)$$

We discover $A3$, make it an SCC, and add a pair of its iterators to *Todo*. The mark we encountered by taking this transition is put on top of *Arc*. We iterate over all combinations of transitions from $A2$ without finding any other valid transition, the `product_iterator` is left

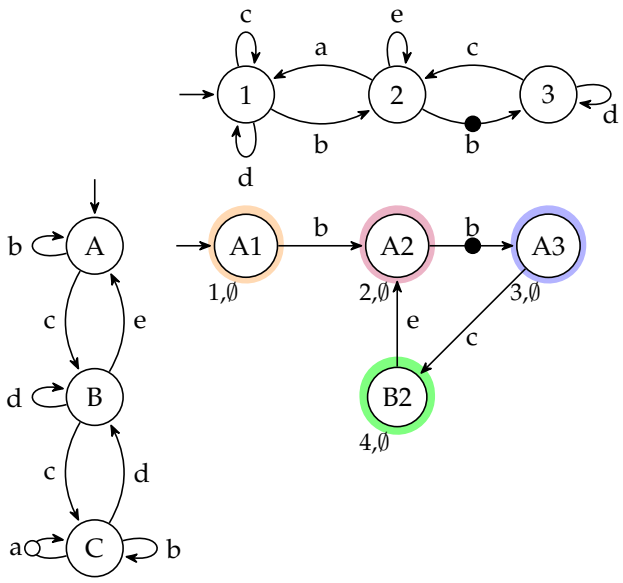
in a *done* state.



$Num = 5$

Root	*Arc*		*Todo*		*Live*	*Hash*
1	∅	∅	$1 \to 1$	$A \to B$	A1	A1: 1
2	∅	∅	$2 \to \times$	$A \to \times$	A2	A2: 2
3	•	∅	$3 \to \times$	$A \to \times$	A3	A3: 3
4	∅	∅	$2 \to 2$	$B \to A$	B2	B2: 4

$Inf(\bullet) \wedge Inf(\circ)$



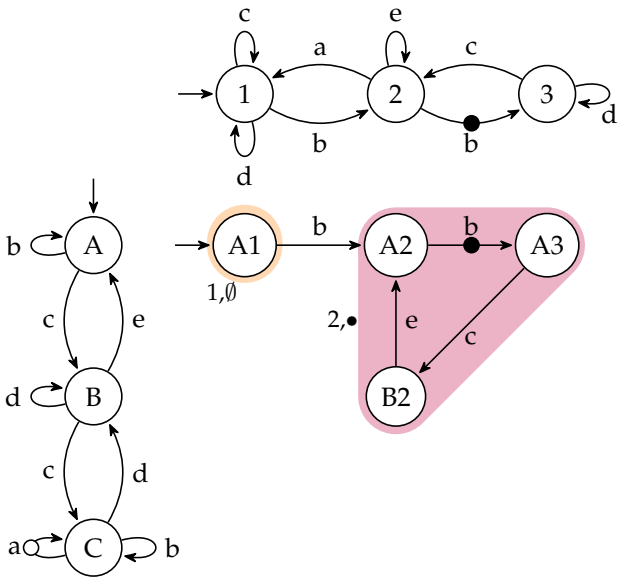
$Num = 5$

Root	*Arc*		*Todo*		*Live*	*Hash*
1	∅	∅	$1 \to 1$	$A \to B$	A1	A1: 1
2	∅	∅	$2 \to \times$	$A \to \times$	A2	A2: 2
3	•	∅	$3 \to \times$	$A \to \times$	A3	A3: 3
4	∅	∅	$2 \to \times$	$B \to \times$	B2	B2: 4

$Inf(\bullet) \wedge Inf(\circ)$

We get to $A2$: it already has an order in *Hash*, so we know we already discovered it. We detected a cycle between SCCs, so we take the order of $A2$ and pop and merge SCCs in *Root* until we find the one of same or lower order, the one $A2$ was in. Each time we pop *Root*, we also pop *Arc* and add the mark of the transition to the merged SCC. Note that the order of an SCC is

the order of its root.



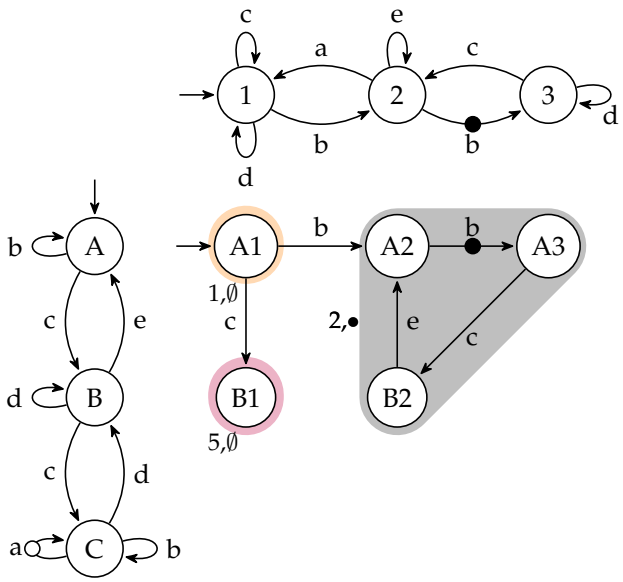
$Num = 5$

Root		*Arc*		*Todo*		*Live*	*Hash*
1		∅	∅	$1 \to 1$	$A \to B$	A1	A1: 1
2	•	∅	∅	$2 \to \times$	$A \to \times$	A2	A2: 2
				$3 \to \times$	$A \to \times$	A3	A3: 3

$$Inf(\bullet) \wedge Inf(\circ)$$

SCC #2 has mark •, this means that there exists an infinitely long word such that • marks are seen infinitely often; this is however not sufficient to satisfy our acceptance condition, which also needs to see ∘ marks infinitely often. We continue our depth-first traversal.

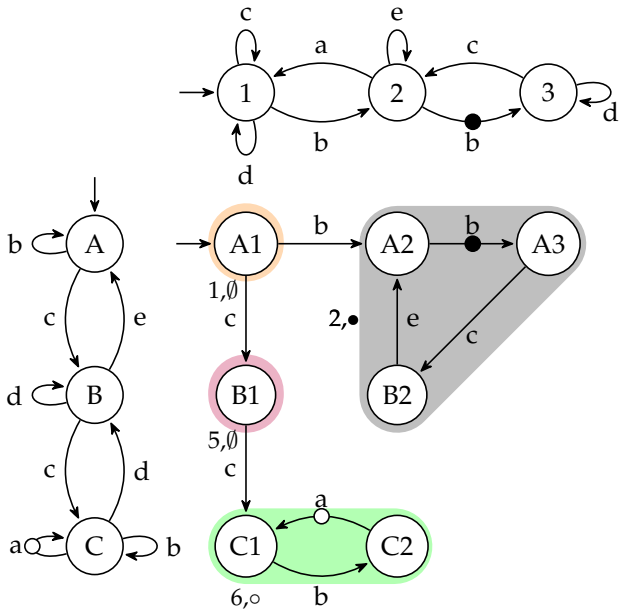
We now have *done* iterators on top of *Todo*. This means that we are done with a state, and need to backtrack in our depth-first traversal. When backtracking, we check that the order of the SCC on top of *Root* is not the order of the state we are backtracking from; otherwise it means we have explored our SCC entirely without finding any accepting word, we need to mark this SCC as dead. This is actually the case when we backtrack from $A2$: we look at *Live* and pop and mark as dead every state until $A2$, $A2$ included; we then pop *Root*. We continue back from A1.



$Num = 6$

Root	*Arc*		*Todo*		*Live*	*Hash*
1	∅	∅	$1 \to \times$	$A \to \times$	A1	A1: 1
5	∅	∅	$1 \to 1$	$B \to C$	B1	A2: 0
						A3: 0
						B2: 0
						B1: 5

$$Inf(\bullet) \wedge Inf(\circ)$$

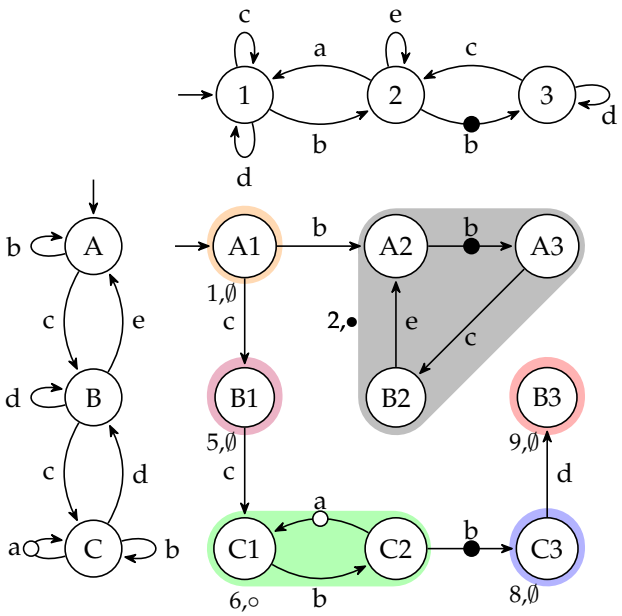


$Num = 8$

Root		*Arc*		*Todo*		*Live*	*Hash*
1		∅	∅	$1 \to \times$	$A \to \times$	A1	A1: 1
5		∅	∅	$1 \to 1$	$B \to B$	B1	A2: 0
6	∘	∅	∅	$1 \to 1$	$C \to B$	C1	A3: 0
				$2 \to 3$	$C \to C$	C2	B2: 0
							B1: 5
							C1: 6
							C2: 7

$$Inf(\bullet) \wedge Inf(\circ)$$

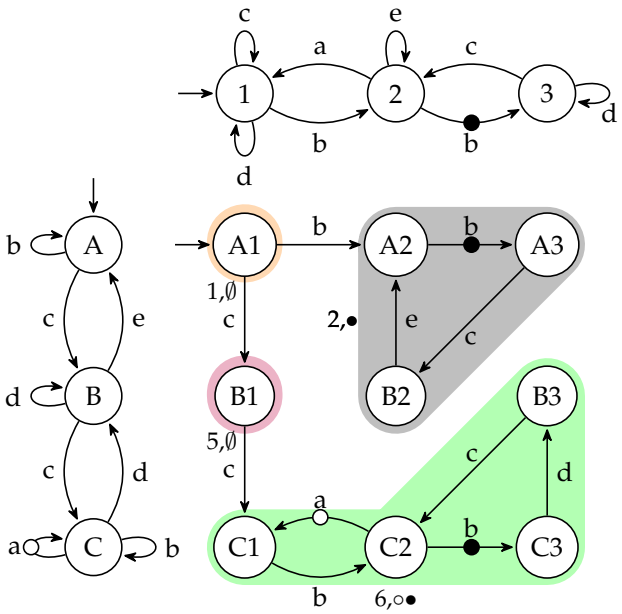
Upon discovering the $C2 \to C1$ transition, we find an SCC which contains a $\circ$ mark, so there exists a infinitely long word such that $\circ$ marks are seen infinitely often. But again, this is not enough to satisfy out acceptance condition.



$Num = 10$

Root	*Arc*	*Todo*		*Live*	*Hash*
1	∅ ∅	1 → ×	A → ×	A1	A1: 1
5	∅ ∅	1 → 1	B → B	B1	A2: 0
6 ○	∅ ∅	1 → 1	C → B	C1	A3: 0
8	• ∅	2 → ×	C → ×	C2	B2: 0
9	∅ ∅	3 → 3	C → C	C3	B1: 5
		3 → 2	B → C	B3	C1: 6
					C2: 7
					C3: 8
					B3: 9

At this point, we have discovered all possible states, but not all possible transitions.



$Num = 10$

Root	*Arc*	*Todo*		*Live*	*Hash*
1	∅ ∅	1 → ×	A → ×	A1	A1: 1
5	∅ ∅	1 → 1	B → B	B1	A2: 0
6 ○•	∅ ∅	1 → 1	C → B	C1	A3: 0
		2 → ×	C → ×	C2	B2: 0
		3 → 3	C → C	C3	B1: 5
		3 → 3	B → B	B3	C1: 6
					C2: 7
					C3: 8
					B3: 9

With transition $B3 \to C2$, we enrich SCC #6 with another cycle that contains the • mark. With that, this SCC contains both ○ and • marks: this satisfies our acceptance condition, we have found an accepting SCC, there exists an infinitely long word that sees ○ and • marks infinitely often, the language of the product of the two automata is not empty, the emptiness check is negative.

Note that although we discovered all the states, we did not discover all transitions in the product. This is because Couvreur's algorithm is an on-the-fly algorithm: it does not require full knowledge on or full exploration of the automaton it is executed on to find an accepting word.

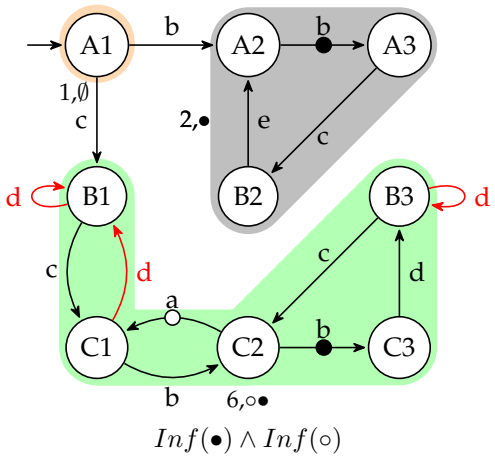


Figure 4.3 – The final computation of the SCCs if we did not stop.
The transitions we did not see are in red.

If we did not find an accepting SCC, we would have continued to explore all transitions and backtracked states we were done with, up to the initial state, resulting in Figure 4.3. Popping the initial state from *Todo* means we have completely explored the automaton without finding any accepting SCC; there cannot be a word that is accepted by this automaton, the emptiness check is positive.

Chapter 5

Improvements

The existing implementation has evolved to optimize the computation of the product and the emptiness check based on what can be known from the input automata. The last optimization of the emptiness check allowed for roughly 4 times faster execution when using explicit automata, by dropping dynamic method call resolution to the on-the-fly interface, and resolving them at compile time through templates; this, however, is done at the expense of compilation duration and binary size, which is something we would also like to improve. We did not allow ourselves to use virtual methods nor dynamic memory allocation, to avoid any overhead in time or size. This chapter presents the various optimizations we implemented, based on the ones in the existing algorithms the two-automaton emptiness check is based on.

5.1 Explicit automata

When an explicit automaton is given to the two-automaton emptiness check, we should not use its on-the-fly interface. This makes for a radical change in code factoring, since the data structures used are different in size and usage.

We create a class `tae_iterator` that is templated, and takes place of the regular iterators manipulated by `product_iterators`; a single `product_iterator` now handles two `tae_iterators`, which are templated independently. This class will interface an underlying explicit or on-the-fly transition iterator to get a common set of operations that the algorithms will be able to use. Since the algorithms already make use of the on-the-fly iterator's operations, we chose to adapt explicit iterators to this interface. While on-the-fly iterators are independent structures that provide all the operations needed, explicit iterators are much closer to the STL concept of iterators, and do not provide all the operations we need by themselves. As such, a `tae_iterator` templated as explicit will aggregate 3 iterators: the current one, on which operations are applied, the `begin()` iterator, to be able to reset the current iterator to the first transition, and the `end()` iterator, to be able to check if the current iterator has successors.

Since we do not use a lot of operations on states, the specialization for explicit automata went through static methods of a templated class, `tae_element`, which also gives information on types, to construct, hash, compare or destroy states.

This change would not be visible in the pseudocode shown in Section 4.1, since it is a change in the types aggregated by `product_iterators` and `product_states`.

Explicitness specialization templates are instanced in both possibilities, for both inputs, for a total of 4 specializations. This allows for faster execution and lower memory usage.

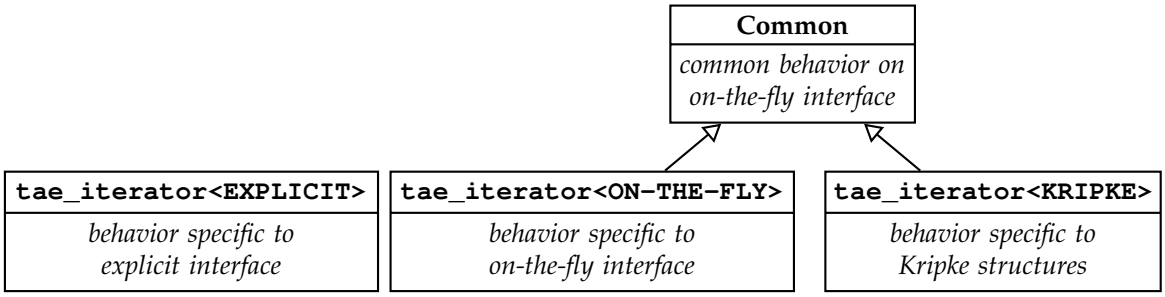


Figure 5.1 – UML diagram representing our solution to avoid dynamic dispatch and code duplication

5.2 Kripke structures

As seen in Section 2.4, Kripke structures are implemented in Spot by pushing the conditions and marks of the state-based ω-automatonto its transitions to get an equivalent transition-based automaton. This means that all outgoing transitions of a state will have the same label. We can use this property to our advantage by checking only once the label of the iterators in Algorithm 4.2, instead of at each iteration of the loop, thus reducing the amount of function calls.

The way we implemented this optimization was to have a third specialization on the automaton type, by having `tae_iterator<KRIPKE>` store the condition of the first transition on instantiation, and returning it every time it was asked for, instead of interfacing with the underlying iterator; being a very small method, calls to `cond` would get inlined as variable readings, and be optimized out of the checking loop.

The problem with this was that other than the condition retrieval, all behaviors are strictly identical with the on-the-fly specialization. Since virtual methods and dynamic dispatch were not allowed, we designed a superclass that would implement the common behaviors between the on-the-fly and Kripke specializations, which they would inherit from before implementing their own behaviors. This is illustrated in Figure 5.1. This allows for static resolution of tokens on template instantiation, avoiding a duplication of code and the need for virtual resolution.

This improvement *does* rely on a change in Algorithm 4.2, but we implemented it in a way that the compiler would be able to easily optimize the code itself; therefore, again, no change in the shown pseudocode.

Since Kripke structures are mostly used for models, we are unlikely to get two Kripke structures as inputs, so we save on template instances by only having Kripke structures as the first parameter, and swapping inputs if needed. This puts to 6 the number of specializations on the type of automata. Optimization on Kripke structures removes a recurrent virtual method call, which allows for faster execution.

5.3 Strength of automata

From the strength of automata seen in Section 2.2 we can deduce the following properties:

- In a weak automaton, the marks of a single transition in an SCC can tell us if the SCC is accepting, so we do not need to store the marks of the SCC.
- In a terminal automaton, once we encounter an accepting transition, we can state that there exists an accepting run.

These properties can be determined by the algorithm generating the automaton, specified by the user, or checked during the execution of another algorithm.

Even though optimizing for terminal automata represents a consequent optimization when running Couvreur's algorithm on a single automaton, on two automata, specializing the algorithm to stop running on the terminal one once we have found that it is accepting but continue running on the other automaton represents a too complicated change. In practice, computing the product with an accepting SCC of a terminal automaton adds no complexity. We chose to not implement optimizations for terminal automata, and rather focus on weak automata, since terminal implies weak.

With two automata, the different combinations are:

Strong-Strong This is the case we have been working with until now.

Weak-Weak We do not need the *Arc* stack, as we can retrieve the acceptance sets of the SCC as we go. SCC structures only have to contain their order, lightening *Root* a lot.

Weak-Strong We can use usual sets of marks instead of `product_mark`s, essentially halving the memory usage of *Arc* and reducing the size of SCC structures and therefore *Root*.

Strong-Weak As for Kripke structures, when no input is a Kripke structure, the inputs are swapped to get back to a **Weak-Strong** case, to reduce template instances. If there is a Kripke structure in input, then the **Strong-Strong** instance is used.

A problem arises when executing the algorithm: we may not need to store the marks, but we still need to retrieve and check them. That is where we introduce a distance between the data structures and the algorithm: the data structures may be reduced or even unused, but the code will still manipulate **Strong-Strong** `product_mark`s. We need to make templates of `product_mark` that are able to be converted from one specialization to the other, and are able to be unioned. Conversion shall lose the marks we do not need to store, while union shall give us a **Strong-Strong** `product_mark`.

5.4 Summary

Let us list the different instantiations of the two-automaton emptiness check.

For the first argument, all types of automata are instanced.

$$A_1 = \{explicit, on\text{-}the\text{-}fly, kripke\}$$

For the second argument, however, only explicit and on-the-fly automata are instanced.

$$A_2 = \{explicit, on\text{-}the\text{-}fly\}$$

Only 3 strength configuration are available.

$$strength = \{(strong, strong), (weak, strong), (weak, weak)\}$$

Let us now compute the number of instances.

$$instances = A_1 \times A_2 \times strength$$

$$|instances| = |A_1| \times |A_2| \times |strength| = 3 \times 2 \times 3 = 18$$

We have 18 different outcomes in our dispatch of automata. On the other hand, if we did not limit the possibilities on Kripke structures and strength combinations, this number would be $3 \times 3 \times 9 = 81$, which means that the final binary object would be more than 4 times its current size.

Chapter 6

Benchmarks

We generated 70 random automata with 900 states, acceptance conditions being random combinations of 16 acceptance sets, and transitions labeled as conditions over 10 different variables. The variables are shared between automata. We then generated the 2485[1] different possible pairs such that for any pair (a, b) there was not (b, a) but there was (a, a) and (b, b). We then compared the old and new implementations of `twa::intersects`.

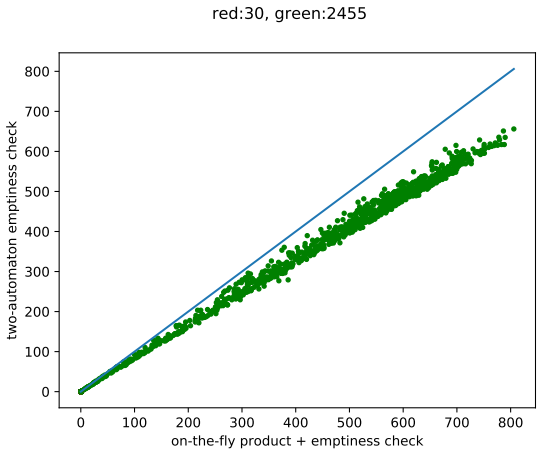


Figure 6.1 – Checking against the old implementation
In red, pairs where the old implementation was faster, in green where it was slower, in blue the identity line. Data are in seconds.

In Figure 6.1 we can see that in almost all cases the two automaton emptiness check was faster than the old implementation of `intersects`. The 30 cases where it was slower took under one second.

We did the same test under the same conditions with automata with 200 states and checked the two-automaton emptiness check against an explicit product and an emptiness check. The results are in Figure 6.2.

[1] $\binom{70}{2} + 70$ or $(70 + 2 - 1)!/2!/(70 - 1)!$, depending how you want to count.

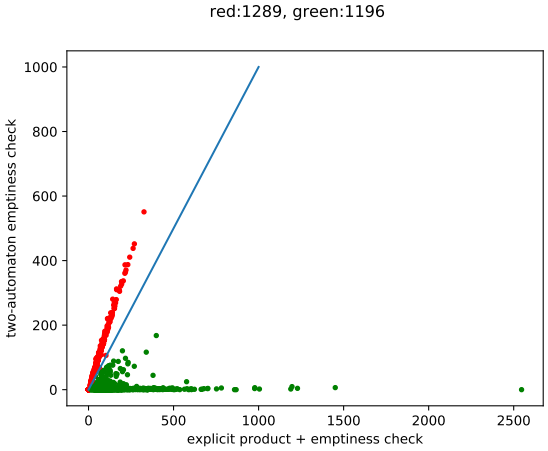


Figure 6.2 – Checking against an explicit product and an emptiness check
In red, pairs where an explicit product with an emptiness check was faster, in green where it was slower, in blue the identity line. Data are in seconds.

As expected, even though we reduced the size of the automata, we saw a higher consumption of memory due to the explicit product exploring all possible transitions and states where the two-automaton emptiness check computes them as needed. However, we can see that some tests are faster with the explicit product, when about as many are faster with the two-automaton emptiness check, in a well spread fashion. We will need to explore this further.

Strength is a factor easier found in smaller automata, so this time we generated 100000 pairs of random automata with 10 states and conditions expressed with 3 variables, and the same acceptance conditions as before. We then checked their strength.

We can again see in Figure 6.3 that in a vast majority our implementation is faster by a seemingly constant factor.

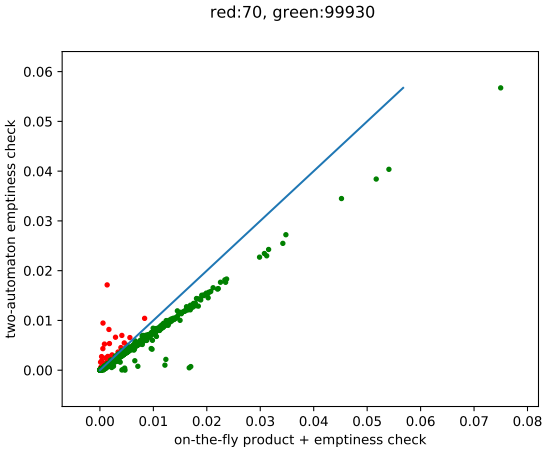


Figure 6.3 – Checking small automata with known strength against the old implementation
In red, pairs where the old implementation was faster, in green where it was slower, in blue the identity line. Data are in seconds.

Chapter 7

Conclusion

We wanted to remove the limitations of the previous way to check if the intersection of the languages of two automata was empty, while keeping all the optimizations of the existing algorithms. Our new implementation, besides lifting the limitation on the number of acceptance sets of the inputs, is heavily templated to account for various cases of time and space optimizations. We saw that this new implementation is faster than the previous one, while allowing for more optimization cases.

On top of proving the existence of an accepting word in the product of two automata, we would then like to be able to provide such a word: it would be a counterexample of the model checking, a word that is accepted by the model but violates the property. This requires to implement a second traversal of the automata to look for a run going to and cycling in the accepting SCC we found.

Chapter 8

Bibliography

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 10 and 13)

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 10)

Dijkstra, E. W. (1973). EWD 376: Finding the maximum strong components in a directed graph. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF. (page 10)

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)

Gaiser, A. and Schwoon, S. (2009). Comparison of algorithms for checking emptiness on büchi automata. In Hliněný, P., Matyáš, V., and Vojnar, T., editors, *Proceedings of the 5th Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, Znojmo, Czech Republic. (page 10)

Renault, E., Duret-Lutz, A., Kordon, F., and Poitrenaud, D. (2013). Three SCC-based emptiness checks for generalized Büchi automata. In McMillan, K., Middeldorp, A., and Voronkov, A., editors, *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'13)*, volume 8312 of *Lecture Notes in Computer Science*, pages 668–682. Springer. (page 10)

Schwoon, S. and Esparza, J. (2005). A note on on-the-fly verification algorithms. In Halbwachs, N. and Zuck, L., editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, Edinburgh, Scotland. Springer. (page 10)

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160. (page 10)