Simulation-based Reductions for TGBA

Thomas Badie

Technical Report n^o1214, January 2013
revision 2439

The Automata-Theoretic approach to model checking traditionally relies on Büchi Automata (BA) which we want as small as possible. Spot, a model-checking library, uses mainly a BA generalization: TGBA. We have already presented a simulation reduction (called *direct*) that works on TGBA. This algorithm is included in Spot 0.9 and led to produce smaller automata than in the previous versions of Spot.

The simulation consists in merging states that recognize the same infinite suffixes. We show that we can also work on infinite prefixes (it is called *cosimulation*), and that we can iterate these two simulations to create the *iterated simulation*. This iteration-based simulation, included in Spot 1.0, is a clear improvement over our previous simulation procedure.

We finally experiment a method that consists in considering some acceptance conditions as *don't care*. Since the acceptance conditions on transitions that are not on a Strongly Connected Component have no influence on the language, we can change them to help the simulations.

L'approche par automates du model checking s'appuie traditionnellement sur des Automates de Büchi (BA) qu'on souhaite les plus petits possible. Spot, bibliothèque de model checking, utilise principalement une généralisation des BA : les TGBA. Nous avons déjà présenté une méthode de réduction par simulation (dite *directe*). Cette technique a permis de produire des automates plus petits que dans les précédentes versions de SPOT.

La simulation consiste à fusionner les états ayant le même suffixe infini. Nous montrons que nous pouvons aussi fusionner ceux ayant le même préfixe infini (c'est la *cosimulation*). On peut répéter la simulation et la cosimulation pour créer la *simulation itérée*. Cette méthode est incluse dans Spot 1.0 et elle constitue une grande amélioration de la simulation.

On expérimente aussi une méthode qui consiste à modifier certaines conditions d'acceptations (appellées *sans importances*). Puisque celles qui sont sur les transitions entre composantes fortement connexes n'ont pas d'influence sur le langage, on peut les modifier pour aider la simulation.

Keywords
automata, BDD, bisimulation, model checking, reduction, simulation, TGBA

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
badie@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Introduction

The automata-theoretic approach to model checking uses Büchi automata to express properties. Model checkers that use this approach want these automata to be as small as possible to save time and memory. So, if one can have two automata that represent the same property, let's assume A and B with B is larger than A (in states and in transitions), an algorithm will run faster and use less memory on A than on B for the same result.

There is a kind of methods to reduce an automaton are simulation-based reductions. These methods consist in eliminating redundancy in an automaton. The bisimulation consists in merging states that recognize the same languages. The direct simulation removes a state if the language recognized by this one is included by the one recognized by another state: one can keep only the one which recognizes the larger language. Etessami and Holzmann (2000) present an adaptation of this algorithm for the Büchi Automata. There are several other algorithms that are derived from the simulation: the Reverse Simulation (Somenzi and Bloem, 2000), the don't care Simulation (Somenzi and Bloem, 2000). These two methods are meant to work on BA and we present an adaptation for the TGBA. There is also the Iterated Simulation that is presented later in this report.

SPOT is a C++ library that relies on the automata theoretic approach to model checking. Figure 1 presents an overview of the automata theoretic approach. On the left branch of the tree is represented the model to be checked. It can be represented with several kinds of structures. A Kripke structure can be used, and this structure can be very large for complex systems. On the right branch is represented the property to verify on the model. The negation of the property is then translated into a Büchi automaton.

To check if the model verifies the property, we compute the synchronized product between these two automata, and we check its vacuity. If there is an accepting path, the property is not verified and we can exhibit a counterexample, otherwise the property is verified. The product automaton can become a really huge automaton, this is called combinatorial explosion.

To represent properties, SPOT uses Linear-time Temporal Logic formulae (presented in Section 1.1) that are translated into a Transition-based Generalized Büchi Automata (TGBA). TGBA are a BA generalizations. SPOT uses TGBA because TGBAs have a more concise representation (Duret-Lutz, 2011).

Badie (2011) presents a generalization of the bisimulation for TGBA. The bisimulation consists in merging all the states that recognize exactly the same suffixes. A suffix for a state in a TGBA is the label of the transitions and the acceptance conditions on them.

Badie (2012) presents a generalization of the simulation for TGBA. In this report, we still present how to generalize the direct simulation. The direct simulation consists in merging two states when all the suffixes seen by a state are included by the one seen by the other. In fact, the bisimulation (suffixes are equals) is a sub-case of the simulation because to merge two states A and B, all the suffixes seen by A must be seen by B and vice versa.

We also present several algorithms that extend of the Simulation: the Reverse Simulation

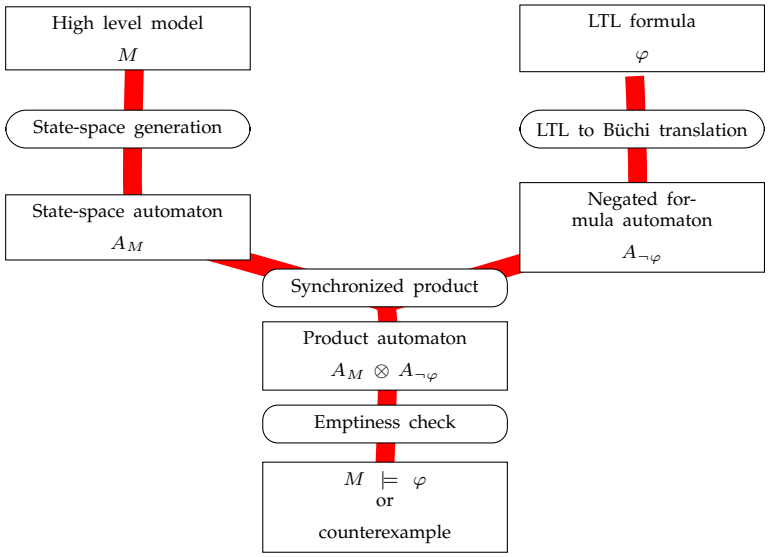


Figure 1: Automata-theoretic approach to LTL model checking.

(Somenzi and Bloem, 2000), the iterated Simulation and the don't care Simulation (Somenzi and Bloem, 2000).

Then we show how these algorithms improved the performance of SPOT on TGBAs and BAs.

Acknowledgment

I would like to thank Alexandre Duret-Lutz for his help and his supervision. I also want to thank Victor Lenoir, David Moreira and Pierre Parutto for their activity.

Chapter 1

Preliminaries

This chapter introduces the basic knowledge needed to understand this report: LTL formulae and why they are so important in the context of the model checking; Binary Decision Diagrams and a short presentation of the library we are using to represent them. Then we present the definitions of an ω-automaton, of a Büchi Automaton, which are used to represent an LTL formula, and of a Transition-based Generalized Büchi Automaton, which generalizes the BA. We then see that translating an LTL formula into a TGBA produces smaller automata than with the BA representation. Then we introduce SPOT, a C++ model-checking library, and present some of its characteristics.

1.1 Linear-time Temporal Logic

The goal of model checking is to verify whether the model is correct. The first order Boolean logic is not expressive enough to represent what we want. For example, with the propositional logic, we cannot express a time-related condition. To solve this problem, one can use Linear-time Temporal Logic formulae. Commonly called LTL formulae, they allow us to represent logical formulae with a time component. In fact, we can say: 'Once in the future, this will be true'.

Several new operators are added to the basic operators ($\vee, \wedge, \neg$):

G 'Globally'. $\mathsf{G}\,a$ means every time, a.

F 'Finally'. $\mathsf{F}\,a$ means at least once in the future, a.

X 'Next'. $\mathsf{X}\,a$ means next time, a.

U 'Until'. $a\,\mathsf{U}\,b$ means a until b.

This is a non exhaustive list of operators, but they are the base of the LTL formulae (with U and X). The other operators can be built with the "Next" and the "Until" operators. Now, we have a more powerful expressiveness.

Example "$\mathsf{F}(\neg(a \wedge b))$" means at least once in the future, "a and b" will be false.

LTL formulae are used to express properties. We translate them into automata. Because a run of the model is infinite, we have to use ω-automata: Büchi Automata.

A reference article about LTL formula is Emerson (1995).

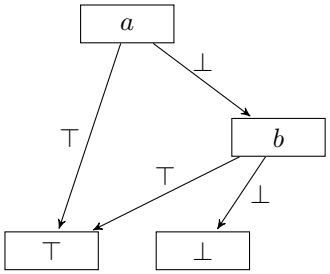


Figure 1.1: The BDD representing $(a \vee b) \wedge (a \vee b \vee c)$

1.2 Binary Decision Diagrams

For a complete explanation about Binary Decision Diagrams, we recommend Bryant (1986).

The problem of representing Boolean functions is a complex one. Indeed, some operations, such as the equivalence, are NP-Complete or coNP-Complete problems. Binary Decision Diagrams (BDD) are a way to represent Boolean functions, and to manipulate them simply. A BDD is an acyclic graph which represents a Boolean function in a canonical form. Bryant (1986) shows that using a BDD can reduce every graph representing a function to a canonical form graph in a linear time. Its representation allows to make useful operations (equivalence, satisfiability, ...) simpler.

In SPOT, we use the BuDDy Library (Lind-Nielsen, 2002). It provides some useful facilities: operator | and & are redefined; it allows to represent different formulae in a smaller way:

Example Let a, b and c be three Boolean variables. If the BDD represents the function $f(a, b, c) = (a \vee b) \wedge (a \vee b \vee c)$, the BDD represents it as $a \vee b$. This representation is shown in Figure 1.1.

Figure 1.1 shows that $a \vee b$ can be represented as $a \vee (\neg a \wedge b)$, because there are two paths to $\top$: a and $(\neg a \wedge b)$. There is a redundancy in the $\neg a$. Because of our usage of this data structure, we must avoid them. We want to have a BDD into an irredundant sum of product $(a \vee b)$. We use an algorithm from Minato (1992) to that effect.

1.3 ω-Automata

An ω-automaton is an automaton that recognizes words of infinite length. We define a *run* as an infinite sequence of states. A *run* is accepting if it satisfies some acceptance conditions.

1.3.1 Büchi Automata

Definition 1 *A Büchi automaton can be defined as a tuple* $A = \langle \Sigma, \mathcal{Q}, \mathcal{F}, q^0, \delta \rangle$ *where:*

- Σ *is an alphabet,*
- $\mathcal{Q}$ *is a finite set of states,*
- $\mathcal{F} \in \mathcal{Q}$ *is a finite set of acceptance states,*
- $q^0 \in \mathcal{Q}$ *is the initial state,*
- $\delta \in \mathcal{Q} \times \Sigma \to \mathcal{Q}$ *is the transition function.*

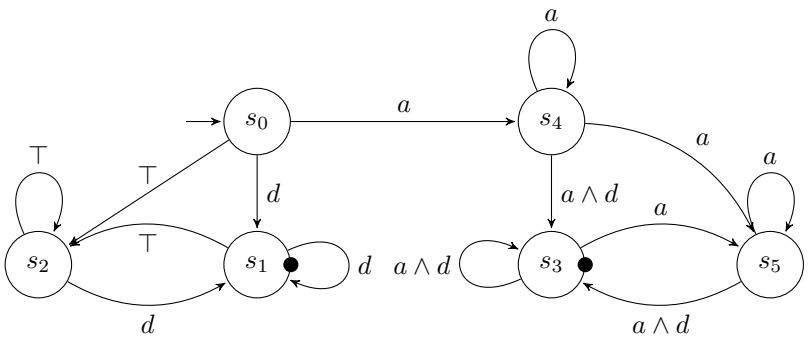


Figure 1.2: A BA representing $\mathsf{G}\, a\, \mathsf{U}\, \mathsf{G}\, \mathsf{F}\, d$ without any simplification.

Let $\mathcal{S}$ be a set of states that appears infinitely often in the run ρ. A run ρ of a Büchi automaton over an infinite word is *accepting* if and only if $\mathcal{S} \cap \mathcal{F} \neq \emptyset$, in other words, when some accepting states appear in ρ infinitely often. In this report, accepting states are represented with ● (If there are several acceptance conditions, we use different colors: ○, ●, ...). In a BA or a TGBA, $\top$ accepts every letters.

This kind of automaton can be called State-based Büchi Automaton or just a Büchi Automaton. Figure 1.2 represents a BA. In this case:

- Σ is $\{a, d\}$,
- $\mathcal{Q}$ is $\{s_0, s_1, s_2, s_3, s_4, s_5\}$,
- $\mathcal{F}$ is $\{s_1, s_3\}$,
- q^0 is s_0,
- δ is $\{(s_0, \top, s_2), (s_0, d, s_1), \ldots\}$.

An accepting run in this automaton could be an infinity of d or an alternation of d and a. A non accepting run could be an infinity of a.

1.3.2 Transition-based Generalized Büchi Automata

A Transition-based Generalized Büchi Automaton (TGBA) is a Büchi Automaton where acceptance conditions are not on states, but on transitions and that has a set of acceptance conditions instead of a unique acceptance condition. So the definition given in Definition 1, must change a little. We redefine δ and the acceptance condition. The new definition is:

Definition 2 *A Transition-based Generalized Büchi automaton can be defined as a tuple* $A = \langle \Sigma, \mathcal{Q}, \mathcal{F}, q^0, \delta \rangle$ *where:*

- Σ *is an alphabet,*
- $\mathcal{Q}$ *is a finite set of states,*
- $q^0 \in \mathcal{Q}$ *is the initial state,*
- $\delta \subseteq \mathcal{Q} \times \{2^\Sigma \setminus \{\emptyset\}\} \times 2^\mathcal{F} \times \mathcal{Q}$ *is the* translation relation, *where each transition carries a nonempty set of letters of the alphabet and a set of acceptance conditions.*

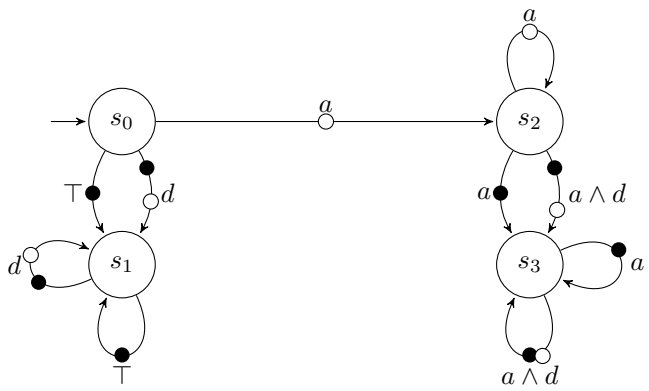


Figure 1.3: A TGBA representing $\mathsf{G}\, a\, \mathsf{U}\, \mathsf{G}\, \mathsf{F}\, d$ without any simplification.

- $\mathcal{F}$ *is a set of* acceptance conditions.

This modification involves changing the definition of an accepting run. A run in a TGBA is accepting if there exists an infinite sequence ρ of transitions of δ, where each transition matches the corresponding letter of the word, and ρ visits each acceptance condition infinitely often.

Duret-Lutz and Poitrenaud (2004) showed that translating LTL formulae into a TGBA is better than translating them into a Büchi automaton in term of number of states and transitions. It is a good idea to use a representation which leads to manipulate smaller automata because the computation on these automata will necessarily be faster. Figure 1.3 represents a TGBA. In this case:

- Σ is $\{a, d\}$,
- $\mathcal{Q}$ is $\{s_0, s_1, s_2, s_3\}$,
- $\mathcal{F}$ is $\{\bigcirc, \bullet\}$,
- q^0 is s_0,
- δ is $\{(s_0, a, \bigcirc, s_2), (s_0, d, (\bullet, \bigcirc), s_1), \ldots\}$.

Note 1 *Acceptance condition on transitions that are not in a Strongly Connected Component can be added or removed because they don't have any impact on whether a word is accepted. For example, the acceptance condition* $\bigcirc$ *on the transition which links* s_0 *to* s_2 *can be removed without impacting the language recognized by this automaton.*

1.3.3 Promise Automata

A Promise Automaton is the dual of a TGBA. We take all the acceptance conditions, and for each transition of the automaton, we replace the ones on the transition by a promise to see the other acceptance conditions.

If we have an automaton $\mathcal{A} = \langle \Sigma, \mathcal{Q}, q^0, \delta, \mathcal{F} \rangle$, its dual is $\mathcal{A}' = \langle \Sigma, \mathcal{Q}, q^0, \delta', \mathcal{F} \rangle$ where $\delta' = \{(s, l, \mathcal{F} \setminus Acc, d) \mid (s, l, Acc, d) \in \delta\}$.

For example, if we take the TGBA represented in Figure 1.3, and if we translate it into a Promise Automaton, we will get the automaton presented in Figure 1.4.

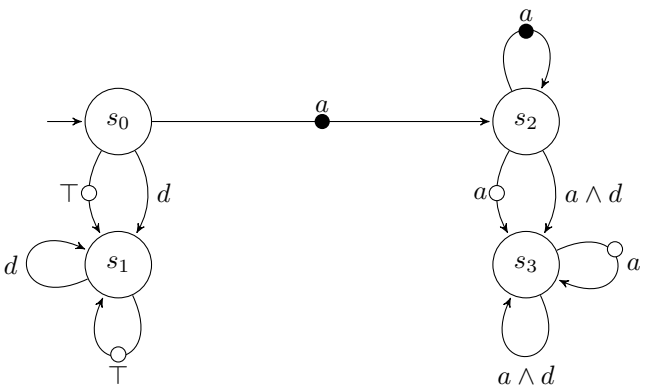


Figure 1.4: A Promise Automaton representing G a U G F d without any simplification.

Note 2 *The acceptance conditions of a TGBA are encoded with a BDD variable and represented with ●. In the aim to use as few BDD variables possible, and because promises and acceptance conditions will never be used at the same time, we chose to reuse these variables: Promises[●] is encoded with ●.*

We have to change the definition of an accepting run. A run in a TGBA is accepting if there is no infinite sequence ρ of transitions of δ, where each transition matches the corresponding letter of the word, and ρ sees the same promise continuously.

An accepting run in this automaton is an accepting run in the Promise Automaton, so an infinity of d or an alternation of d and a are accepting.

1.4 SPOT

1.4.1 A Brief Overview

SPOT (SPOT Produces Our Traces) is a C++ library that provides several tools to build model checkers. In fact, a model checker built on top of SPOT can be separated into three parts.

- The first part consists in translating LTL formulae (presented in Section 1.1) into automata.
- The second part is the computation of the product of the negated formula, and of the state space. Followed by the emptiness check, and the result.
- The third part is a front end for translating models into automata. This part is not a part of SPOT. It belongs to the user to provide it.

We can see the structure of a model checker built on the top of SPOT in Figure 1.

The branch on the left is the front end that represents the model. The part that catches our attention in this report is the one after the translation of the formula into automaton, and before the computation of the synchronized product.

1.4.2 SPOT uses TGBA

SPOT uses TGBA (as presented in Section 1.3.2) to express formulae as automata, because there are some implementation advantages at different levels. Not only because it leads to smaller au-

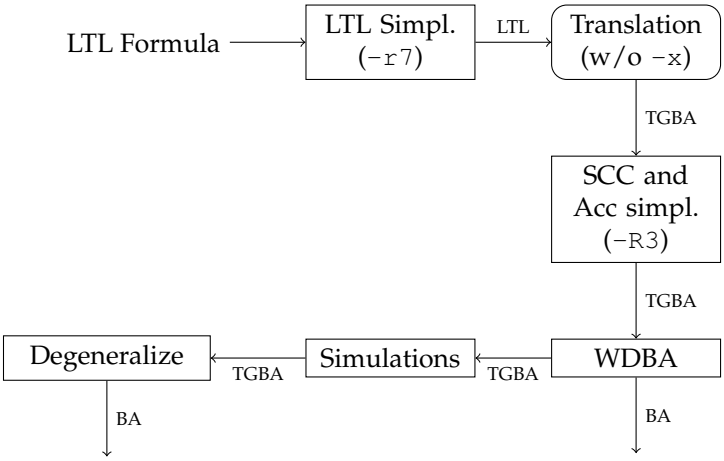


Figure 1.5: The SPOT's LTL-to-Büchi toolchain.

tomata, but also because its interface allows to compute some on-the-fly algorithms (Couvreur, 1999).

Each automaton of SPOT is an object from a class that inherits from `tgba` class. It is an abstract class, which essentially provides a `get_init_state` and `get_succ_iter` methods. They allow to visit each TGBA, and so each automaton in SPOT. Moreover, some classical automata can be simulated with the proxy pattern (Gamma et al., 1995). For example we can see a class state based automaton (`sba_proxy`), or a class transition-based automaton (`tba_proxy`). These two examples are representative of the particularities of SPOT: SPOT works with TGBA, and does an intensive usage of on-the-fly algorithms.

All the algorithms that work on classical Büchi automata can work on TGBA, even if sometimes we need to transform our TGBA on-the-fly (into a Transition-based Büchi Automaton or a BA).

1.4.3 From LTL to BA

In this section we give an idea of the toolchain to transform an LTL formula into a TGBA or a BA.

Note 3 *Classical automata-theoretic approach uses BA for model-checking. Since* SPOT *uses TGBA, a process to transform a TGBA into a BA is mandatory, this is called the* degeneralization. *That's why we are interested in the sizes of the BA and of the TGBA.*

Figure 1.5 gives an overview of the SPOT's LTL-to-Büchi toolchain. Each box represents an algorithm; the only mandatory box is the translation. It takes as input an LTL formula, which can be rewritten by the "LTL Simplification" box to be translated in a more efficient way. It is then given to the "Translation" box. In SPOT there are several translation algorithms. One of these has a `-x` option which allows to try to determinize the automaton (Duret-Lutz, 2011).

Then we can take this as the result of the translation, or use the `-R3` option that reduces the number of acceptance conditions and deletes the dead Strongly Connected Components.

It is possible to run WDBA (Weak Deterministic Büchi Automata) (Abecassis, 2010). This algorithms works on TGBA representing obligation formulae. If the TGBA represents an obligation formula, it is translated into a minimal BA. Otherwise, the TGBA is returned unchanged.

We have added our algorithms after WDBA, and we run them only if WDBA fails. Because the BA generated with WDBA is minimal, running our algorithms after is just a loss of time. After our simulation-based reductions, the user can take it as a TGBA, or pass it through the degeneralization algorithm (Parutto, 2011) if a BA is requested.

Chapter 2

Simulation-based Automata Reduction

The goal of this report is to present methods that reduce automata, and in this case, after the translation of the formula. We want to reduce automata because it saves time and memory: the automaton that represents the formula is not the automaton on which the model checker is working, it works on the product of the model and the formula. Because the state space can be huge, the product can be even bigger (at worst Cartesian). This is called combinatorial explosion. Parutto (2012) presents a method to reduce the size of the structure that represents the model, and our job is to work on the automaton that is the formula translated into a TGBA. If we reduce its size in term of states, we reduce the memory we use, and if we reduce its size in term of transitions, we reduce the time we spend on the emptiness check. The methods we present in this report are called simulation-based reduction. The main idea of these algorithms is to delete the redundant states and transitions. My previous reports (Badie, 2011, 2012) present bisimulation that merges two states if they express exactly the same language. Simulation is a better algorithm: it merges two states if the language accepted from a state is included in the language of the other state. In this report, we present these two algorithms because they are the bases of the ones that are presented later, and then we present several algorithms that are derived from the simulation.

2.1 Bisimulation

The way to check if some states are redundant is to use a signature (Wimmer et al., 2006). We compute a signature for each state and this signature represents what we can see from this state. The signature includes the label of the transitions, the class of the outgoing state and the acceptance conditions. All the states that have the same signature are put into the same class.

We use a BDD to represent the signature. According to Note 2, each acceptance condition is encoded with a Boolean variable. So a set of three acceptance conditions is just a conjunction of these three variables.

If two states have the same signature, we can merge them, because they have the same suffix. This is the bisimulation. Figure 2.1 shows an example of bisimulation. If we assume that the original class of each state is $\mathcal{C}$, the signature of the state s_1 and s_2 are:

$$sig(s_1) = b \wedge a \wedge \bullet \wedge \mathcal{C}$$

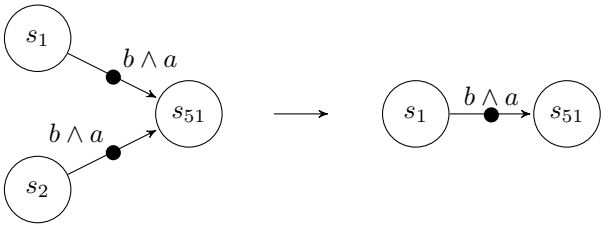


Figure 2.1: An example of bisimulation: states s_1 and s_2 simulate each other.

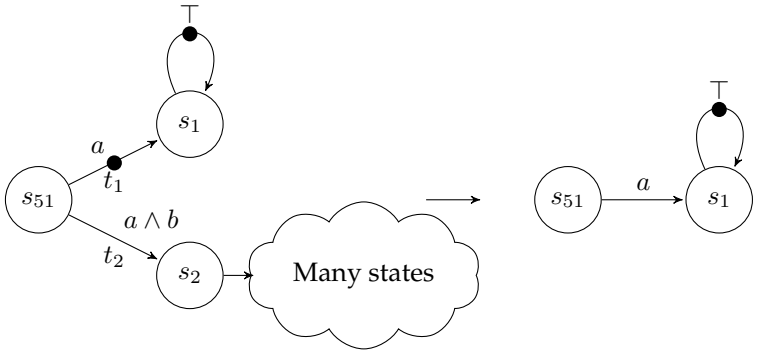


Figure 2.2: An example of a Simulation.

$$sig(s_2) = b \wedge a \wedge \bullet \wedge \mathcal{C}$$

They are the same, so the bisimulation can merge them into a new state s_1 that keeps the same outgoing transitions.

2.2 Simulation

Simulation is more efficient than bisimulation because it can detect if the suffix recognized by a state is a subset of the suffix recognized by another one.

Let's assume we have a TGBA with two transitions $t_1 = (L_1, A_1, \mathcal{C}_1)$ and $t_2 = (L_2, A_2, \mathcal{C}_2)$ where L_i is a label, A_i is the set of acceptance conditions, and $\mathcal{C}_i$ the class of the destination state. We say that a transition t_1 is dominated by a transition t_2 if and only if:

$$L_1 \rightarrow L_2 \tag{2.1a}$$
$$A_1 \leftarrow A_2 \tag{2.1b}$$
$$\mathcal{C}_1 \rightarrow \mathcal{C}_2 \tag{2.1c}$$

Equations (2.1a) and (2.1b) represent that the label of t_1 dominates those of t_2, and Equation (2.1c) represents that s_1 simulates s_2.

Let's take the example of Figure 2.2 to understand these formulae. The cloud of states represents the rest of the automaton, and whatever it is, it is not needed to understand the example.

Let's take a look at the words recognized by the automaton of Figure 2.2. If we take the transition t_1, we have to read a, and after that, we are on the state s_1 that accepts everything. To explore the transition t_2 we have to read $a \wedge b$. And after that, we don't know what we have

to read but it does not matter because if we have read $a \wedge b$, we have read a, which means that we are also in s_1. And because s_1 accepts everything, the transition t_2 is useless. This is the intuition. But now, let's explain how to detect this redundancy algorithmically.

Equation (2.1a) focuses on the label part of the transitions. The $\rightarrow$ symbol is at the Boolean level, it is the "implies" relation of the propositional logic. Let's take the two transitions of the state s_{51} as an example. t_1 is the name of the transition to the state s_2 and t_2 the name of the one to s_1. In this case, l_1 is $a \wedge b$ and l_2 is a. And because $a \wedge b \rightarrow a$, Equation (2.1a) is verified.

Equation (2.1b) focuses on the acceptance condition part of the transition. If we consider the set of acceptance conditions as a conjunction of Boolean variables and "no acceptance condition" as "true", we can consider the $\rightarrow$ as above. So, considering we are keeping the same t_1 and t_2, "$a_1 \leftarrow a_2$" because "$\top \leftarrow \bullet$".

The third equation works on the classes. As in the bisimulation, the simulation algorithm creates classes and then refines them. A class represents all the states that recognize the same suffix. But the suffix recognized by a class could be included in the one recognized by another one. In this case, the more restrictive class is said to *imply* the other. The class of the destination state of the transition t_2 verifies everything (because we accept all with an acceptance condition). So this class implies the class of the state s_1 whatever it verifies (even if it also verifies everything). So "$\mathcal{C}_2 \rightarrow \mathcal{C}_1$".

As a result, $t_1 \rightarrow t_2$. Thus, we can remove transition t_1 thanks to the simulation, and it leads to the simplified automaton of Figure 2.2.

In case of the bisimulation, we used a signature to represent the suffix of a state. We want to use a signature in the simulation too, because we can use a BDD (presented in Section 1.2) to represent it. It allows to check the equality between two BDD in $\mathcal{O}(1)$, and it is easy to know if a BDD implies another one because we let BuDDy (the BDD library) do it.

The problem is that the rules to know if a transition implies another (seen in Equation (2.1)) are not always in the same direction for the three equations. Equation (2.1b) is in the other direction. Otherwise we could have put the signature in a BDD and let it make the simplifications. But in this form, $\mathsf{sig}(t_1) \rightarrow \mathsf{sig}(t_2)$ does not imply that t_2 $\mathsf{dominates}$ t_1.

In fact, if we change the representation of our automaton from a TGBA to a Promise-Automaton (presented in Section 1.3.3), Equation (2.1b) is reversed. This transformation can be done on the fly, when working on the automaton. So we can define the signature of a transition like this:

Definition 3 *Let t be a transition such as t = (l, acc, $\mathcal{C}$) where "l" is the label of the transition, "acc" is the set of acceptance conditions and "$\mathcal{C}$" is the class of the outgoing state. Let R_{acc} be a relation that associates a set of acceptance conditions to a conjunction of all the acceptance conditions known by the automaton and that are not currently on the transition i.e. the promises. Let R_{class} be a relation which associates a class to a conjunction of all the class that implies this one. The signature of t is defined by:*

$$sig(t) = l \wedge R_{acc}(acc) \wedge R_{class}(\mathcal{C})$$

Definition 4 *Let s be a state that has a set of transitions $s.tr$. The signature of s is:*

$$sig(s) = \bigvee_{t \in s.tr} sig(t)$$

With sig(t) corresponding to Definition 3.

The simulation algorithm, as shown by Etessami and Holzmann (2000), is defining the canvas of the method on which we have built our work. The difference with the bisimulation is that we keep only non-redundant transitions in the signature. Non-redundant means there is no

transition in the result that is dominated by another one. To be able to do this we have to keep the classes partially ordered with respect to their signature inclusion.

A partial order is an order where only a subset of all the elements is in relation with the others. A partial order is a binary relation "$\leq$" over a set $\mathcal{P}$ that is reflexive ($\forall a, a \leq a$), antisymmetric ($\forall a, b; a \leq b$ and $b \leq a$ implies $a = b$) and transitive ($\forall a, b, c; a \leq b$ and $b \leq c$ implies $a \leq c$).

In our use case, $\mathcal{P}$ corresponds to the set of classes, and "$\leq$" is the relation: $\mathsf{class}(s_1) \leq \mathsf{class}(s_2)$ if and only if $\mathsf{sig}(s_1) \rightarrow \mathsf{sig}(s_2)$. And this implies "the suffixes seen by the elements of $\mathcal{C}_2$ is included by the one seen by the elements of $\mathcal{C}_1$". To compute this, we use the signature: behind each class there is the signature of all the states that are in this class. If the signature of class $\mathcal{C}_2$ is implied by the signature of class $\mathcal{C}_1$, then "$\mathcal{C}_2 \leq \mathcal{C}_1$".

For example, in Figure 2.2, if all the states are in class $\mathcal{C}$, the signature of the state s_{51} according to Definition 4 is:

$$sig(s_{51}) = (a \wedge \mathcal{C}) \vee (a \wedge b \wedge \bullet \wedge \mathcal{C}) = a \wedge \mathcal{C}$$

Assume that state s_1 is in the class $\mathcal{C}_1$, s_2 is in the class $\mathcal{C}_2$ and that $\mathcal{C}_1 \rightarrow \mathcal{C}_2$. The signature of the state s_{51} is:

$$sig(s_{51}) = (a \wedge \mathcal{C}_1) \vee (a \wedge b \wedge \bullet \wedge \mathcal{C}_2 \wedge \mathcal{C}_1) = a \wedge \mathcal{C}_1$$

If we suppose there is no class creation in the unknown part of the automaton, we can stop our algorithm because there is no class creation, and because the partial order is unchanged.

So when building the transitions of the initial state, we only see one transition, and we remove all the parts after s_2.

2.2.1 The Algorithm

Now that we have explained the intuition on the signature and how it works on a small example, we can specify the algorithm with a pseudo code.

In line 1, we initialize all the variables:

$\mathcal{C}l$ At the beginning, is the only class. It contains all the states.

R_{acc} Relation that takes an acceptance set, and associates its dual promise.

$\mathcal{C}^i$ A relation that takes a state and associates its class.

R_{class} A relation that takes a class and associates all the classes implied by this one. At the beginning, there is only one class, so this relation only associates $\mathcal{C}l$ to itself.

We use two variables $prev\_po\_size$ and po_size to know the number of implications between the classes, and i to represent the number of iterations.

This algorithm works by partition refinement. The idea is to put states into classes and to compute implications between classes. Loop line 9 computes the signature of each state. The signature is explained by Definition 4. All the states with the same signature are put in the same class.

Then relation R_{class} must be updated, line 12 starts by cleaning all the implications of the previous iteration. Loop line 13 checks the implication between all the classes. As an optimization, we avoid "does c_i implies c_i?" because the implication relation is reflexive.

Line 16 is a test on the signature of the class. This test allows us to know if a class c_1 implies a class c_2. In this case, we add c_2 to the list of classes implied by c_1, and we increment the number of implications between classes.

Algorithm 1 Direct Simulation Algorithm.

1: /* Initialization:
   Build a relation R_{acc} that associates an acceptance condition to its dual promise.
   $\forall q \in \mathcal{Q}$, $\mathcal{C}^{-1}(q) := \mathcal{C}l$ where $\mathcal{C}l$ is the initial class.
   Build a relation R_{class} that associates $\mathcal{C}l$ to $\mathcal{C}l$ */
2: $i := 0$
3: $prev\_po\_size$:= -1
4: $po\_size := 0$
5: **while** $\mid \mathcal{C}^i(\mathcal{Q}) \mid \neq \mid \mathcal{C}^{i-1}(\mathcal{Q}) \mid$ or $prev\_po\_size \neq po\_size$ **do**
6: $i := i + 1$
7: $po\_size := 0$
8: $prev\_po\_size := po\_size$
9: **for all** $q \in \mathcal{Q}$ **do**
10: $\mathcal{C}^i(q) := \bigvee_{t \in q.tr}(t.label \wedge R_{acc}[t.acc\_cond] \wedge R_{class}[\mathcal{C}^i(t.state)])$
11: **end for**
12: R_{class}.clean()
/* Update the implication relation between classes. */
13: **for all** $c_1 \in \mathcal{C}^i(Q)$ **do**
14: Set $accu := c_1$
15: **for all** $c_2 \in \mathcal{C}^i(Q) \setminus c_1$ **do**
16: **if** $\mathsf{sig}(c_1) \rightarrow \mathsf{sig}(c_2)$ **then**
17: $accu := c_2 \wedge accu$
18: $++po\_size$
19: **end if**
20: **end for**
21: $R_{class}(c_1) := accu$
22: **end for**
/* Rename the classes for the next iteration. */
23: **end while**
24: $\mathcal{C} := \mathcal{C}^i$
25: **return** $A' := \langle \Sigma' := \Sigma, Q' := \mathcal{C}(\mathcal{Q}), \delta', q'_0 := \mathcal{C}(q_0), \mathcal{F}' := \mathcal{F} \rangle$
/* Where $t \in \delta'$ defined if t appears in the signature of the state $\mathsf{SRC}(t)$. */

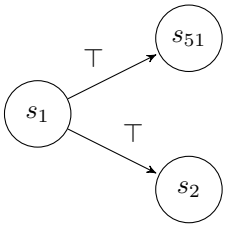


Figure 2.3: The use of Irredundant Sum-of-Products Form.

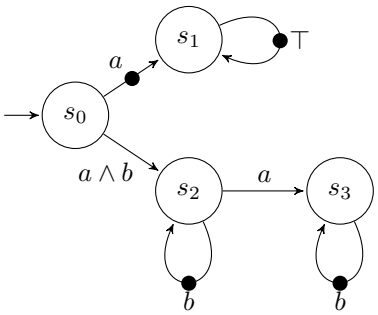


Figure 2.4: An example of direct simulation.

After this loop, we have to rename the classes, because we have assigned new names to the classes created in this iteration, and we want that the names of the classes in the signature correspond to the names of the classes.

The main loop of the direct simulation algorithm (line 5) stops when no new class has been created, and when there is no new implication between the classes. After that we have to build the result.

We create a state by class, and we compute the transition between them by reading the signature. For this, we have to change the signature into an irredundant sum of product as explained in Section 1.2. In this form, each product represents a transition. We just have to split each product to extract the label, the promises, and the destination class; it is easy because the set of variables that represents labels, promises and classes is known. If we don't use this form, we can get unreadable signature. Figure 2.3 shows the example where the signature we expects is $\mathcal{C}_1 \vee \mathcal{C}_2$. But this can be written $\mathcal{C}_1 \vee (\neg\mathcal{C}_1 \wedge \mathcal{C}_2)$ as explained in Section 1.2. And when we tried to recreate the automaton that corresponds to this signature, we see a class $\neg\mathcal{C}_1 \wedge \mathcal{C}_2$, which does not make sense.

2.2.2 Example

To show the algorithm in action, here is a complete example of how the simulation works.

Figure 2.4 shows the TGBA on which we will work. We start by putting all the states in the class $\mathcal{A}$, R_{acc} consists in a simple "not" on the acceptance condition. The signature, the classes and the implication between them are shown in Table 2.1.

Before going to the next iteration, we have to explain why the signature of s_0 is just $a \wedge \mathcal{A}$. The BDD eliminates all redundancy in the signature, and at first it was: $(a \wedge b \wedge \bullet \wedge \mathcal{A}) \vee (a \wedge \mathcal{A})$. But if we have the first product of this expression, we have a and we have $\mathcal{A}$, so we have the

St.	Signature	Class	R_{class}(Class)
s_0	$a \wedge \mathcal{A}$	$\mathcal{B}$	$\mathcal{B} \wedge \mathcal{C}$
s_1	$\mathcal{A}$	$\mathcal{C}$	$\mathcal{C}$
s_2	$(b \wedge \mathcal{A}) \vee (a \wedge \bullet \wedge \mathcal{A})$	$\mathcal{D}$	$\mathcal{D} \wedge \mathcal{C}$
s_3	$b \wedge \mathcal{A}$	$\mathcal{E}$	$\mathcal{E} \wedge \mathcal{C}$

Table 2.1: The signatures, classes and R_{class} at the first iteration.

St.	Signature	Class	R_{class}(Class)
s_0	$a \wedge \mathcal{C}$	$\mathcal{F}$	$\mathcal{F} \wedge \mathcal{G}$
s_1	$\mathcal{C}$	$\mathcal{G}$	$\mathcal{G}$
s_2	$(b \wedge \mathcal{D} \wedge \mathcal{C}) \vee (a \wedge \bullet \wedge \mathcal{E} \wedge \mathcal{C})$	$\mathcal{H}$	$\mathcal{D} \wedge \mathcal{G}$
s_3	$b \wedge \mathcal{E} \wedge \mathcal{C}$	$\mathcal{I}$	$\mathcal{I} \wedge \mathcal{G}$

second product. The first part is fully redundant and is eliminated.

All the signature are different, this leads to have one class per state.

$\mathcal{B}$ implies $\mathcal{C}$ because its signature implies $\mathcal{C}$'s. It is the same for $\mathcal{D}$ and $\mathcal{E}$. Now, we can go to the next iteration.

We explain again why the signature of s_0 is $a \wedge \mathcal{C}$. This time the signature without simplification would be: $(a \wedge \mathcal{C}) \vee (a \wedge b \wedge \bullet \wedge \mathcal{D} \wedge \mathcal{C})$. We see that we are able to make this simplification thanks to R_{class} which adds $\mathcal{C}$ to the signature.

Neither the number of classes nor the number of implications have changed. So we can stop the algorithm and build the result.

For building the result, we have to rename all the classes to be able to associate the class names in the signature and the current ones. So we just rename $\mathcal{F}$ into $\mathcal{B}$, etc.

We want to have each signature in an Irredundant Sum Of Product because each product represents a transition, and we use an algorithm from Minato (1992) as explained in Section 2.2.3.

We start by creating the initial state and its transitions by reading the signature of class $\mathcal{B}$ (which contains the initial state). The signature is composed of only one product, so there is only one transition. The label part of this condition is a, the acceptance condition part is empty, and the class part is $\mathcal{C}$. So we create a transition a with an acceptance condition $\bullet$ (because $R_{class}(\top) = \bullet$) and to the state representing the class $\mathcal{C}$.

We create the transition of the state representing $\mathcal{C}$. The label part of the signature is $\top$, the acceptance condition part is empty, and the class part is $\mathcal{C}$. We create a loop labelled by $\top$ and with an acceptance condition $\bullet$. We have seen no new state, so the algorithm ends by returning the automaton shown in Figure 2.5.

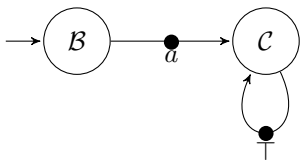


Figure 2.5: The result of the simulation example.

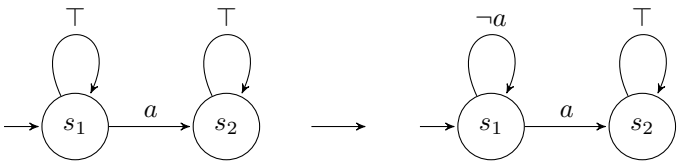


Figure 2.6: An example of a determinization on the TGBA which represents F a.

2.2.3 Implementation

In this section, we present some implementation details and choices we made.

Instead of making a relation "R_{acc}" as said in the algorithm, we chose to develop an algorithm to work on the automaton, and to replace each set of acceptance conditions by its complement.

We detail the method to test the correctness of our algorithm in Section 2.6.1.

Determinization

We have added a loop around the call to Minato's algorithm which runs through all the possible valuations of each atomic proposition. This loop helps to determinize the automaton. As an example, it makes the transformation represented in Figure 2.6.

To have a better understanding of the interest of this determinization, let's assume we have run the whole simulation algorithm. At the end the signature of s_1 is $(a \wedge \bullet \wedge \mathcal{B}) \vee (\bullet \wedge \mathcal{A})$ where $\mathcal{B}$ is the class of s_2 and $\mathcal{A}$ the class of s_1. So we verify how the different valuations of a influence the signature:

$$\neg a \ \wedge \mathsf{sig}(s_0) = \neg a \wedge \bullet \wedge \mathcal{A}$$

$$a \ \wedge \mathsf{sig}(s_0) = a \wedge \bullet \wedge \mathcal{B}$$

We create a transition for each product, and we have a loop labelled by $\neg a$, and a transition labelled by a to the state s_2 (the acceptance condition we can see on the signature are here because of R_{acc}).

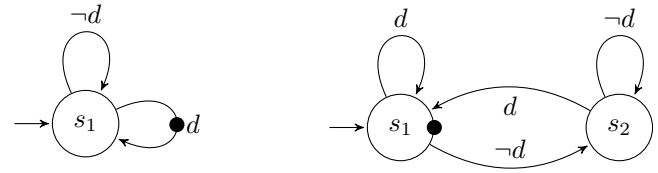
Example

In order to give an example of the benefits of the simulation to reduce automata, we have applied it on the two automata presented in the introduction. One is a BA and the other a TGBA. Figure 2.7 presents the results.

2.2.4 Limits

There are some cases where the simulation is useless, and we are going to present three of them.

Figure 2.8 shows the first case where we would like to see a simplification and where the simulation cannot do it. In this case we want to reduce it by removing the first state which is useless. The intuition is that we are able to transform our transition from $\top$ to a transition a and another one $\neg a$. This transformation is correct because these transitions are not in a Strongly Connected Component (see Note 1). So we want to add an acceptance condition to the a transition.



(a) Simplified TGBA. (b) Simplified BA.

Figure 2.7: The simplified examples on "G a U G F d".

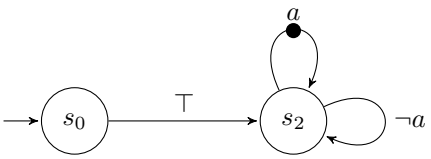


Figure 2.8: An example where the simulation is useless: X G F a without any reduction.

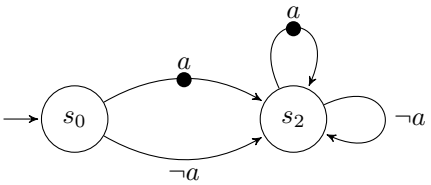


Figure 2.9: Splitting the transition between s_0 and s_2 doesn't change the language.

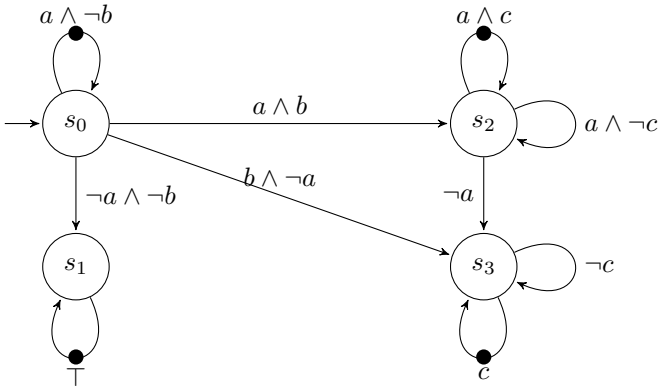


Figure 2.10: Another formula where the simulation is useless.

Figure 2.9 shows the automaton after a small rewriting of the transition. Now we are able to see that this can be reduced with a bisimulation. But it is not easy to detect automatically.

Another case, similar, but a little more complex is shown by Figure 2.10. We are interested in merging s_2 and s_3. We can apply the same transformation as before on the transition (s_2, s_3) and by splitting $\neg a$ into $\neg a \wedge \neg c$ and $\neg a \wedge c \wedge \bullet$.

So the signature of these two states becomes (if they all are in the same class $\mathcal{C}$ initially):

$$sig(s_2) = (a \wedge c \wedge \mathcal{C}) \vee (a \wedge \neg c \wedge \bullet \wedge \mathcal{C}) \vee (\neg a \wedge \neg c \wedge \bullet \wedge \mathcal{C}) \vee (\neg a \wedge c \wedge \mathcal{C}) = (\neg c \wedge \bullet \wedge \mathcal{C}) \vee (c \wedge \mathcal{C}) = sig(s_3)$$

The two previous cases are not reduced by the simulation because the algorithm takes the automaton as it comes, and does not try to be smart about what is on the transitions that are out of the SCCs. On these transitions, we can add or remove an acceptance condition, it won't change the set of words accepted by the automaton. We developed an algorithm on this idea, it is explained in Section 2.5.

The third case is different from the previous ones. It does not depend on the transitions that link two SCCs. The simulation is an algorithm that works on the suffixes of the states. Sometimes, simplifications can be done on the prefixes of the states. Let's see an example on Figure 2.11.

If we take a look at the states s_1 and s_2, we can see that we explore these states by reading a. There is no other transitions that go to this state. They have the same prefix signature. The simulation cannot do anything, but if we consider the prefixes instead of the suffixes, we can merge these states. We could have the automaton of Figure 2.12.

2.3 Reverse Simulation

The third case of Section 2.2.4 can be managed by the reverse simulation. Somenzi and Bloem (2000) present a version for the BA. The simulation is an evolution of the bisimulation that is inspired by the quotient operation on a deterministic finite automaton. The quotient merges states that recognize the same suffix. The coquotient merges states that recognize the same prefix.

The reverse simulation can be seen as a simple adaptation of the simulation. We could see it as the following operation:

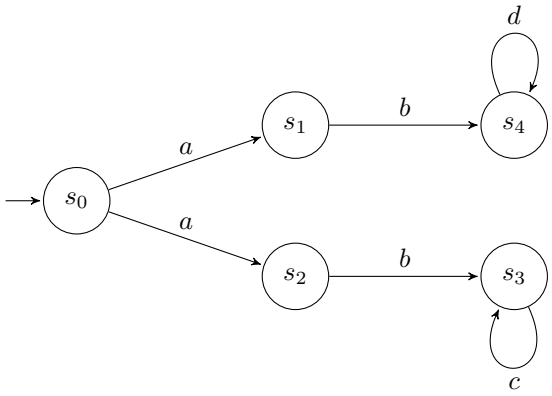


Figure 2.11: A case where the reverse simulation is useful.

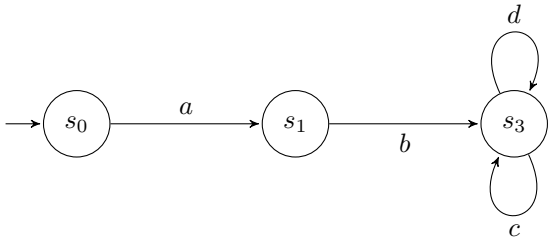


Figure 2.12: The result of the reverse simulation.

Algorithm 2 Direct Reverse Simulation Algorithm.

Input: automaton : tgba

```
1: transposed_tgba := transpose_tgba(automaton)
2: transposed_result := simulation(transposed_tgba)
3: result := transpose_tgba(transposed_result)
4: return result
```

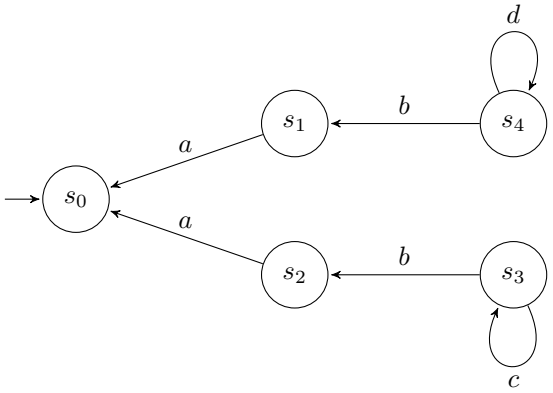


Figure 2.13: The TGBA of Figure 2.11 reversed.

The problem is that there is no `transpose_tgba` function in SPOT. In fact, this operation would be hard to create. Transposing a finite automaton is quite easy, each final state becomes an initial one, and each initial state becomes a final state. Each transition swaps source and destination. That's it. There are several characteristics that make the modification of this algorithm for TGBA a real problem:

- There is no final state in a TGBA;
- SPOT does not handle several initial states.

One does not simply manage several initial states in SPOT. So we have thought about another method. The simulation algorithm is an algorithm that works on each state separately. So we do not have to run through it, and our automaton can be invalid during the process of the simulation. We can simply reverse the source and the destination of each transition, without caring about the real meaning of this automaton.

Before that, we register each state, and since our implementation does not run through the automaton and just works on a list of states, we can do that. To have an example of what does a TGBA transposed by our method looks like, see Figure 2.13.

It is clear that we cannot do anything with this automaton by a depth-first run or any kind of run. But the lifetime of this automaton is very short, it is constructed at the beginning of the simulation and destroyed at the end.

We just have to be careful with the initial state because the reverse Simulation works on finite prefix. That's why we add a flag to its signature.

If we run the simulation on the automaton represented by Figure 2.13, it will merge states s_1 and s_2 because they have the same signature. And it will return the automaton of Figure 2.12 as expected.

We have succeed in reusing the simulation's implementation and it took less than 15 lines of code to make it work.

Experimentally, it appears that reductions only occurs when the automaton is non-deterministic. If two states share the same prefix, that means there is some non-determinism somewhere.

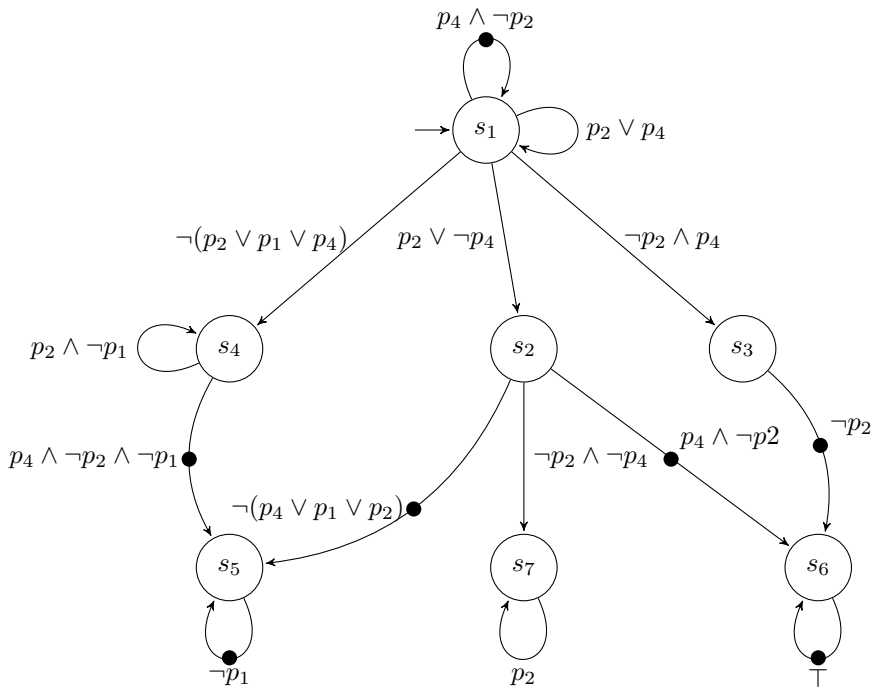


Figure 2.14: Iterated simulations on $\neg(\mathsf{X}\, p_2\, \mathsf{U}\, \mathsf{G}(\neg(p_4 \vee (\mathsf{F}\, p_1 \Leftrightarrow p_4))\, \mathsf{W}\, p_2))$ without SCC filter.

2.4 Iterated Simulation

The Iterated Simulation is an iteration on the direct simulation and the direct reverse simulation. We loop until we have a fixpoint.

The idea is that after we have done the direct simulation, it may still be some redundancies on the prefix of the states. Same thing when we have ran the direct reverse simulation, etc.

We have included scc_filter in this loop, because sometimes, the reverse simulation let things in a bad situation that is not reducible by the simulation. A perfect example is represented by Figures 2.14 and 2.15. Without the SCC filter, we have a state s_7 that only contains a non accepting self loop. It is useless for the automaton, but the simulation is unable to reduce it. Adding an SCC filter removes this state from this automaton.

On average, the Iterated Simulation does two loops before reaching a fixpoint, that means that only one iteration is enough to obtain the final result.

2.5 Don't Care Simulation

Several limits of the simulation are shown in Section 2.2.4. The main cases that are not reduced by the simulation are due to the existence (or not) of an acceptance condition on a transition that is out of an SCC.

Figure 2.16 shows that a simple missing acceptance condition on the transition t led the simulation to not merge states s_0 and s_1. But if we add an acceptance condition on t, the merge occurs.

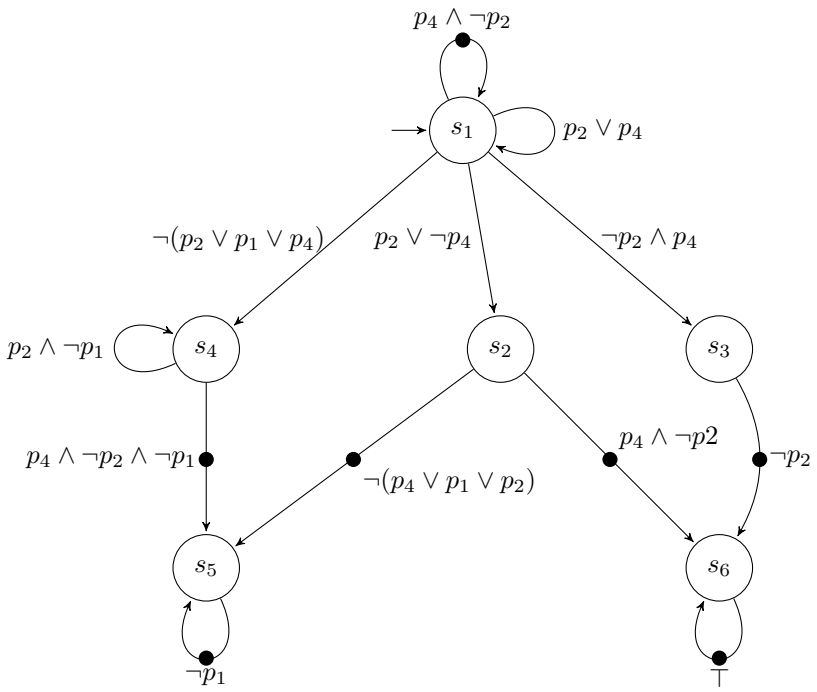


Figure 2.15: Iterated simulations on $\neg(\mathsf{X}\, p_2 \,\mathsf{U}\, \mathsf{G}(\neg(p_4 \vee (\mathsf{F}\, p_1 \Leftrightarrow p_4))\, \mathsf{W}\, p_2))$ with SCC filter.

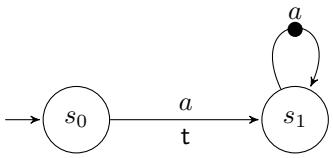


Figure 2.16: An acceptance condition is enough to avoid simplification.

There are two options for `tgbatest/ltl2tgba` that play with the transitions that are out of an SCC: `-R3f` and `-R3`. The first one removes each acceptance condition of each transition out of an SCC, while the other let them as-is. So the results of the simulation may differ in function of which option we use.

The aim of the don't care simulation is to avoid being dependant of the acceptance conditions on the transitions that are out of an SCC. We try to get the best configuration possible for each transition and so obtain the best reduction possible by the simulation.

2.5.1 The Idea

At first, we thought that we could develop a kind of brute force that tests every combination of acceptance conditions on each transition that is out of an SCC. And then, we run the simulation on each newly created automaton, and we keep only the smallest. But it may require many combinations. Indeed, let be T the number of transitions out of an SCC, and A the number of acceptance conditions, the number of simulation needed is $2^{A \times T}$. In function of the automaton in input, it could be very long.

Moreover, this idea does not solve all the problems we want to solve. For example, the automaton of Figure 2.8 would not be reduced by this brute force.

Alexandre Duret-Lutz has proposed a new signature, slightly different from the one presented in Definition 4. This new signature, called don't care signature, integrates the fact that a transition is (or not) a part of an SCC.

Definition 5 *Let s be a state, which has a set of transitions $s.tr$. The don't care signature of s is:*

$$sig(s) = \Big(\bigvee_{\substack{t \in s.tr \\ t \notin SCC}} noprom(\mathsf{sig}(t)) \wedge \bar{\circlearrowleft} \Big) \vee \Big(\bigvee_{\substack{t \in s.tr \\ t \in SCC}} \mathsf{sig}(t) \wedge \circlearrowleft \Big)$$

With $\mathsf{sig}(t)$ corresponding to Definition 3, $\bar{\circlearrowleft}$ representing "out of an SCC", $\circlearrowleft$ representing "in an SCC" and noprom is a function that removes all acceptance conditions from a signature.

With this "don't care signature" comes up a new "don't care implication".

Definition 6 *Let be s_1 and s_2 two states, $\mathsf{sig}(s_1)$ and $\mathsf{sig}(s_2)$ their don't care signatures, f_1 (resp. f_2) the part of the signature of s_1 (resp. s_2) that is in an SCC, and g_1 (resp. g_2) the other part.*

We say that s_1 could implies s_2 if and only if:

$$\begin{cases} f_1 \Rightarrow f_2 \vee g_2 \\ g_1 \Rightarrow noprom(f_2) \vee g_2 \end{cases}$$

Note 4 *A helpful reminder to compute the don't care implication by hand: if a state does not have an f or a g, the corresponding part is automatically verified since $\bot$ implies everything.*

With these new tools we are able to detect what are all the possible implications (we call this "could imply"). Our idea is to compare all possible implications with the ones that are already verified with the current configuration. This was our first step, then we have to detect how to make the possible implications effective. This is what we present in Section 2.5.2.

2.5.2 The Method

Because we are able to build the "imply" table (for the simulation), and the "could imply" table (for the don't care simulation), we are able to compute their difference. The difference creates a list of possible implications. A possible implication means that with a particular combination of acceptance conditions or atomic propositions on some of the transition that are out of an SCC, there is an implication. With this, we can work on both signatures of the state representing the class involved in this implication, and find what would be a better choice for these transitions.

It is important to have the assumption that "one state $\Leftrightarrow$ one class" in the automaton we work on. So, we need to run the simulation before starting the don't care simulation.

First, we have to detect which transitions can influence the result of the implication. For this, we know that the only transitions that can change are the ones that are out of an SCC.

The signature is composed of several parts, where each one represents a transition to a particular destination (see Definitions 3 and 4). The fact that a signature sig_1 implies another one sig_2 is equivalent to: "each transition of the sig_2 is implied by one in the sig_1". We can extract a constraint from this relation: a transition can be implied only by a transition that has either the same destination or a destination that implies the first one. Currently, in our implementation, we only test if the two transition the equal destination, it is something to explore to obtain better results.

We take each transition that is out of an SCC, and the corresponding one in the other signature. And then, we try to find what we should add to the transition to make the implication possible. We can add any acceptance condition, and we can split the transition by adding some atomic properties only if the atomic property part we add is equal to $\top$. For example, adding $a \vee \neg a$ is fine, adding a is not.

There are three cases when we pick these two transitions:

1. The transition out of an SCC is in the signature that could imply;
2. The transition out of an SCC is in the signature that could be implied;
3. The two transitions are out of an SCC.

We are not sure about what we should do in the latter case. We were unable to find an interesting case from where we could deduce a rule. Mostly, we have encountered cases where the two transitions were equal. So this is experimental.

The first case is handled as follow (the first transition is out of a SCC). Let's assume the first transition is $\alpha \wedge A_1 \wedge \mathcal{C}_1$ and the second is $\beta \wedge A_2 \wedge \mathcal{C}_2$ where A_i is the acceptance condition part, and $\mathcal{C}_i$ is the class part. We can remove $\mathcal{C}_1$ and $\mathcal{C}_2$ because they have no influence on what we should add on the first transition. In fact, we want to find an x such that the following statement holds:

$$\underbrace{x \wedge \alpha \wedge A_1}_{t_1} \Rightarrow \underbrace{\beta \wedge A_2}_{t_2}$$

To this end, we need to discover what is missing in $\alpha \wedge A_1$ to imply $\beta \wedge A_2$. Because t_1 is out of an SCC we can remove A_1 from this equation. The simplest way to respect this equation is to replace the "imply" by an equal. So we rewrite the equation like that:

$$x \wedge \alpha = \beta \wedge A_2 \Rightarrow x = \exists \alpha.(\alpha \wedge \beta \wedge A_2)$$

Now we have x, we just have to be sure that it respects the constraints we want.

St.	Sig	Cl.	R_{class}(Cl)	DC Sig	DC R_{class}(Cl)
s_0	$\bullet \wedge \mathcal{C}_0 \wedge \mathcal{C}_2$	$\mathcal{C}_0$	$\mathcal{C}_0$	$\circlearrowleft \wedge \mathcal{C}_0 \wedge \mathcal{C}_2$	$\mathcal{C}_0 \wedge \mathcal{C}_2$
s_2	$((\neg a \wedge \bullet) \vee a) \wedge \mathcal{C}_2 \wedge \mathcal{C}_0$	$\mathcal{C}_2$	$\mathcal{C}_0 \wedge \mathcal{C}_2$	$\circlearrowleft \wedge((\neg a \wedge \bullet) \vee a) \wedge \mathcal{C}_2 \wedge \mathcal{C}_0$	$\mathcal{C}_0 \wedge \mathcal{C}_2$

Table 2.2: Don't care Simulation at the first iteration.

Since we have replaced the implication by an equal, we can use the same process for the case 2. x is just on the other side of the equation:

$$x = \exists \beta.(\alpha \wedge \beta \wedge A_1)$$

And for the step 3. we just use the two formulae above.

Each x is called a constraint. For each "could imply", we associate a list of constraints. These constraints are local, and we don't exactly know how they can interact. So, we run a kind of brute force to compute every possibility, and for each one we run a simulation. We keep and return the best result. This is the source of a problem that Section 2.5.3 discusses.

Example

Let's see the don't care simulation in action on the example of Figure 2.8. Table 2.2 shows the state of the don't care (DC) simulation.

The difference between the "imply" (R_{class}) and "could imply" (DC R_{class}) is that $\mathcal{C}_0$ could imply $\mathcal{C}_2$.

There is only one transition out of an SCC. We take the one that has the same destination in the other signature. Because there is two transitions that go to the state s_2, there is a $\vee$ between them. So we want to solve:

$$x \wedge \top \Rightarrow (\neg a \wedge \bullet) \vee a$$

So x is simply equal to $(\neg a \wedge \bullet) \vee a$. We check that it does not violate any constraint on what x could be: $noprom(x) = \top$. It is okay, so we add this constraint to the simulation.

We have only one constraint, so we have to run every combination with one element, there are two. Without this constraint, we do not change anything, but with the constraint, we make the simulation runs on the automaton of Figure 2.9, and on this automaton, the simulation removes state s_0. This is how the don't care simulation works on a small example.

2.5.3 Combinatorial Problem

Because we don't know how the different constraints interact, we tried all the possible combinations. Let be T the number of transitions out of an SCC, and A the number of acceptance conditions, the number of simulation needed is $2^{A \times T}$.

It can be the source of some problems, because if $A \times T$ is big enough, we can become unable to store the number of simulations needed. The greatest exponent we encountered was 30. On that case, our algorithm took 5 hours to finish. To solve this problem, we have decided to find another method to compute all possibilities, but to not compute dummy things. For example, if we have the class C_1 that could imply the classes C_2 and C_3, and C_2 implies C_3, we don't run 4 simulations. We only run 3. Because testing C_2 without C_3 doesn't make sense.

This seems to be a small optimisation, but in fact, on the same example that took 5 hours, it now takes 15 seconds. Algorithm 3 shows exactly what is done. At the beginning, $S := \emptyset$. The function 'Imply' takes a class, and returns all the class that are implied by this one.

Algorithm 3 rec(C, S)

Input: Constraint_List: C
Input: Chosen_Constraints: S
  while C $\neq \emptyset$ **do**
    x := pick_one(C)
    rec(C $\setminus$ Imply(x), S $\vee$ Imply(x))
    C := C $\setminus${x}
  end while
  simulation(S)

It takes one class x randomly, and the ones that are implied by x. It can save a large amount of time. There is some heuristics to do in "pick_one" instead of selecting one randomly.

2.5.4 Implementation Details

There are some implementation problems that need to be explained. The simulation requires to work on a copy of the input automaton because we need to revert all the acceptance conditions. But in order to know the SCCs of the automaton, we use `tgba_dupexp_dfs` that does another copy of the automaton. So we get two extra copies per simulation. We were able to reduce to one by replacing the second copy by an in-place complement of the acceptance conditions.

We do not want to create a new automaton for each new combination, there are already enough copies. The problem is how to give the information to the simulation that there is something to change on the transition. Our method is to keep the same automaton in the don't care simulation, and to give an optional argument to the simulation, that is a hash table. The keys of this hash table are the address of the pointer of the source and destination states. The values are the x computed before.

The problem is that since there is some copy, we work with two hash tables. One that corresponds to the copy done by the `tgba_dupexp_dfs` algorithm (`new_original`) and another one that corresponds to the copy (if any) done by the simulation (`old_name`). It can lead to hard to read code, but it was the simplest way we found.

To factor as much code as possible, the don't care simulation inherits from the simulation, so it can run it, and have the implications to start the algorithm without duplicating any code.

2.6 Experimental Results

In this section, we present the experimental results for all the algorithms presented in this report. Before that, we present how to test the correctness of our algorithms. Then we present some benchmarks.

2.6.1 Tests

In order to test the correctness of our algorithms, we use LBTT (Tauriainen, 2008). It takes several translation algorithms, generates random formulae, and compares the results. To achieve this goal, it uses three algorithms (Tauriainen and Heljanko, 2002):

Emptiness Check This test takes as input an LTL formula ϕ. $\mathcal{L}_\phi$ is the language represented by ϕ and $\mathcal{L}_{\neg\phi}$ its negation. First, it computes A_ϕ the automaton representing ϕ and its negation: $A_{\neg\phi}$. Then it computes ψ the intersection: $A_\phi \otimes A_{\neg\phi}$. It finally checks the emptiness of ψ. If it is not empty, one of the two part does not recognize the language $\mathcal{L}_\phi$ or $\mathcal{L}_{\neg\phi}$.

Cross Comparison test Let S be a random state space, for each algorithm (A_ϕ (resp. $A_{\neg\phi}$) will be the result of the translation into a TGBA for the positive (resp. negative) formula. i for the number of the translation algorithm), emptinesscheck(A_ϕ * S) will give the same result $\forall i$. emptinesscheck($A_{\neg\phi}$ * S) will give the same result $\forall i$. In every other case, the translation is not correct.

Consistency check This test checks that all states of S are in an acceptant component of A_ϕ or $A_{\neg\phi}$ or the two. If the state is not in A_ϕ nor $A_{\neg\phi}$, the translation is incorrect.

Now we know how our implementation was tested, and we have a good argument to suppose the correctness of this algorithm, let's see its results.

One of the advantages of LBTT is that at the end of its execution, some statistics about the generated automata are shown. So we can easily get an idea of which is the best algorithm.

2.6.2 Benchmarks

In this section, we present some benchmarks.

The algorithm used for translating formulae into automata is still Couvreur FM. We also run three simplifications:

`-R3` Delete the dead SCCs, and reduce the number of acceptance conditions to its minimum;

`-r7` Make some formulae simplifications;

`-Rm` Run WDBA when possible.

We have run the following algorithm:

No Sim No more options than the ones described above.

RDS The Direct Simulation.

RRS The Reverse Simulation.

RIS The Iterated Simulation.

RDCS The Don't Care Simulation.

RDCIS The Don't Care Iterated Simulation. It is similar to the Iterated Simulation, except that instead of calling the simulation, we call the don't care simulation.

The timeout was set to 15 minutes. If a timeout occurs, a ☠ appears next to the name of the algorithm. The automaton generated for this corresponding formula is ignored for all the algorithms. That means that if there is one ☠, the number of automata will be 199 instead of 200 for all the algorithms, it allows us to compare the different algorithms.

The percentage are reduction relative to the "No Sim" line. So we can see how each algorithm contributes to have smaller automaton in SPOT.

Algorithms	Automata					Product	
SPOT	states	trans.	Non Det. St	Non Det.	time (s)	states	trans.
TGBA							
No Sim	852	1590	41	20	2.64	170356	5565670
RDS	848	1577	28	16	2.79	169542	5454349
RRS	848	1573	37	20	2.81	169556	5472219
RIS	846	1566	27	16	2.93	169142	5435193
RDCS	848	1575	28	16	2.97	169542	5454349
RDCIS	846	1564	27	16	3.42	169142	5435193
BA							
No Sim	861	1626	50	20	2.52	172156	5677398
RDS	857	1611	32	16	2.82	171342	5529081
RRS	857	1607	46	20	2.62	171356	5558364
RIS	856	1603	31	16	3.22	171142	5513759
RDCS	865	1625	34	16	2.95	172942	5562282
RDCIS	863	1614	33	16	2.77	172542	5543126

Table 2.3: Benchmark on small formulae.

Algorithms	Automata					Product	
SPOT	states	trans.	Non Det. St	Non Det.	time (s)	states	trans.
TGBA							
No Sim	1970	7032	581	77	7.22	385844	14358119
RDS	1881	6403	360	74	7.19	366944	11841814
RRS	1894	6336	483	76	7.32	370689	12582622
RIS	1860	6251	335	74	8.16	362802	11445298
RDCS (1 ×☠)	1880	6377	357	74	24.07	366723	11813236
RDCIS (1 ×☠)	1858	6216	328	74	63.22	362381	11400888
BA							
No Sim	2091	7688	690	77	7.2	409873	15661768
RDS	1972	6833	415	74	7.14	384814	12558421
RRS	1988	6783	565	76	7.35	389332	13453999
RIS	1971	6747	390	74	8.29	384141	12191100
RDCS (1 ×☠)	2091	7169	457	74	8.25	408177	13198186
RDCIS (1 ×☠)	2067	6991	426	74	11.38	403395	12690214

Table 2.4: Benchmark on big formulae.

Algorithms	Automata					Product	
SPOT	states	trans.	Non Det. St	Non Det.	time (s)	states	trans.
TGBA							
No Sim	702	1901	233	70	3.46	140167	6371627
RDS	676	1761	120	51	3.45	134677	4917433
RRS	668	1688	185	67	3.43	133369	5352279
RIS	659	1638	102	51	3.73	131507	4668891
RDCS	676	1750	120	51	5.18	134677	4917433
RDCIS	659	1614	93	51	5.52	131507	4644451
BA							
No Sim	768	2156	293	70	3.60	153364	7345375
RDS	742	2015	142	51	3.57	147727	5405417
RRS	732	1918	224	67	3.50	146159	6145500
RIS	721	1854	122	51	3.71	143810	5118850
RDCS	766	2080	151	51	5.19	152463	5551454
RDCIS	744	1899	121	51	5.58	148334	5230622

Table 2.5: Benchmark on known formulae.

Small Formulae Table 2.3 shows the results of the translation algorithms on 200 small formulae. Small means size[1] ten and four propositions.

Table 2.6 shows the percentage of reduction we have on these kinds of formulae.

Algorithms	Automata		Product	
SPOT	% states	% trans.	% states	% trans.
TGBA				
RDS	0.5	0.8	0.5	2.0
RRS	0.5	1.1	0.5	1.7
RIS	0.7	1.5	0.7	2.3
RDCS	0.5	0.9	0.5	2.0
RDCIS	0.7	1.6	0.7	2.3
BA				
RDS	0.5	0.9	0.5	2.6
RRS	0.5	1.2	0.5	2.1
RIS	0.6	1.4	0.6	2.9
RDCS	-0.5	0.1	-0.4	2.0
RDCIS	-0.2	0.7	-0.2	2.4

Table 2.6: Percentage of reduction on small formulae.

Big Formulae Table 2.4 the results of the translation algorithms on 199 big formulae. Big means size 12 to 15 and 8 propositions.

There was one timeout on a formula. This formula has 29 states and 126 transitions, which explains the timeout. To be able to compare equally all the algorithms, we have removed the size of the automata generated by this formula for all the algorithms.

[1]Size corresponds to the number of nodes allocated in the parse tree of the generated LTL formula.

Table 2.7 shows the percentage of reduction we have on these kinds of formulae.

Algorithms	Automata		Product	
SPOT	% states	% trans.	% states	% trans.
TGBA				
RDS	4.5	8.9	4.9	17.5
RRS	3.9	9.9	3.9	12.4
RIS	5.6	11.1	6.0	20.2
RDCS	4.6	9.3	4.9	17.7
RDCIS	5.7	11.6	6.0	20.6
BA				
RDS	5.7	11.1	6.1	19.8
RRS	4.9	11.8	5.0	14.1
RIS	5.7	12.2	6.3	22.2
RDCS	0.0	6.7	0.4	15.7
RDCIS	1.1	9.0	1.6	19.0

Table 2.7: Percentage of reduction on big formulae.

Known Formulae The known formulae shown in Table 2.5 are taken from Dwyer et al. (1998), Etessami and Holzmann (2000), and Somenzi and Bloem (2000).

Table 2.8 shows the percentage of reduction we have on these kinds of formulae.

Algorithms	Automata		Product	
SPOT	% states	% trans.	% states	% trans.
TGBA				
RDS	3.7	7.4	3.9	22.8
RRS	4.8	11.2	4.8	16.0
RIS	6.1	13.8	6.2	26.7
RDCS	3.7	7.9	3.9	22.8
RDCIS	6.1	15.1	6.2	27.1
BA				
RDS	3.4	6.5	3.7	26.4
RRS	4.7	11.0	4.7	16.3
RIS	6.1	14.0	6.2	30.3
RDCS	0.3	3.5	0.6	24.4
RDCIS	3.1	11.9	3.3	28.8

Table 2.8: Percentage of reduction on known formulae.

On the TGBA, the algorithms are sorted by performance in the following way: RRS < RDS < RDCS < RIS < RDCIS.

But RDCS and RDCIS are able to run for a very long time while RIS is able to give an answer almost instantaneously.

On the BA, the results are more surprising. On these, RDCS and RDCIS create sometimes bigger automata than their inputs, which is quite annoying. This is due to the fact that the

automaton comes with an original configuration on its transition out of an SCC. The don't care simulation changes that, and the degeneralization can give different results. Sometimes better, sometimes worse.

The formula $\mathsf{X}\,\mathsf{X}\,\mathsf{X}(p_2\ \mathsf{U}\ (\bot\ \mathsf{U}\ (p_1\ \mathsf{V}\ p_0)))$ is better reduced with just `-R3` than with the $RDCS$. It is because the don't care simulation tries to find the best configuration for the simulation, but does not know anything about the `degeneralize` function. A solution would be to apply these algorithms after the degeneralization. But that may require some modifications of the algorithms.

There exists degeneralization algorithm that is not influenced by the acceptance condition on transitions that are out of an SCC. It would be a good test, but for a lack of time, we didn't test the effect of this new algorithm.

Conclusion and Future Work

In this report, we have presented several methods to reduce Büchi Automata and Transition-based Generalized Büchi Automata that are called Simulation-based. Our Simulation is built on the Simulation presented by Etessami and Holzmann (2000) that was meant to work on BA. We had to generalize it to apply it on TGBA. We also generalized the Reverse Simulation (Somenzi and Bloem, 2000). We took advantage of the Direct and the Reverse Simulation to create an algorithm that iterates on them until a fixpoint is reached. This is called the Iterated Simulation.

We have also developed an experimental version of the Simulation that doesn't take into account the acceptance conditions on the transition that are out of an SCC. It is called the Don't Care Simulation (Somenzi and Bloem, 2000). We have also created a Don't Care Iterated Simulation.

Each algorithm contributes to a better reduction of the automaton outputted by SPOT. Section 2.6.2 shows how these algorithms makes TGBA and BA smaller. It is not perfect yet, for example, sometimes the Don't Care Simulation makes the BA bigger, because of the `degeneralize` algorithms.

The best results are obtained with the Don't Care Iterated Simulation on the TGBA. On the formulae from the literature, we obtain 6.1% of reduction on the states of the TGBA and 15% on the transitions. The simulation was able to reduce up to 3.7% on the states, and 7.3% on the transitions. We have nearly doubled the effect of the reduction of the simulation with these new algorithms.

This work leaves some questions opened. First, for now, we run the simulation after WDBA, and before the degeneralization. It seems that running it after the degeneralization can reduce the automaton a second time. It may need some rewriting on the algorithms to make them efficient on the BA.

The Don't Care Simulation is still experimental, and has several changes that need to be explored. For example, when we pick a transition that is out of an SCC, we look at the one that has exactly the same set of destination. This is not enough because we can also look at the destinations implied.

When we want to find x, we replace the implication by an equal. Maybe we miss something here, and it could be interesting to see how to extend this rule. We are also not sure of the rule to find x when there are two transitions that are out of an SCC. Finally, it would be interesting to develop a "Don't Care Reverse Simulation".

Index

Bibliography

Abecassis, F. (2010). Minimization of automata representing obligation formulae. Technical Report 1004, EPITA Research and Development Laboratory (LRDE).

Badie, T. (2011). Bisimulation-based reductions on TGBA. Technical Report 1104, EPITA Research and Development Laboratory (LRDE).

Badie, T. (2012). Direct-simulation reduction for TGBA. Technical Report 1203, EPITA Research and Development Laboratory (LRDE).

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691.

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)*, Lecture Notes in Computer Science, pages 253–271. Springer-Verlag.

Duret-Lutz, A. (2011). LTL translation improvements in Spot. In *Proceedings of 5th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'11)*.

Duret-Lutz, A. and Poitrenaud, D. (2004). Spot: an extensible model checking library using transition-based generalized büchi automata. *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1998). Property specification patterns for finite-state verification. In Ardis, M., editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15, New York. ACM Press.

Emerson, E. A. (1995). Temporal and modal logic. In *Handbook Of Theoretical Computer Science*, pages 995–1072. Elsevier.

Etessami, K. and Holzmann, G. J. (2000). Optimizing Büchi automata. In Palamidessi, C., editor, *Proceedings of the 11th International Conference on Concurrency Theory (Concur'00)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167, Pennsylvania, USA. Springer-Verlag.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY.

Lind-Nielsen, J. (2002). BuDDy: Binary Decision Diagram package. Release 2.2.

Minato, S.-I. (1992). Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proceedings of the third Synthesis and Simulation and Meeting International Interchange workshop (SASIMI'92)*, pages 64–73, Kobe, Japan.

Parutto, P. (2011). Improving degeneralization in Spot. Technical Report 1111, EPITA Research and Development Laboratory (LRDE).

Parutto, P. (2012). Partial order reduction methods for Spot. Technical Report 1205, EPITA Research and Development Laboratory (LRDE).

Somenzi, F. and Bloem, R. (2000). Efficient Büchi automata for LTL formulæ. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 247–263, Chicago, Illinois, USA. Springer-Verlag.

Tauriainen, H. (2008). LBTT. http://www.tcs.hut.fi/Software/lbtt/.

Tauriainen, H. and Heljanko, K. (2002). Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer*, 4(1):57–70.

Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., and Becker, B. (2006). Sigref - a symbolic bisimulation tool box. In Graf, S. and Zhang, W., editors, *4th International Symposium on Automated Technology for Verification and Analysis, ATVA 2006*, volume 4218 of *Lecture notes in Computer Science*, pages 477–492. Springer Verlag.