Combining Partial Order Reduction with Symbolic Model Checking

José Vander Meulen

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

April 19, 2012

Institute of Information & Communication Technologies,
Electronics and Applied Mathematics
(ICTEAM institute)
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Olivier **Bonaventure** (Chair)	UCL/ICTM/INGI
Gianfranco **Ciardo**	UC Riverside, USA
Baudouin **Le Charlier**	UCL/ICTM/INGI
Charles **Pecheur** (Advisor)	UCL/ICTM/INGI
Marco **Roveri**	FBK, Italy
Pierre **Wolper**	ULG, Belgium

Abstract

Model checking is an efficient technique for verifying properties on asynchronous systems. Unfortunately, it suffers from the so-called combinatorial state-space explosion problem. Two common approaches are used to fight this problem, with different perspectives. On the one hand, partial-order reduction (POR) methods explore a reduced state space in a property-preserving way. On the other hand, symbolic techniques use efficient structures such as binary decision diagrams (BDD's) to concisely encode and compute large state spaces. By using symbolic model checking algorithms, it is possible to verify systems with a very large number of states. However, in some cases, the size of the BDD structures can become unmanageable. Bounded Model Checking (BMC) uses SAT-solvers instead of BDD's to search for errors on bounded execution path. In practice, BDD-based approaches and BMC approaches are two fruitful complementary techniques. This thesis presents algorithms which, one way or another, combine symbolic model checking and partial-order reduction, allowing efficient verification of CTL_X and LTL_X properties on models featuring asynchronous processes.

At the root of our work was the ImProviso algorithm for computing reachable states [LST03], which combines POR and symbolic verification and the FwdUntil method that supports verification of a subset of CTL [INH96]. We present the PartialExploration algorithm which adapts and extends ImProviso to support the verification of a fragment of CTL_X. Then, the evalCTLX algorithm merges the PartialExploration algorithm with the classical backward algorithm to support the totality of CTL.

The evalLTLX algorithm checks LTL_X properties. We start from the tableau-based reduction of LTL verification of Clarke et al. [CGH97],

which translates an LTL problem into a fair path detection problem. While classical BDD-based model-checking would perform this search with a backward traversal, we use the forward-traversal approach of Iwashita et al. [INH96]. Part of the resulting computation amounts to computing the reachable state space; to that end we use our PartialExploration algorithm.

The BPE algorithm adapts our PartialExploration algorithm and bounded model checking techniques [BCC$^+$03] in an original way. The encoding to a SAT problem strongly reduces the complexity and nondeterminism of each transition step, allowing efficient analysis even with longer execution traces.

The Milestones model checker implements the algorithms developed in this thesis. It allows us to check the absence of deadlock, LTL properties, and CTL properties. In order to compare our approach to others, Milestones is able to translate a model into an equivalent Spin model or NuSMV model.

We use Milestones, as well as Spin and NuSMV to assess the scalability and the effectiveness of the approaches developed in this thesis. To that end, we model and verify four examples. For each of those systems, we compute both the whole state space and two reduced state space. Then, we check whether they verify various CTL$_X$ or LTL$_X$ properties. Those experiments show that in some cases the approaches developed in this thesis achieve an improvement with respect to traditional methods.

Finally, we review some approaches which are related to ours. To that end, we present the considered approaches in the same algorithmic framework as the one used throughout this thesis. Then, we compare them with our partial-order reduction method.

The main achievements of this thesis are three new algorithms, as well as a proof of their correctness, that allow efficient symbolic model-checking of asynchronous systems based on POR techniques, and the Milestones model checker that implements the three previous algorithms.

Acknowledgments

First of all, I would like to thank Charles Pecheur for the 6 years that he spent helping and guiding me while I was carrying out this thesis. Without his help and support, from a technical as well as from a human point of view, this thesis would have been much harder to finish. He gave me the freedom I needed and showed me how to avoid traps and pitfalls.

I would also like to thank Baudouin Le Charlier for the numerous conversations. In particular, I remember the discussions which focused on the algorithm constructions, and the ones about the way of constructing a logical reasoning. Those conversations showed me that it is possible to construct, at the very first try, a program which works in a correct way.

I am also thankful to the members of the Jury, Gianfranco Ciardo, Marco Roveri, Pierre Wolper, and Olivier Bonaventure for their useful comments on earlier versions of this thesis.

Contents

Chapter 1

Introduction

Nowadays, hardware and software systems have great significance in our daily lives. Such systems are everywhere, from the latest trendy smartphones to medical instruments. The structure of those systems can be very complex. For instance, it is not uncommon for a piece of software to be composed of millions of lines of code. Due to this complexity, it frequently happens that a system does not do what it is expected to do. This phenomenon is commonly know as a "bug". bugs. The consequences of such bugs might be unacceptable. For example, in June 1996, the maiden flight of the Ariane 5 launcher ended in an explosion as a result of a software error in the computer that was responsible for calculating the rocket's movement.

In order to develop error-free systems, it is becoming more and more important to develop methods to increase our confidence in the correctness of such systems. To achieve that goal, there is a wide range of validation methods such as testing, simulation, static analysis, deductive verification and model checking. Some of these techniques explore only some of the possible behaviors of the systems [Mye79], while others lead to an exhaustive exploration of all possibilities.

Model checking is a technique which exhaustively explores a system in order to verify whether a property is satisfied or not. It is used to verify finite state models of concurrent systems such as sequential circuit designs and communication protocols. It has a two main advantages: it is entirely automatic, and it produces a counterexample when an error is detected [CGP99]. It also suffers from two drawbacks. Firstly,

the property under consideration can be hard to express because it is generally translated into a rich formal logic. Secondly, model checking can be resource consuming because the size of the system can be extremely large. The goal of this thesis is to alleviate the problem of the resources needed when model checking techniques are applied on a system.

Model checking describes the behavior of a system by a state machine. Temporal logic is used to express some properties of that state machine. In a temporal logic, the usual operators of propositional logic are extended with temporal operators. There are several types of temporal logics, such as linear temporal logic (LTL and LTL_X), computational tree logic (CTL and CTL_X) or CTL*. By using temporal logic model checking algorithms, one can check automatically whether a given system, modeled as a state machine, satisfies a given temporal logic property.

In the 1980's, several researchers introduced temporal logic model checking algorithms, especially for LTL and CTL. McMillan achieved a breakthrough with the use of symbolic representations based on the use of Ordered Binary Decision Diagrams (BDD) [Bry86]. By using this approach, some systems with a very large number of states, i.e. more than 10^{20} states, can be verified [BCM+92]. However, in some cases, the size of the BDD structures can become unmanageable, and so other approaches have been developed. For instance, Bounded Model Checking (BMC) uses SAT-solvers instead of BDD's to search for errors on bounded execution paths [BCC+03]. BMC offers the advantage of polynomial space complexity and in practice, it has proven to provide competitive execution times in practice.

A common approach to verify a concurrent system is to compute the parallel composition of the processes involved. Unfortunately, the size of this composition is frequently prohibitive due, among other causes, to the modeling of concurrency by interleaving. The aim of partial-order reduction (POR) techniques is to reduce the number of interleaving sequences that must be considered. When a specification cannot distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyze only one of them [God96].

There is actually a lot of work which has been done in POR in model checking, but only little in symbolic model checking. Traditionally, both symbolic and POR techniques have been developed by two different

schools. Two approaches to combine both methods have been devised by Alur et al. [ABH+97] and Kurshan et al. [AJKP98]. In [LST03], Lerda et al. attempt to solve the problem called in-stack proviso in particular. POR algorithms are based on the idea of postponing transitions without affecting the property to be checked, and provisos are used to guarantee that no transition is postponed indefinitely. Lerda et al. introduce ImProviso, a new algorithm for model checking of software that combines the advantages of POR with symbolic exploration.

1.1 Presentation of This Thesis.

In this thesis we explore how symbolic model checking and partial-order reduction approaches can fit together. At the root of our work were two approaches:

- The *Two-Phase* algorithm and its symbolic version, the *ImProviso* algorithm, are two partial-order reduction approaches. Intuitively, both take as input a state machine M, and produce a reduced version M_R of M such that M and M_R respect the same properties.
- The *forward CTL model checking* approach of H. Iwashita et al. [INH96]. Classically, symbolic model checking verifies if a CTL property is satisfied in a backward manner. Intuitively, it starts from the part of the state machine that one wants to reach. Then, it looks for all states having a path leading to that part. Forward CTL model checking takes the opposite approach. It starts from the initial states and looks for all states which are reachable from them.

We adapt and merge those two approaches to check either LTL_X or CTL_X properties by means of BDD-based model checking or SAT-based model checking (c.f. Table 1.1).

In the first part of this thesis, we start by constructing the PartialExploration algorithm. It is a variant of BDD-based ImProviso algorithm. It is used to incorporate partial-order reduction into the forward model checking approach. Then, we develop the evalCTLX algorithm which relies on our PartialExploration method. It checks whether a system verifies a CTL_X property.

Table 1.1: The Three Approaches Introduced in This Thesis

	CTL_X	LTL_X
BDD	evalCTLX	evalLTLX
SAT	—	BPE

In the second part of this thesis, we use the PartialExploration algorithm to construct the evalLTLX algorithm. It verifies whether a system respect a LTL_X properties. As the evalCTLX algorithm, it is a BDD-based algorithm.

In the third part, we present the BPE algorithm which combines together the bounded model checking approach and a variant of our PartialExploration algorithm. Intuitively, from a model and a property, the BMC method constructs a propositional formula which represents a finite unfolding of the transition relation and the property. Our method proceeds in the same way, but instead of using the entire transition relation during the unfolding of the model, we only use a safe subset based on POR considerations. This produces a propositional formula which is well suited for most modern SAT solvers.

In the fourth part, we present the Milestones model checker, which implements the algorithms developed within the context of this thesis. Milestones defines a language for describing transition systems. The absence of deadlock, LTL or CTL temporal logic properties can be verified on such systems. In order to compare our approaches against the state-of-the-art, Milestones can also translate its model into a Promela model [Hol97] or into a NuSMV model [CCGR99]. In order to make the comparison as fair as possible, the resulting state machines are similar to those generated by Milestones.

In the fifth part, we illustrate and evaluate the techniques described in this thesis by applying them to four examples. For each of these case studies, we compute its state space as well as a reduced state space. Then we verify some CTL_X or LTL_X properties with one or more of the different methods developed in this this thesis, as well as comparable methods from other authors.

In the sixth part, we present and discuss six different symbolic methods which are directly or indirectly related to the ones presented in this thesis. The first three approaches are related to ours because they combine partial-order reduction and symbolic method in one way or another. The three others tackle the same problems, but in a different way. All the algorithms presented in this part are expressed with the same conventions as the ones introduced in the previous parts of this thesis. In particular, for each of them, we provide specifications as well as loop invariants.

This thesis presents the following scientific contributions:

1. Three new algorithms that allow efficient symbolic model-checking of asynchronous systems based on POR techniques:

   (a) a new algorithm that combines and extends the Two-Phase algorithm and forward symbolic model-checking for verifying CTL_X properties.

   (b) a new algorithm that checks LTL_X properties by combining and adapting our new algorithm mentioned in (1a) and the Clarke et al.'s tableau-based symbolic LTL model checking,

   (c) a new algorithm that checks LTL_X properties by combining and extending our new algorithm mentioned in (1a) and bounded model checking to allow the verification of LTL properties,

2. A new model checker that implements the three previous algorithms as well as other classical approaches for comparison purposes. It also translates its programs into NuSMV or Spin.

3. A meticulous development and proof of correctness of these three algorithms, as well as other approaches from which they derive or to which they compare, all within a unifying mathematical and algorithmic framework.

4. An evaluation of the techniques described in this thesis on four models.

1.2 Publication

Our research has given rise to four publications. We stress that in this thesis we rename the algorithms which were initially presented in the following publications:

- The PartialExploration and the evalCTLX algorithms were originally published in the *13th International Workshop on Formal Methods for Industrial Critical Systems* in 2008 [VP08].
- The BPE algorithm was originally published in the *Communicating Process Architectures 2009 Conference* in 2009 [VP09].
- The evalLTLX algorithm was originally published in the *Third NASA Formal Methods Symposium* in 2011 [VP11a].
- The Milestones model checker was originally published in the *Third NASA Formal Methods Symposium* in 2011 [VP11b].

1.3 Structure of the Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 recalls some background concepts, definitions and notations that are used throughout the thesis. In particular, it introduces the model checking problem, LTL and CTL temporal logics, partial-order reduction concepts, and symbolic model checking.
- Chapter 3 starts by presenting two existing POR algorithms: the Two-Phase algorithm and its symbolic version ImProviso. From those two algorithms, we construct the PartialExploration algorithm which combines partial-order reduction and set-based model checking. Then, we develop the evalCTLX algorithm which relies on the PartialExploration algorithm, and which is used to check whether a system verifies a CTL property or not.
- Chapter 4 starts by presenting how a LTL property can be verified with set-based model checking. Then, we present the evalLTLX algorithm which merges POR and set-based model checking to check LTL properties.

- Chapter 5 starts by introducing bounded model checking. It then constructs the BoundedPartialExploration (BPE) algorithm which combines partial order reduction and bounded model checking to check LTL properties.
- Chapter 6 presents the Milestones model checker. It implements the algorithms presented in the three previous chapters, as well as other classical approaches.
- Chapter 7 presents an evaluation of the three algorithms presented in this thesis. To that end, we check four examples of models.
- Chapter 8 reviews related work.
- Chapter 9 gives a general conclusion of our work, with perspectives on future research.

Chapter 2

Background

In this chapter, we describe the principles of model checking, its advantages and its drawbacks. Moreover, we introduce the approaches which were developed to overcome the inherent limitations of model checking.

When someone wants to check a system, the first step consists in making a model of that system. For that purpose, most of the available model checkers provide a description language which is similar to a programming language, e.g. the NuSMV language [CCGR99], the Promela language [Hol97], the HyDI language [CMT11], the Verilog language [IEE95]. The resulting model is then automatically translated into a finite state machine.

The second task consists in providing a formal property. Typical properties are functional properties (does the system do what it is supposed to do), reachability properties (is it possible to reach some predetermined states), safety properties (make sure nothing bad ever happens), liveness properties (something good will eventually happen). Generally, these properties are expressed with a temporal logic. Intuitively, a temporal logic is an extension of the propositional calculus with temporal operators. Given a finite state machine M, such logics allow us to express properties about traces of M. For example CTL, and LTL are two such logics [CGP99].

Model checking takes as inputs a finite state machine and a property. It then performs a exhaustive exploration to verify that the state machine satisfies the property. At the end of the process, the model checker indicates whether the property is valid or not. If it is not valid, a counter-

example is generally provided. Unfortunately, model checking algorithms sometimes fail because the given models are too large to fit into memory, or because the algorithms take to much time to give an answer. *Symbolic model checking* and *partial-order reduction* are two techniques which try to alleviate that problem. The next sections of this chapter are mainly devoted to explaining them.

Model checking suffers from three problems:

- It verifies a model of the system and not the real system itself. Hence, if the model encoding does not reflect reality precisely enough, the model checking results might not reflect reality as well.

- It suffers from the *state-space explosion* problem. The model can be extremely large, or even infinite. In consequent, the available memory might not be sufficient to represent it, or the time which is needed by the verification process might be prohibitive. Partial-order reduction approaches attempt to tackle that problem by not verifying the model itself, but an equivalent reduced version with respect to a given class of properties.

- The formal logics which are generally employed to expresses a property are rich and complex. Therefore, it can be difficult to write a property which expresses the required behavior. Sometimes, it is more difficult to write it than to model the system. Therefore, it can be hard to determine if a non-trivial property reflects what one wants to say. Moreover, when an error is detected, it could also be hard to determine whether the error comes from the property or from the model.

Despite these problems, model checking has some precious benefits:

- It is a general mechanism which is applicable to a wide range of applications such as hardware components, communication protocols, etc.

- Contrary to interactive theorem provers, it is fully automatic.

- Contrary to testing and simulation methods, it is exhaustive.

- Most of the of the available model checkers provide detailed counterexamples when an error is detected.

The remainder of this chapter is structured as follows. Section 2.1 introduces the concept of transition systems which are used to model a system. Section 2.2 presents three temporal logics: CTL*, CTL, and LTL. Section 2.3 is devoted to four bisimulation approaches which allow one to deduce if two transition systems have equivalent behaviors. Section 2.4 introduces the partial-order reduction method. Section 2.5 presents the symbolic model checking approach. Section 2.6 gives conclusions.

2.1 Modeling a System

We represent the behavior of a system as a transition system, with labelled transitions and propositions interpreted over states.

Definition 2.1 (Transition System)**.** *A* transition system *is a structure* $M = (A, AP, S, R, I, L)$ *where:*

- A *is a set of actions, and*
- AP *is a set of atomic propositions, and*
- S *is a set of states, and*
- $R \subseteq S \times A \times S$ *is a transition relation, and*
- $I \subseteq S$ *is a set of initial states, and*
- $L : E \to 2^{AP}$ *is an interpretation function such that* $S \subseteq E$*.*

Figure 2.1 graphically represents a transition system $M_{ex} = (A, AP, S, R, I, L)$ where:

- The actions of A correspond to the labels on the arrows.
- The set AP contains four propositions `g, x, y,` and `z`.
- Each state of S is a 4-tuple $(g, x, y, z) \in \{0, 1\}^4$ which gives a valuation of the four propositions `g, x, y,` and `z`. The value of a proposition p in s is noted value(s, p). Figure 2.1 represents each state by a circle.

- The transition relation is graphically represented by the arrows between the states.

- The unique initial state is $(0, 0, 0, 0)$.

- The labeling function maps each state $s = (g, x, y, z)$ to the set of propositions which have a value equal to 1 in s, i.e. $\{p \in \{g, x, y, z\} \mid \text{value}(s, p) = 1\}$, e.g. $L(0110) = \{x, y\}$.

To be more precise, only the states which are reachable by an finite sequence of arrows from the initial state (0000) are displayed. Those states form the *reachable state space*. In general, we are only interested in such states.

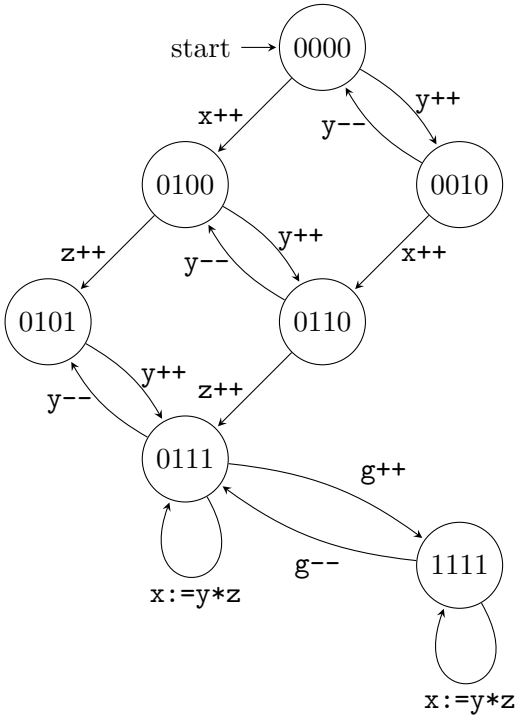


Figure 2.1: A Transition System where each state is a 4-tuple $(v(g), v(x), v(y), v(z)) \in \{0, 1\}^4$

From a theoretical point of view, a transition system can be either finite or infinite. In this document, a transition system is a *finite transition system* when both its set of actions A and its set of of states S are finite. When a transition system in not finite, we call it an *infinite transition system*. In practice, the model checking algorithms which are presented in this thesis explore either a finite transition system or a finite part of an infinite transition system. Despite that, some concepts are easier to explain on infinite transition systems than on finite ones. In the rest of the document, the transition systems which are manipulated are finite transition systems, except if the contrary is explicitly mentioned.

Throughout this document, we use the following concepts and notations when we reason about transition systems. In the remainder of this section, we suppose that $M = (A, AP, S, R, I, L)$ is a transition system, $S' \subseteq S$, and $R' \subseteq R$.

- When both the set of actions A and the set of propositions AP are not relevant to the current context, or when the context is clear, we often write $M = (S, R, I, L)$ instead of $M = (A, AP, S, R, I, L)$.
- We write $s \xrightarrow{a}_{R'} s'$ for $(s, a, s') \in R'$. When the context is clear, we write $s \xrightarrow{a} s'$ instead of $s \xrightarrow{a}_{R'} s'$.
- We write $s \longrightarrow_{R'} s'$ as "there exists an action $a \in A$ such that $(s, a, s') \in R'$." When the context is clear, we write $s \longrightarrow s'$ instead of $s \longrightarrow_{R'} s'$.
- The states s and s' are respectively the *source state* and the *target state* of $s \xrightarrow{a}_{R'} s'$, and $s \longrightarrow_{R'} s'$.
- An action a is *enabled* in a state $s \in S$ with respect to R' if and only if there is a state $s' \in S$ such that $s \xrightarrow{a}_{R'} s'$.
- The set enabled(R', S') is the set of enabled actions of states of S' with respect to R', i.e. $\{a \in A \mid \exists s' \in S' \cdot \exists s \in S \cdot s' \xrightarrow{a}_{R'} s\}$. When the context is clear, we write enabled(S') instead of enabled(R', S').
- A state $s \in S'$ is a *dead state* with respect to R' if and only if no actions are enabled with respect to R' and to states of S'. We write dead(R', S') for the set of dead states of S' with respect to R', i.e.

$\{s \in S' \mid \text{enabled}(R', s) = \emptyset\}$. When the context is clear, we write $\text{dead}(S')$ instead of $\text{dead}(R', S')$.

- When a state s is a dead state with respect to the whole transition relation R, s is a *deadlock*.
- If the transition relation R' is a *transition function*: $R' : S \times A \to S$, we say that R' is *deterministic*. If R is deterministic, we say that M is a *deterministic transition system*. In such a case, $R(s, a)$ represents the state which results from the application of R to arguments $s \in S$ and $a \in A$. If M is not deterministic, we say that it is a *nondeterministic transition system*.
- The transition relation R is a *total transition relation* if all the states of S have at least one enabled action. A transition relation which is not total is a *partial transition relation*. In the rest of the document, the transition systems which are manipulated have a total transition relation, except if the contrary is explicitly mentioned.
- The set of all *successors* of S' with respect to R', or the *post-image* of S' with respect to R', is noted $\text{post}(R', S')$. It is the set $\{s \in S \mid \exists s' \in S' \cdot s' \longrightarrow_{R'} s\}$. When the context is clear, we write $\text{post}(S')$ instead of $\text{post}(R', S')$.
- The set of all the *predecessors* of S' with respect to R', or the *pre-image* of S' with respect to R', is noted $\text{pre}(R', S')$. It is the set $\{s \in S \mid \exists s' \in S' \cdot s \longrightarrow_{R'} s'\}$. When the context is clear, we write $\text{pre}(S')$ instead of $\text{pre}(R', S')$.
- A transition relation $R'' \subseteq S \times A \times S$ is the inverse transition relation of R' if and only if for all $a \in A$ and all $s, s' \in S$, $s \xrightarrow{a}_{R'} s'$ if and only if $s' \xrightarrow{a}_{R''} s$. The inverse transition relation of R' is noted R'^{-1}.
- A transition system $M'' = (A, AP, S, R'', I, L)$ is an inverse transition system of M if and only if $R'' = R^{-1}$. The inverse transition system of M is noted M^{-1}.
- A *finite trace* (or *finite path*) of M with respect to R' is a finite sequence $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ such that each s_i of π

belongs to S, and $\forall i \in [0, \ldots, n[\cdot s_i \xrightarrow{a_i}_{R'} s_{i+1}$. The length of π is the number of transitions of π. It is noted $|\pi|$ and is equal to n.

- An *infinite trace* (or *infinite path*) of M with respect to R' is a infinite sequence $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ such that each s_i of π belongs to S, and $\forall i \in \mathbb{N} \cdot s_i \xrightarrow{a_i} s_{i+1}$. Its length is noted $|\pi|$. It is infinite. In other words, it is equal to ∞.

- A trace π of M with respect to R' is *maximal* if and only if either it is infinite, or it is finite and its last state is dead with respect to R'.

- Let $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} s_i \xrightarrow{a_i} s_{i+1} \xrightarrow{a_{i+1}} \ldots$ be a finite or infinite trace, a *prefix* of π is a path $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} s_i$.

- Let $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} s_i \xrightarrow{a_i} s_{i+1} \xrightarrow{a_{i+1}} \ldots$ be a finite or infinite trace, a *suffix* of π is a path $s_i \xrightarrow{a_i} s_{i+1} \xrightarrow{a_{i+1}} \ldots$.

- Let $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} s_i \xrightarrow{a_i} \ldots$ be a finite or infinite trace, $\pi^i = s_i \xrightarrow{a_i} s_{i+1} \xrightarrow{a_{i+1}} \ldots$ denotes the suffix of π starting at s_i.

- The set of traces (resp. maximal traces) of M with respect to R' and S' is noted $\mathrm{tr}\,(R', S')$ (resp. $\mathrm{mtr}\,(R', S')$). It is the set of all paths (resp. maximal paths) with respect to R' which start from a state of S'.

- The set of traces of M of length k with respect to R' and S' is noted $\mathrm{tr}\,(k, R', S')$. It is the set of all paths of length k with respect to R' which starts from a state of S'. The set of traces of M of length k is noted $\mathrm{tr}(k, M)$. It is the set $\mathrm{tr}(k, R, I)$.

- The *reachable state space* of M with respect to R' and S' is noted $\mathrm{post}^*(R', S')$. It is the set of states which are reachable from a state of S', i.e. $\{s_k \in S \mid \exists s_0 \in S' \cdot \exists \pi \in tr(R', S') \cdot \pi = s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{k-1}} s_k\}$. The reachable state space of M is noted $\mathrm{post}^*(M)$. It is the set $\mathrm{post}^*(R, I)$.

- We occasionally restrict the set of initial states to a singleton. In such a case, and when the context is clear, we write $M = (A, AP, S, R, i, L)$ instead of "$M = (A, AP, S, R, I, L)$ where I is a singleton".

We write $M \sqsubseteq M'$ iff M is a *sub-transition system* of M', in the following sense:

Definition 2.2 (Inclusion of transition system)**.** *Let $M = (A, AP, S, R, I, L)$, $M' = (A, AP, S', R', I', L')$ be two transition systems over the same sets of actions and atomic propositions. M is a* sub-transition system *of M', denoted $M \sqsubseteq M'$, if and only if $S \subseteq S'$, $R \subseteq R'$, $I \subseteq I'$, $L(s) = L'(s)$ for all $s \in S$,*

If $M \sqsubseteq M'$, each path of M is a path of M'.

Lemma 2.3. *if $M \sqsubseteq M'$ then* $\mathrm{tr}(M) \subseteq \mathrm{tr}(M')$.

Given a subset of AP' of AP, it can happen that an action a never modifies the propositions AP' for all states s where it is fired. Such an action is called an *invisible action*:

Definition 2.4 (Invisibility of an action)**.** *Given $M = (A, AP, S, R, I, L)$ a transition system, $AP' \subseteq AP$, and a action $a \in A$. The action a is invisible with respect to AP' if and only if for all $s, s' \in S$, such that $(s, a, s') \in R$, $L(s) \cap AP' = L(s') \cap AP'$. An action is visible if and only if it is not invisible.*

For instance, the action `y++` of Figure 2.1 is invisible with respect to $AP_i = \{g, x\}$. But it is visible with respect to the set $AP_i = \{x, y\}$ because $(0000) \xrightarrow{\texttt{y++}} (0010)$, $L(0000) \cap \{x, y\} = \emptyset$, and $L(0010) \cap \{x, y\} = \{y\}$.

2.1.1 Computation Tree

Given a transition system M and one of its states s, a *computation tree* which represents all the execution paths from s can be built. It is a transition system. Its transition relation forms a tree. It contains a root "equivalent" to s. It is generated by unwinding M from the root state into a tree [AU73]. It can also be seen as a kind of *labelled prefix tree* [KNU73] constructed from all the paths of M which start at s.

Definition 2.5 (Computation Tree)**.** *Given a transition system* $M = (S, R, I, L)$*, a state* $s \in S$ *of* M*, and a transition system* $M_{ct} = (S_{ct}, R_{ct}, \{s_{ct}\}, L_{ct})$*.* M_{ct} *is a computation tree of* M *if and only if* R_{ct} *is a tree, and there is a bijective function* B *from* $\mathrm{mtr}(M)$ *to* $\mathrm{mtr}(M_{ct})$ *which satisfies the following condition:*

For all $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots$ *and* $\pi' = s'_0 \xrightarrow{a_0} s'_1 \xrightarrow{a_1} \dots$ *such that* $B(\pi) = \pi'$*:*

(1) $|\pi| = |\pi'|$*, and*

(2) $\forall i \in [0, \dots, |\pi|[\cdot L(s_i) = L_{ct}(s'_i)$*.*

Figure 2.2 shows a part of the M_{ex} computation tree which is derived from (0000).

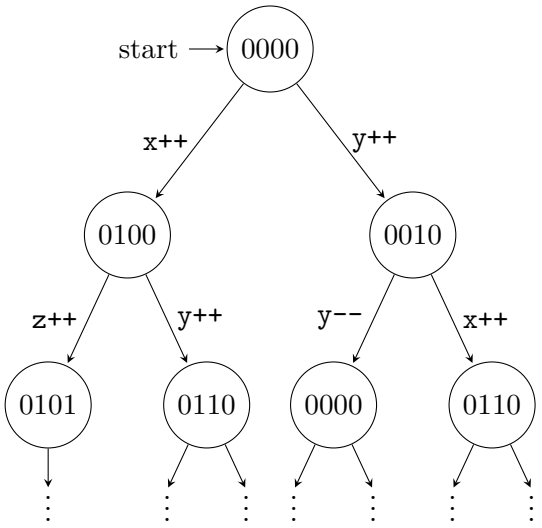


Figure 2.2: M_{ex}'s Computation Tree

We notice that a finite transition system M generally induces an infinite computation tree. It arises when there exists at least one infinite path from s. In other words, it arises when M contains a cycle. In particular, it always happens when R is total. Conversely, an a priori infinite transition system M can induce a finite computation tree. It only happens when there is a finite number of finite paths which start at s.

We point out that for each s there is a "unique" computation tree. More rigorously, for each s it is possible to derive at least one computation by unrolling M from the tree root. Moreover, if from s two computation trees M_{ct_1} and M_{ct_2} are derived, M_{ct_1} and M_{ct_2} are isomorphic, in the sense that there exists a bijection from post$^*(M_{ct_1})$ to post$^*(M_{ct_2})$ such that when two states are correlated by f, they share the same labeling. We write $CT(M)$ for designating the "unique" computation tree of M.

Finally, we stress that roughly speaking M and its computation tree have exactly the same behavior (c.f. Section 2.3.4). The first part of the next section presents the *Computation Tree Logic* CTL*. Given a transition system M, it describes properties of $CT(M)$.

2.2 Temporal Logics

In this document, we express the properties to be verified on a transition system in temporal logics. We consider either the *Computation tree Logic* (CTL) or the *Linear Temporal Logic* (LTL). Both are extensions of the propositional calculus. They define the X, F, G and U operators to reason about paths. Two quantifiers A and E are also added. Intuitively, given a states s and a temporal property f, $\mathsf{A}f$ states that all paths which start from s respect f, $\mathsf{E}f$ states there exists a path from s which satisfies f. Given the path π, $\mathsf{X}f$ (next f) states that f holds in the second state of π, $\mathsf{G}f$ (globally f) states that f holds in all states of π, $\mathsf{F}f$ (finally f) states that f holds in some state of π, $f\,\mathsf{U}\,g$ (f until g) states that g holds in some state s_i of π and, for all previous states of s_i, f holds.

2.2.1 Syntax of CTL*

Before explaining CTL and LTL, we describe the syntax and the semantic of CTL*. Actually, CTL and LTL are two different CTL* sublogics. CTL* allows one to define two types of formulae: *state formulae* and *path formulae*. The truth value of a state formula is evaluated on a specific state, while the truth value of a path formulae depends on a specific path. Let AP be a set of atomic propositions. A formula is a CTL* formula if and only if it is a finite formula which can be derived from the syntax below. The syntax of a state formula is given by the following rules:

(1) The symbols true and false are state formulae.

(2) For all $p \in AP$, p is a state formula.

(3) If g_1 is a state formula, $\neg g_1$ is a state formula.

(4) If g_1 and g_2 are state formulae, $g_1 \wedge g_2$, and $g_1 \vee g_2$ are state formulae.

(5) If p_1 is a path formula, $\mathsf{A}p_1$ and $\mathsf{E}p_1$ are state formulae.

The syntax of a path formula is given by the following rules:

(6) If g_1 is a state formula, then g_1 is also a path formula.

(7) If p_1 is a path formula, then $\neg p_1$, $\mathsf{X}\, p_1$, $\mathsf{F}\, p_1$, and $\mathsf{G}\, p_1$ are path formulae.

(8) If p_1 and p_2 are path formulae, then $p_1 \vee p_2$, $p_1 \wedge p_2$, and $p_1 \,\mathsf{U}\, p_2$ are path formulae.

Invisibility of a transition with respect to a property f

The notion of invisible transitions with respect to a set of propositions (c.f. Definition 2.4) can be extended to CTL* properties.

Definition 2.6 (Invisibility of a transition with respect to a property f)**.** *Given $M = (A, AP, S, R, I, L)$ a transition system, an action $a \in A$, and a CTL* property f. The transition a is invisible with respect to f if and only if a is invisible with respect to the set of propositions which appear in f.*

In the rest of this section, we suppose that:

- $M = (A, AP, S, R, I, L)$ is a transition system such that R is total, and
- s is a state of S, and
- π is an infinite path of M, and
- g_1 and g_2 are two state formulae defined on AP, and
- p_1 and p_2 are two path formulae defined on AP.

2.2.2 Semantics of CTL*

The notation $M, s \models g_1$ (resp. $M, s \not\models g_1$) means that the state formula g_1 holds (resp. not hold) in s. The notation $M \models g_1$ (resp. $M \not\models g_1$) means that the state formula g_1 holds (resp. not hold) for all the initial states of M, i.e. for all $i \in I$. The notation $M, \pi \models p_1$ (resp. $M, \pi \not\models p_1$) means that the path formula p_1 holds (resp. not hold) along π. The notation $M \models p_1$ (resp. $M \not\models p_1$) means that the path formula p_1 holds (resp. not hold) along all paths of M. When either M is not relevant, or the context is clear we write $s \models g_1$ (resp. $\pi \models p_1$), instead of $M, s \models g_1$ (resp. $M, \pi \models p_1$). The *language of the state formula* g_1 is noted $\mathcal{L}(g_1)$. It is the set of states which satisfy g_1, i.e. $\{s \in S \mid M, s \models g_1\}$. The *language of the path formula* p_1 is noted $\mathcal{L}(p_1)$. It is the set of M paths which satisfies p_1, i.e. $\{\pi \in \mathrm{tr}(M) \mid M, \pi \models p_1\}$. The semantics of CTL* is now defined:

(1) $M, s \models \text{true}$.

(2) $M, s \not\models \text{false}$.

(3) $M, s \models p \Leftrightarrow p \in L(s)$.

(4) $M, s \models \neg g_1 \Leftrightarrow M, s \not\models g_1$.

(5) $M, s \models g_1 \wedge g_2 \Leftrightarrow M, s \models g_1$ and $M, s \models g_2$.

(6) $M, s \models g_1 \vee g_2 \Leftrightarrow M, s \models g_1$ or $M, s \models g_2$.

(7) $M, s \models \mathsf{E}\, p_1 \Leftrightarrow$ there is an infinite path π starting at s such that $M, \pi \models p_1$.

(8) $M, s \models \mathsf{A}\, p_1 \Leftrightarrow$ for every infinite path π starting from s, $M, \pi \models p_1$.

(9) $M, \pi \models g_1 \Leftrightarrow M, s_0 \models g_1,$ where s_0 is the first state of π.

(10) $M, \pi \models \neg p_1 \Leftrightarrow M, \pi \not\models p_1$.

(11) $M, \pi \models p_1 \wedge p_2 \Leftrightarrow M, \pi \models p_1$ and $M, \pi \models p_2$.

(12) $M, \pi \models p_1 \vee p_2 \Leftrightarrow M, \pi \models p_1$ or $M, \pi \models p_2$.

(13) $M, \pi \models \mathsf{X}\, p_1 \Leftrightarrow M, \pi^1 \models p_1$.

(14) $M, \pi \models \mathsf{F}\, p_1 \Leftrightarrow$ there exists a $k \in \mathbb{N}$ such that $M, \pi^k \models p_1$.

(15) $M, \pi \models \mathsf{G}\, p_1 \Leftrightarrow$ for all $k \in \mathbb{N}$, $M, \pi^k \models p_1$.

(16) $M, \pi \models p_1\, \mathsf{U}\, p_2 \Leftrightarrow$ there exists a $k \in \mathbb{N}$, such that the following conditions hold:

   (a) $M, \pi^k \models p_2$, and

   (b) for all $j \in [0, \ldots, k[$, $M, \pi^j \models p_1$.

The CTL^*_X logic is the fragment of CTL* which consists of all the CTL* formulae which do not contain any X operator. It is show in [CGP99] that the operators $\vee$, $\neg$, X, U, and E are sufficient to express any other CTL* formulae. Actually, if f_1 and f_2 are CTL* formulae. the following equivalences are valid:

(1) The formula $f_1 \wedge f_2$ is equivalent to the formula: $\neg(\neg f_1 \vee \neg f_2)$.

(2) The formula $\mathsf{F} f_1$ is equivalent to the formula: true $\mathsf{U}\, f_1$.

(3) The formula $\mathsf{G} f_1$ is equivalent to the formula: $\neg\mathsf{F}\, \neg f_1$.

(4) The formula $\mathsf{A} f_1$ is equivalent to the formula: $\neg\mathsf{E}\, \neg f_1$.

We now describe two sublogics of CTL*, the *Computation Tree Logic* CTL, and the *Linear Temporal Logic* LTL. Given a transition system M, the former reasons about the whole computation tree (c.f. Section 2.1.1) of M, while the latter reasons about the traces of M.

CTL In CTL, every temporal operator F, G, U, and X has to be preceded by a path quantifier, i.e A, and E. More formally, the syntax of CTL is the same as the one of CTL* but without rule (6). Besides, rule (7) and rule (8) are replaced by the following ones:

(7) If p_1 is a state formula, then $\mathsf{X}\, p_1$, $\mathsf{F}\, p_1$, and $\mathsf{G}\, p_1$ are path formulae.

(8) If p_1 and p_2 are state formulae, then $p_1\, \mathsf{U}\, p_2$ is a path formula.

Finally, a CTL formula has to be a state formula.

For instance, the property $\mathsf{AG}\,\mathsf{EF}\,g$ is a well-formed CTL formula, but $\neg \mathsf{F} g$ is not. Besides, the transition system M_{ex} verifies $\mathsf{AG}\,\mathsf{EF}\,g$. Actually, all the M_{ex} reachable states satisfy $\mathsf{EF}\,g$, and so all the paths from (0000) satisfy $\mathsf{AG}\,\mathsf{EF}\,g$. The CTL_X logic is the fragment of CTL which consists of all the CTL formulae which do not contain any X operators.

LTL In LTL, each formula has the form p_1 where p_1 does not contain any path quantifiers. It is evaluated to true if and only if $M \models \mathsf{A}\ p_1$. In other words, a transition system M satisfies p_1 if and only if all the traces of M satisfy p_1. Formally, the syntax of LTL is defined with the same syntax as the one of CTL* but without rule 5. For instance, $\mathsf{FG}\,g$ is a LTL formula which is not verified by M_{ex}. Actually, the M_{ex} path $(0000) \xrightarrow{\mathtt{y++}} (0010) \xrightarrow{\mathtt{y--}} (0000) \xrightarrow{\mathtt{y++}} (0010) \xrightarrow{\mathtt{y--}} (0000)\ldots$ does not contain any states s_i such that $g \in L(s_i)$. So, it cannot satisfy the property $\mathsf{FG}\,g$. As CTL_X, LTL_X is the fragment of LTL which consists of all the LTL formulae which do not contain any X operators.

It is shown in [CD89] that the CTL logic and the LTL logic are not comparable, i.e. they have different expressive powers. For instance, there is no corresponding LTL formula of the CTL formula $\mathsf{AG}\,\mathsf{EF}\,g$, and it is impossible to translate the LTL formula $\mathsf{FG}\,g$ into CTL. The conjunction of the two previous formulae $(\mathsf{AG}\,\mathsf{EF}\,p) \wedge (\mathsf{AFG}\,g)$ is a CTL* formula but it is not expressible in either LTL or CTL.

2.3 Bisimulation Relations

Intuitively, two transition systems M and M' are *bisimilar* with respect to a class of properties, when M and M' respect the same properties of this class. More precisely, a *simulation* is a binary relation between the states (or the paths) of two transition systems M and M'. It connects the states (or the paths) of M to the states (or the paths) of M' such that the former respect the same properties as the latter. If there exists a single relation that is a simulation from M to M' and a simulation from M' to M, there is a *bisimulation* between M to M'. We say that M and M' are bisimilar. For instance, minimization modulo bisimulation techniques are used in model checking to reduce the number of states of M while preserving some kinds of properties, e.g. deadlocks,

LTL_X, or CTL_X. The literature proposes a large number of variants of bisimulation relations [GKPP99, Mil89]. This section describes four kinds of bisimulation relations used in the sequel.

2.3.1 The Stuttering Equivalence

This section introduces the *stuttering equivalence*. It is the least restrictive bisimulation presented in this chapter. It preserves LTL_X properties [CGP99]. Intuitively, given a LTL_X property f and two transition systems M and M' which are stutter-equivalent, M contains a path π which violates f if and only if M' contains a path π' which violates f.

Definition 2.7 (Stuttering Equivalence [CGP99])**.** *Given a set $AP' \subseteq AP$ of propositions which appears in f, two transition systems $M = (A, AP, S, R, I, L)$ and $M' = (A', AP, S', R', I', L')$ are stuttering equivalent if and only if the following conditions hold:*

(1) For every infinite path π of M that starts at a $i \in I$, there is an infinite path π' in M' that starts at a $i' \in I'$, a partition $P_1, P_2, \ldots$ of π, and a partition $Q_1, Q_2, \ldots$ of π' such that for each $i \geq 0$, P_i and Q_i are nonempty and finite, and all states $s \in P_i$ have the same labeling with respect to AP' as every state in Q_i, i.e. $\forall s \in P_i \cdot \forall s' \in Q_i \cdot L(s) \cap AP' = L'(s') \cap AP'$.

(2) For every infinite path π' of M' that starts at a $i' \in I'$, there is an infinite path π in M that starts at a $i \in I$, a partition $Q_1, Q_2, \ldots$ of π', and a partition $P_1, P_2, \ldots$ of π such that for each $i \geq 0$, Q_i and P_i are nonempty and finite, and all states $s \in Q_i$ have the same labeling with respect to AP' as every state in P_i, i.e. $\forall s \in Q_i \cdot \forall s' \in P_i \cdot L'(s') \cap AP' = L(s) \cap AP'$.

2.3.2 The Stuttering Bisimulation

This section is devoted to the *stuttering bisimulation* which is stronger than stuttering equivalence. It preserves CTL^*_X properties [BCG88]. On the one hand, stuttering equivalence connects the paths of two transition systems M and M'. On the other hand, stuttering bisimulation connects the states of M to the states of M'.

Definition 2.8 (Stuttering Bisimulation [GKPP99]). *Given a set $AP' \subseteq AP$ of propositions, a relation $B \subseteq S \times S'$ is a* stuttering simulation *relation between two structures $M = (A, AP, S, R, I, L)$ and $M = (A', AP, S', R', I', L')$ if and only if for every $s \in S$, $s' \in S'$ such that $(s, s') \in B$, the following conditions hold:*

(1) $L(s) \cap AP' = L'(s') \cap AP'$

(2) For every infinite path π of M that starts at s, there is a infinite path π' in M' that starts at s', a partition $P_1, P_2, \ldots$ of π, and a partition $Q_1, Q_2, \ldots$ of π' such that for each $i \geq 0$, P_i and Q_i are nonempty and finite, and every state in P_i is related by the relation B to every state in Q_i.

B is a stuttering bisimulation relation *if and only if both B and its inverse relation B^{-1} are stuttering simulations. M and M_1 are stuttering-bisimilar if and only if there is a stuttering-bisimulation relation B such that*

(3) $\forall i \in I \cdot \exists i' \in I' \cdot (i, i') \in B$, and

(4) $\forall i' \in I' \cdot \exists i \in I \cdot (i, i') \in B$.

2.3.3 The Visible Bisimulation

This section introduces the visible bisimulation. It preserves CTL^*_X properties and action-based logics such as Hennessy-Milner logic and its derivatives [HM85]. Thus, it also preserves CTL_X and LTL_X properties. Intuitively, the visible-bisimulation associates two states s and s' that are impossible to distinguish, in the sense that if from s a visible action a is attainable in the future, a also belongs to future of s'. In [GKPP99], it is shown that a visible bisimulation is also a stuttering bisimulation.

Definition 2.9 (Visible-Bisimulation [GKPP99]). *Given a set $AP' \subseteq AP$ of propositions, a relation $B \subseteq S \times S'$ is a visible-simulation relation between two structures $M = (A, AP, S, R, I, L)$ and $M' = (A, AP, S', R', I', L')$ if and only if for every $s \in S$, $s' \in S'$ such that $(s, s') \in B$, the following conditions hold:*

(1) $L(s) \cap AP' = L'(s') \cap AP'$

(2) Let $s \xrightarrow{a} t$. *There are two cases:*

If a *is invisible with respect to* AP', *then there exists a path* $s' = s'_0 \xrightarrow{b_0} s'_1 \xrightarrow{b_1} \ldots \xrightarrow{b_{n-1}} s'_n$ *in* M', *such that* $(s, s'_i) \in B$ *for* $0 \leq i < n$, $B(t, s'_n)$, *and* b_i *is invisible with respect to* AP' *for* $0 \leq i < n$.

If a *is visible with respect to* AP', *then there exists a path* $s' = s'_0 \xrightarrow{b_0} s'_1 \xrightarrow{b_1} \ldots \xrightarrow{b_{n-1}} s'_n \xrightarrow{a} s'_{n+1}$ *in* M', *such that* $(s, s'_i) \in B$ *for* $0 \leq i \leq n$, $B(t, s'_{n+1})$, *and* b_i *is invisible with respect to* AP' *for* $0 \leq i < n$.

(3) If there is an infinite path $s = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ *in* M, *where all* a_i *are invisible with respect to* AP' *and* $B(s_i, t)$ *for* $i \geq 0$, *then there exists a transition* $s' \xrightarrow{b} t'$ *such that* b *is invisible, and for some* $j > 0$, $B(s_j, t')$

B *is a* visible bisimulation relation *if and only if both* B *and its inverse relation* B^{-1} *are visible simulations.* M *and* M' *are visibly-bisimilar if and only if there is a visible bisimulation relation* B *such that*

(3) $\forall i \in I \cdot \exists i' \in I' \cdot (i, i') \in B$, and

(4) $\forall i' \in I' \cdot \exists i \in I \cdot (i, i') \in B$.

The transition system in Figure 2.3 (a) and 2.3 (b) are visibly-bisimilar. To see this, we construct the relation which puts together states of Figure 2.3 (a) and states of Figure 2.3 (b) that are linked together by a dashed line. The actions a and b can be executed in any order leading to the same result, from the standpoint of verification. Figure 2.3 (a) and Figure 2.3 (c) are not visibly-bisimilar, the sate `12` in Figure 2.3 (c) does not correspond to any states in Figure 2.3 (a). Moreover, state `14` is not visibly-bisimilar to state `3` because they do not have the same labeling.

2.3.4 Bisimulation Equivalence

Bisimulation equivalence is the classical notion [Mil89], here adapted to transition systems by requiring identical state labelings, which ensures

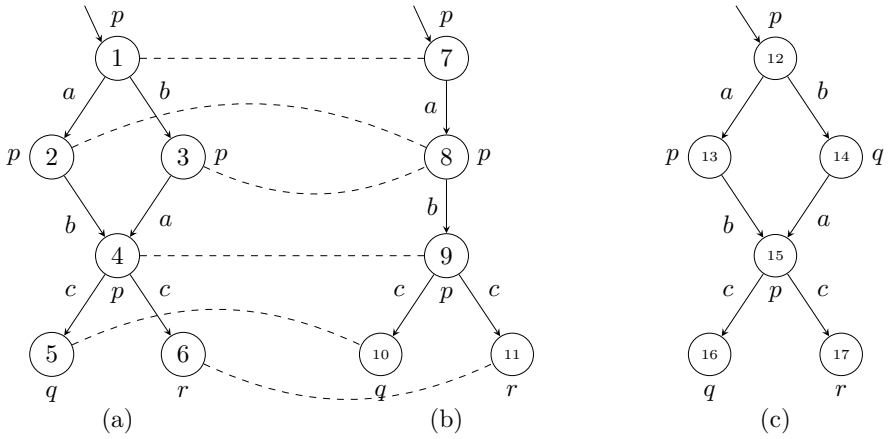


Figure 2.3: Visibly-bisimilar and not visibly-bisimilar structures

that CTL^* properties are preserved [CGP99]. Intuitively, bisimulation equivalence groups states that are impossible to distinguish, in the sense that both have the same labeling and offer the same transitions leading to equivalent states.

Definition 2.10 (Bisimulation Equivalence)**.** *Let* $M = (A, AP, S, R, I, L)$ *and* $M = (A, AP, S', R', I', L')$ *be two transition systems. A relation* $B \subseteq S \times S'$ *is a bisimulation equivalence relation between* M *and* M' *if and only if for all* $s \in S$ *and all* $s' \in S'$*:*

(1) if $(s, s') \in B$*, then* $L(s) = L'(s')$*, and*

(2) if $(s, s') \in B$ *and* $(s, a, t) \in R$ *then there is a state* $t' \in S'$ *such that* $(s', t') \in B$ *and* $(s', a, t') \in R'$*.*

(3) if $(s, s') \in B$ *and* $(s', a, t') \in R'$ *then there is a state* $t \in S$ *such that* $(s, t) \in B$ *and* $(s, a, t) \in R'$*.*

M *and* M' *are* bisimulation equivalent *if and only if there exists a bisimulation equivalence relation* B *such that the two following conditions hold:*

(3) $\forall i \in I \cdot \exists i' \in I' \cdot (i, i') \in B$*, and*

(4) $\forall i' \in I' \cdot \exists i \in I \cdot (i, i') \in B$.

We notice that unwinding a transition system results in a bisimulation-equivalent structure. Given a transition system M, and its computation tree $CT(M)$, M and $CT(M)$ are bisimulation-equivalent [CGP99].

Figure 2.4 (a) and Figure 2.4 (b) are bisimulation-equivalent. For each dashed oval, we can group together every state of Figure 2.4 (b) to state of Figure 2.4 (a) (e.g. $B(1, 3)$). On the other hand, Figure 2.4 (a) and Figure 2.4 (c) are not bisimulation-equivalent because state 7 in Figure 2.4 (c) does not correspond to any states in Figure 2.4 (a).

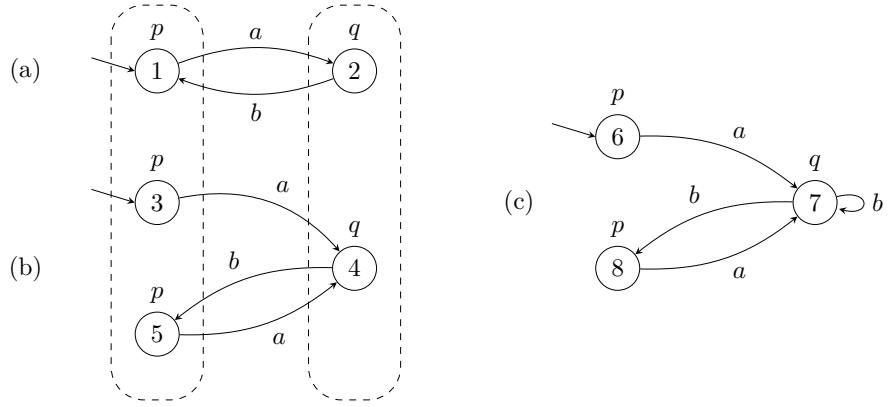


Figure 2.4: Bisimilar and nonbisimilar structures

2.4 Partial-Order Reduction

The goal of partial-order reduction methods (POR) is to reduce either the number of states or the number of interleavings which are explored by model-checking. They try to avoid the exploration of different equivalent interleavings of concurrent transitions [CGP99, GKPP99, God96, NG02, Val90, WW96]. Partial order reduction is based on the notions of *visibility* (c.f. Definition 2.4 and Definition 2.6) of actions and *independence* between actions. Two actions are *independent* if they do not disable one another and executing them in either order results in the same state.

Definition 2.11 (Independence of two actions)**.** *Given* $M = (A, AP, S, R, I, L)$*, a deterministic transition system, and two distinct actions*

$a, b \in A$. The actions a and b are independent if and only if for all $s \in S$ the two following conditions hold:

(1) if $a, b \in \text{enabled}(s)$ *then* $a \in \text{enabled}(R(s, b))$ *and* $b \in \text{enabled}(R(s, a))$*, and*

(2) if $a, b \in \text{enabled}(s)$ *then* $R(R(s, a), b) = R(R(s, b), a)$.

Figure 2.5 draws a transition system which contains two independent transitions a and b. Furthermore, transition a is invisible with respect to the formula $f = \mathsf{F}p$, while transition b is visible with respect to f because $L(s_0) \neq L(s_2)$.

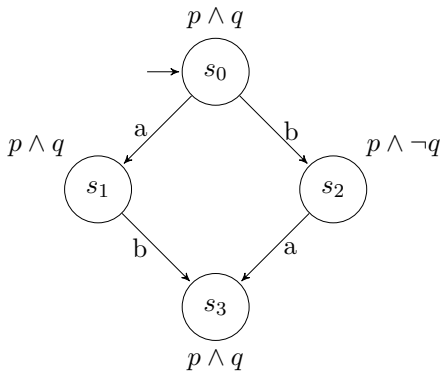


Figure 2.5: Two independent transitions

Intuitively, if two independent actions a and b are invisible with respect to the property f that one wants to verify, it does not matter whether a is executed before or after b. Actually, they lead to the same state and do not affect the truth of f. Partial order reduction consists in identifying such situations and restricting the exploration to either of these two alternatives. The following section develops the concept of partial order reduction in more detail.

2.4.1 A Modified Depth-First Search Algorithm

Given a deterministic transition system $M = (A, AP, S, R, I, L)$, POR amounts to exploring a reduced model $M_R = (A, AP, S_R, R_R, I, L)$, i.e M_R is a sub-transition system of M. In practice, most POR algorithms

[God96, CGP99] execute a modified depth-first search (DFS) algorithm (c.f. Algorithm 2.6). At each state s, it does not consider all the enabled actions in s, but instead it only considers a subset ample(s) of the enabled actions in s. This subset is called an *ample-set*.

To ensure that the verification results on the reduced model M_R hold for the full model M, ample(s) must respect some conditions. Those conditions depend on the type of properties that have to be preserved [CGP99, GKPP99]:

C_0 ample(s) $= \emptyset$ if and only if enabled(s) $= \emptyset$.

C_1 Along every path in the full state graph M that starts at s, an action $a \notin$ ample(s) that is not independent on an action in $a' \in$ ample(s) cannot occur without an action in $a'' \in$ ample(s) occurring first.

Conditions C_0, and C_1 are sufficient to guarantee that the reduced model contains a deadlock if and only if the original model contains a deadlock as well. In the literature, notably in [God96], when the set ample(s) respects the two conditions C_0 and C_1, it is called a *persistent set*.

Theorem 2.12 (c.f. [God96])**.** *Given a transition system M (and any property f), if $M_R = reduce(M, f)$ is a POR reduction of M using an* ample(s) *that satisfies the two conditions C_0 and C_1, then M contains a reachable state which is a deadlock if and only if M_R contains a state which is a deadlock.*

In order to check more elaborated properties than deadlocks the following conditions have been added:

C_2 If ample(s) $\neq$ enabled(s), then all actions in ample(s) are invisible.

C_3 A cycle in M_R is not allowed if it contains a state in which some action is enabled in M, but is never included in ample(s) on the cycle.

Conditions C_0, C_1, C_2 and C_3 are sufficient to guarantee that the reduced model preserves properties expressed in LTL_X, but does not preserve properties expressed in LTL, CTL, or CTL_X.

Algorithm 2.6: A Selective DFS

Header: Reduce(M)

Precondition: $\mathtt{M} = (\mathtt{A}, \mathtt{AP}, \mathtt{S}, \mathtt{R}, \mathtt{i}, \mathtt{L})$ is a deterministic transition system.

Result: $\mathtt{M}_\mathtt{R} = (\mathtt{A}, \mathtt{AP}, \mathtt{S}, \mathtt{R}_\mathtt{R}, \mathtt{i}, \mathtt{L})$ a reduced transition system.

Source code:

```
 1  global Visited ⊆ S := ∅
 2  global R_R ⊆ R := ∅
 3  global Stack := an empty stack of states
 4
 5  Reduce(M) {
 6    dfs(i);
 7    return (A, AP, Visited, R_R, i, L)
 8  }
 9
10  dfs(s) {
11    local Succ ⊆ S := ample(s)
12    push(Stack, s)
13
14    while(Succ ≠ ∅) {
15      local a ∈ A := an action of Succ
16      Succ := Succ \ {a}
17
18      R_R := R_R ∪ {s --a--> R(s,a)}
19
20      if(R(s,a) is not in Visited nor in Stack){
21        dfs(R(s,a))
22      }
23    }
24    pop(Stack, s)
25    Visited := Visited ∪ {s}
26  }
```

Theorem 2.13 (c.f. [CGP99, GKPP99])**.** *Given a transition system M, and a LTL_X property f, if $M_R = reduce(M, f)$ is a POR reduction of M using an* ample(s) *that satisfies conditions C_0–C_3, then $M \models f$ if and only if $M_R \models f$.*

An additional condition is necessary to ensure preservation of branching temporal logics.

C_4 If ample$(s) \neq$ enabled(s), then ample(s) is a singleton.

When conditions C_0–C_4 are fulfilled, [GKPP99] shows that there is a visible bisimulation (c.f. 2.3.3) between the complete and reduced models, which ensures preservation of CTL_X^* logics. By consequence, CTL_X properties are also preserved.

Theorem 2.14 (c.f. [GKPP99])**.** *Given a transition system M , a CTL_X^* property f, if $M_R = reduce(M, f)$ is a POR reduction of M using an* ample(s) *that satisfies conditions C_0–C_4, then $M \models f$ if and only if $M_R \models f$.*

Conditions C_1 and C_3 depend on the whole state graph M. C_1 is not directly exploitable in a verification algorithm. It is at least as hard to check condition C_1 as computing the reachable state space of the original transition system [CGP99]. Instead, one uses sufficient conditions, typically derived from the structure of the model description, to safely decide where reduction can be performed. For instance, in the Spin model checker context, it is derived from the Promela semantics that an assignment which only manipulates local variables and constant values defines a transition which satisfies condition C_1 [Hol97].

Condition C_1 is elaborated with the objective to obtain an ample set that is as small as possible. In [God96], it is shown that choosing the smallest ample(s) at each step is well-suited to partial order reduction. Often, it allows one to generate a reduced system which is substantially smaller.

Contrary to C_1, C_3 can be checked on the reduced graph, though in a nontrivial way. Hence, the following condition which is stronger than C_3 is often employed [CGP99, GKPP99]:

C_{3b} At least one state along each cycle of M_R is fully expanded.

We notice that a broad range of variations around those conditions have been studied [HGP92, ELLL04]. For instance, conditions C_0, C_1 and C_{3c}, which is defined below, are sufficient to preserve invariance of local properties. It is a subclass of reachability properties [ABH$^+$97, HGP92].

C_{3c} If a state s is not fully expanded, then at least one transition in ample(s) does not lead to already visited states.

2.4.2 Process Model

In this section we define a *process model*. It is an extension of a transition system (c.f. Section 2.1). A process model is a set of subsets of actions $\{A_0, A_1, \ldots, A_{m-1}\}$. Intuitively, we use the A_i's to construct the ample-sets which were defined in the previous section. More precisely, given a state s and a set of actions A_i, ample(s) will be equal to enabled(s) $\cap$ A_i. In our prototype tool Milestones presented in Chapter 6, we will define a transition system by means of a programming language which allows one to model one or more processes which communicate together. Given a system described with this language, we generate a process model, and we use this process model when we perform partial-order reduction.

Definition 2.15 (Process Model)**.** *Given a deterministic transition system $M = (A, AP, S, R, I, L)$, a* process model *for M consists of a finite set of sets of actions $\{A_0, A_1, \ldots, A_{m-1}\}$ with $A_i \subseteq A$. Besides, for each A_i, a transition relations R_i is defined as $R_i = R \cap (S \times A_i \times S)$.*

Definition 2.16 (Safe Process Model)**.** *A process model is* safe *with respect to M and a CTL* property f if and only if for all A_i and for all $a \in A_i$, a is invisible with respect to f, and for all $s \in S$,* ample(s) = enabled(R_i, s) *satisfies condition C_1. In such a case, we say that all $a \in A_i$ are* safe*.*

Definition 2.17 (Linear Process Model)**.** *A process model is* linear *with respect to M if and only if for all $s \in S$,* ample(s) = enabled(R_i, s) *is a singleton.*

In the sequel, when no confusion is possible, we write a transition system $M = (A, AP, S, R, I, L)$ with a safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$ to define a transition system M which is accompanied by a property f, a finite set of disjoint sets of actions $\{A_0, A_1, \ldots, A_{m-1}\}$,

and a finite set of transition relations $\{R_0, R_1, \ldots, R_{m-1}\}$ such that this process model is safe with respect to M and f. We proceed in the same way for a linear process model.

2.4.3 Forming-path

In this section, we present a theorem which will be intensively exploited in the sequel to perform partial-order reduction. Moreover, the modified DFS algorithm presented in Algorithm 2.6 which respects the conditions C_0, C_1, C_2, C_{3b}, and C_4 can be seen as a direct application of this theorem. Given a transition system M with a safe and linear process model $\{R_0, R_1, \ldots, R_{m-1}\}$ and a transition system M_R which is a sub-transition system of M, R. Gerth et al. states that if from each state of M_R starts a path $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ such that all the actions a_j belong to a subset A_i and $\text{ample}(s_n) = \text{enabled}(s_n)$, then M_R is visible-bisimilar to M, and so M_R preserves CTL^*_X properties [GKPP99]. Before presenting the theorem itself, we introduce some useful definitions.

A *forming-path* from s_0 to s_n is a path which is only composed of actions belonging to a process model.

Definition 2.18 (Forming Path)**.** *Let $M = (A, AP, S, R, I, L)$ be a transition system accompanied by a process model $\{R_0, \ldots, R_{m-1}\}$. Let s_0 and s_n be two states of S. The path $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ is a forming path if and only if for all a_i there exists a set $A_{j(i)}$ such that $a_i \in A_{j(i)}$.*

A state s is a fully-forming state when each path which starts from s begins with a forming path which leads to a completely expanded state.

Definition 2.19 (Fully-forming State)**.** *Let $M = (S, R, I, L)$ and $M_R = (S_R, R_R, I_R, L_R)$ be two transition systems such that $M_R \sqsubseteq M$. Let s be a state of S_R. The state s is a fully forming state with respect to R if and only if all paths which start from s begin by a prefix $\pi = s \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ such that π is a forming path and s_n is fully expanded with respect to R, i.e.* $\text{enabled}(R, \{s_n\}) = \text{enabled}(R_R, \{s_n\})$.

We are now able to express the main theorem which is presented in [GKPP99]. We point out that the theorem is paraphrased according to the notations and the definitions of this document.

Theorem 2.20 (c.f [GKPP99]). *Let $M = (S, R, I, L)$ and $M_R = (S_R, R_R, I_R, L_R)$ be two transition systems such that $M_R \sqsubseteq M$. Let $\{R_0, \ldots, R_{m-1}\}$ be a safe and linear process model of M. If each state of M_R is a fully forming state then M_R is visible-bisimilar to M.*

Theorem 2.21 (c.f [CGP99]). *Let $M = (S, R, I, L)$ and $M_R = (S_R, R_R, I_R, L_R)$ be two transition systems such that $M_R \sqsubseteq M$. Let $\{R_0, \ldots, R_{m-1}\}$ be a safe process model of M. M_R is stuttering equivalent to M if each state of M_R is a fully forming state,*

2.5 BDD-based Model Checking

2.5.1 Backward Symbolic Model-Checking of CTL

This section is devoted to the so-called backward symbolic model checking algorithm introduced in [BCM$^+$92] and applied in mainstream BDD-based model checkers as NuSMV [CCGR99]. The backward symbolic model checking algorithm supposes a transition system $M = (S, R, I, L)$, and takes as input a CTL formula f. It returns true or false depending on M verifies f or not, i.e. it checks whether $M \models f$. It starts by finding the set of all states which satisfy f, i.e. $\mathcal{L}(f)$. Then, it checks whether I is incluced to $\mathcal{L}(f)$. The algorithm is seen as backward search because the navigation from states to states is performed with the pre-image operation.

We present the eval algorithm which finds the set of all states which satisfy a CTL property f. It is a recursive algorithm which proceeds by induction on the structure of the formula. f. It uses three sub-problems: evalEX, evalEU, evalEG. They find the set of all the states witch satisfy f when f has the form $\mathsf{EG}\, g$, $\mathsf{E}\,[g\ \mathsf{U}\ h]$, or $\mathsf{EX}\, g$.

Those three algorithms are based on the following equivalences expressed in the fixed-point calculus. The notation $\mu Z.\tau(Z)$ and $\nu Z.\tau(Z)$ denotes the *least fixed point* and the *greater fixed point* of the *set transformer* τ. For more details, we refer the reader to [CGH97].

$$\mathcal{L}(\mathsf{EX}f) = \mathrm{pre}(\mathcal{L}(f)) \tag{2.1}$$

$$\mathcal{L}(\mathsf{E}[f\ \mathsf{U}\ g]) = \mu Z \cdot [(\mathcal{L}(f) \cap \mathrm{pre}(Z)) \cup \mathcal{L}(g)] \tag{2.2}$$

$$\mathcal{L}(\mathsf{EG}f) = \nu Z \cdot [\mathcal{L}(f) \cap \mathrm{pre}(Z)] \tag{2.3}$$

Here are the specifications of the sub-problems evalEX, evalEU, evalEG.

- evalEX takes as input a set of states F and returns the pre-image $\text{pre}(F)$.
- evalEU takes as input two sets of states F, and G and returns the least fixed point: $\mu Z \cdot [(\mathcal{L}(f) \cap \text{pre}(Z)) \cup \mathcal{L}(g)]$.
- evalEG takes as input a set of states f and returns the greater fixed point: $\nu Z \cdot [\mathcal{L}(f) \cap \text{pre}(Z)]$.

The recursive eval algorithm which finds the states satisfying f is presented in Algorithm 2.7.

2.5.2 Forward Symbolic Model-Checking of CTL

In [INH96], H. Iwashita et al. present a model-checking algorithm based on forward state traversal, which is shown to be more effective than backward state traversal in many situations. Forward traversal is applicable only to a subset of CTL, but can be combined with backward traversal for the rest of the formulae.

Given a CTL formula f and a set of initial sates I, backward BDD-based symbolic model-checking can be described as evaluating $\mathcal{L}(f)$ over the sub-formulas of f in a bottom-up manner, and checking whether $I \subseteq \mathcal{L}(f)$, or equivalently, whether $I \cap \mathcal{L}(\neg f)$ is empty.

H. Iwashita et al. introduce a forward exploration by transforming a "set emptiness problem" $H \cap \mathcal{L}(op_0(f)) = \emptyset$ into another one $H_{op_1} \cap \mathcal{L}(f) = \emptyset$. The former contains a "future" CTL operator op_0 in the right term. In the latter, op_0 is transformed into a kind of "past operator" op_1. It is used to compute a new left set of states H_{op_1}.

Given two sets of states H and F, the following equivalence over states are defined. Equation 2.4 (resp. Equation 2.5) is the backward (or past) version of Equation 2.2 (resp. Equation 2.3).

$$\text{FwdUntil}(H, F) = \mu Z.[H \cup \text{post}(Z \cap F)] \tag{2.4}$$

$$\text{EH}(F) = \nu Z.[F \cap \text{post}(Z)] \tag{2.5}$$

$$\text{FwdGlobal}(H, F) = \text{EH}(\text{FwdUntil}(H, F) \cap F) \tag{2.6}$$

Algorithm 2.7: The eval algorithm

Header: eval(f)

Global: A transition system $M = (S, R, I, L)$

Precondition: f is a (reduced as in Section 2.2.2) CTL formula.

Result: The set of states which satisfy f.

Induction Parameter: The size of f.

Source code: The implementation proceeds by induction on the structure of the formula f.

- If f is an element p of AP,
  $\text{eval}(f) = \mathcal{L}(p)$.
- If f has the form $\neg g$,
  $\text{eval}(f) = S \setminus \text{eval}(g)$.
- If f has the form $g \vee h$,
  $\text{eval}(f) = \text{eval}(g) \cup \text{eval}(h)$.
- If f has the form $\mathsf{EX}\, g$,
  $\text{eval}(f) = \text{evalEX}(\text{eval}(g))$.
- If f has the form $\mathsf{E}(g\, \mathsf{U}\, h)$,
  $\text{eval}(f) = \text{evalEU}(\text{eval}(g), \text{eval}(h))$.
- If f has the form $\mathsf{EG}\, g$,
  $\text{eval}(f) = \text{evalEG}(\text{eval}(g))$.

FwdUntil (H, F) computes the set of all states s which belong to a path having the following form: $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s$ where $s_0 \in H$ and $\forall i \in [1, \ldots, n[\cdot s_i \in F$. Intuitively, the FwdUntil function compute a kind of reversed EU operator. Given a transition M and its reverse transition system M^{-1}, if $M, s_i \models \mathsf{E}[f \,\mathsf{U}\, g]$ then $s_i \in \text{FwdUntil}\left(\mathcal{L}\left(g\right), \mathcal{L}\left(f\right)\right)$ when $\text{FwdUntil}(\mathcal{L}(g), \mathcal{L}(f))$ is computed on M^{-1}. The reverse is not totally true, because s_i might satisfy neither f nor g, but $M, s_i \models (\mathsf{E}\left[f \,\mathsf{U}\, g\right]) \vee \mathsf{EX}\,\mathsf{E}\left[f \,\mathsf{U}\, g\right]$.

EH (F) computes the set of all states s from which it is possible to go back indefinitely within F. It means that there exists from s an infinite "reverse path": $s = s_0 \xleftarrow{a_0} s_1 \xleftarrow{a_1} s_2 \xleftarrow{a_2} \ldots$ where $\forall i \in \mathbb{N} \cdot s_i \in F$. The EH function computes a kind of reversed EG operator. Actually, given a transition M, its reverse transition system M^{-1}, and a CTL property f, $(M, s_i) \models \mathsf{EG}\, f$ if and only if $s_i \in \text{EH}(\mathcal{L}(f))$ when $\text{EH}(\mathcal{L}(f))$ is computed on M^{-1}.

On this basis, the following equivalences are established:

$$H \cap \mathcal{L}(\mathsf{EX}\, f) = \emptyset \iff \text{post}(H) \cap \mathcal{L}(f) = \emptyset \tag{2.7}$$

$$H \cap \mathcal{L}(\mathsf{E}[f \,\mathsf{U}\, g]) = \emptyset \iff \text{FwdUntil}(H, \mathcal{L}(f)) \cap \mathcal{L}(g) = \emptyset \tag{2.8}$$

$$H \cap \mathcal{L}(\mathsf{EG}\, f) = \emptyset \iff \text{EH}\left(\text{FwdUntil}(H, \mathcal{L}(f)) \cap \mathcal{L}(f)\right) = \emptyset \tag{2.9}$$

Given a transition system M, and a CTL property f, the transformation process starts from $I \cap \mathcal{L}(\neg f) = \emptyset$. The equivalences above are applied recurrently until the right part cannot be reduced further, either because all temporal operators have been eliminated or because no rule applies to those remaining. Disjunctions in f can also be handled by case-splitting. Given the final $H \cap \mathcal{L}(f) = \emptyset$, H is computed using forward traversal, and the resulting set of states is used as the new set of initial sates for a classical, backward model-checking of the remaining f. For instance, it is impossible to translate the following formula $I \cap \mathcal{L}(\mathsf{AG}\,\mathsf{EF}\, p) = \emptyset$. Actually, the translation procedure finishes with the formula $\text{FwdUntil}(I, \mathcal{L}(\text{true})) \cap \mathcal{L}(\neg\mathsf{EF}\, p) = \emptyset$ because there remain no other rules to apply. $\text{FwdUntil}(I, \mathcal{L}(\text{true}))$ is computed in a forward way, while $\mathcal{L}(\neg\mathsf{EF}\, p)$ is computed is a backward way.

By using these equivalences, it is possible to replace an outermost EX, EU or EG operator in f with a forward traversal operator in I. For

instance, one can derive the following equivalence:

$$I \cap \mathcal{L}(\mathsf{AG}(req \rightarrow \mathsf{AF}\, ack)) = \emptyset \iff$$
$$\text{FwdGlobal}((\text{FwdUntil}(h_s, true) \wedge req), \neg ack) = \emptyset$$

2.5.3 Representing a Transition System

In classical BDD-based model checking, the sets of states and the transition relations which are manipulated by the different algorithms are encoded by means of *Ordered Binary Decision Diagrams* (BDDs). BDDs are a canonical form representation for propositional formulae [Bry86]. It is a rooted and directed acyclic graph (DAG) which consists of decision nodes and two terminal nodes called 0-terminal and 1-terminal. Each decision node is labeled by a Boolean variable and has two child nodes called low child and high child. The edge from a node to a low or high child represents an assignment of the variable to 0 or 1. Such a BDD is called *ordered* if different variables appear in the same order on all paths from the root. A BDD is said to be *reduced* if it respects the two following properties [CGP99] hold:

(1) the graph does not contain two isomorphic subgraphs, and

(2) the graph does not contain node whose children are isomorphic.

To explain how a finite transition system can be encoded into BDDs, we suppose a high level description of transition system $M = (A, AP, S, R, I, L)$ which is finite and so S, A, and AP are finite sets. Because S is finite, each state of S can be encoded as a binary number of length $m \approx \log_2(\#S)$, and so S can be encoded with a set S_b of m Boolean variables. In the same way, A can be encoded with a set A_b of Boolean variables. The set of proposition AP does not need to be encoded because it already contains Boolean variables. Finally, M itself can be translated into a new transition system $M_b = (A_b, AP, S_b, R_b, I_b, L_b)$ where R_b, I_b, and L_b refer to A_b and S_b instead of A and S. The BDDs are used to encode I_b, R_b, and L_b. The sets S_b, A_b and AP are not explicitly encoded into BDDs because the model checking algorithms do not use them.

To encode the transition relation R_b into a BDD, a copy S'_b of S_b is needed. S'_b of S_b are disjoint. The S_b is used to represent the source states. The set S'_b is used to represent the target states. A propositional

formula F_R over $(A_b \cup S_b \cup S'_b)$ is constructed. It is used to characterize the transition relation R. To understand the purpose of F_R, we suppose the existence of an action a, and two states s and t. Action a and the two states s and t can be seen as a valuation of respectively the variables of A_b, S_b and S'_b. Hence, F can be evaluated on the action a, and the two states s and t. The formula F_R is constructed in such a way that given an action a, and two states s and t, it is evaluated to true if and only if $s \xrightarrow{a} t$ belongs to R. Because F_R is a propositional formula, it is easily converted into BDDs.

To encode the initial set states I_b into a BDD, a propositional expression F_I over S_b is constructed. It is used to characterize the set of initial states I_b. The formula F_I is constructed in such a way that that given a states s, it is evaluated to true if and only if $s \in I$. Then, The propositional formula F_i is converted into a BDD representing I_b.

Concerning the labeling function, one BDD is created for each proposition $p \in AP$. It represents the set of states labelled with p, i.e. $\{s \in S \mid p \in L_b(s)\}$. Those BDDs are created by a similar mechanism as the set of initial states I_b.

There are efficient algorithms for manipulating the BDDs. For instance those algorithms allow one to compute:

- the intersection or the union of two sets of states,
- the complement of a set of states,
- the pre-image or the post-image of a set of states with respect to a relation on states.

We notice that even if BDD-based implementations have been shown very efficient [BCM+92], all the algorithms which perform backward and forward model checking remain correct with any other encoding for sets. For instance, the sets could be represented by enumeration, by a propositional formula, or by other kinds of decision diagrams such as Multi-valued Decision Diagrams [MD98]. For this reason, in the sequel, we use the name *set-based* model checking when we refer to algorithms which manipulate set of states instead of individual states.

2.5.4 Ordering of the Variables

As seen in the previous section, BDDs require a fixed ordering among the boolean variables used to represent the system. The size of BDDs, and therefore the performance of BDD-based model-checking, strongly depends on this ordering. For instance, the size of the BDD representing an n-bit comparator $(x_1 = x'_1 \wedge \cdots \wedge x_n = x'_n)$ can go from $3 * n + 2$ nodes with the order $x_1 \prec x'_1 \prec \cdots \prec x_n \prec x'_n$ to $3 * 2^n - 1$ nodes with the order $x_1 \prec \cdots \prec x_n \prec x'_1 \prec \cdots \prec x'_n$. In general, finding the best variable ordering is an NP-complete problem. The topic has been intensively studied and several heuristics have been developed for finding a good ordering between variables.

We now show two heuristics to order the variables used for representing the transitions relation R_b which is defined in the previous section. The BDD which represent R_b ranges over three sets of variables $A_b = a_0, a_1, \ldots, a_a$, $S_b = s_0, s_1, \ldots, s_s$ and $S'_b = s'_0, s'_1, \ldots, s'_s$. An intuitive approach would be to start with A_b, followed by S_b, then S_b. In the case of strongly asynchronous systems, this approach leads to an explosion of the BDD size [EFT93]. A better solution for asynchronous models is proposed in [EFT93]. The action variables are encoded first, followed by an "interlacing" between the source variables and the target variables: $a_0 \prec a_1 \prec \cdots \prec a_a \prec s_0 \prec s'_0 \prec s_1 \prec s'_1 \prec \cdots \prec s_s \prec s'_s$

Experimental results show that the resulting BDDs typically grow linearly in the number of asynchronous components. Intuitively, the ordering works well due to the fact that, in the case of asynchronous processes, most of the time a small number of processes proceed, so only the variables of those processes change while most variables remains the same (i.e. $s_i = s'_i$). These constraints are more efficiently encoded in the BDD if s_i and s'_i are next to each other in the ordering, similarly to the n-bit comparator example above.

Table 2.1 compares the transition relation BDD size and the time between the intuitive and the interlaced ordering, based on the turntable case study of Section 7.2. The size of the model is driven by the parameter #drill, and the time corresponds to verifying property $p6$. It confirms the much reduced growth rate of the interlaced ordering, allowing a much larger number of components to be added.

# drills	# vars	interlaced		non-interlaced	
		size	time	size	time
1	24	1 543	.041	153 056	6.222
2	31	1 913	.070	4 051 081	409.078
3	38	2 307	.114	—	—
20	157	12 184	4.436	—	—
40	297	31 572	30.884	—	—

Table 2.1: Size of the transition relation BDD (in # nodes) and verification time (in seconds) for property p6 of the Turntable case study, using interlaced vs. non-interlaced orderings, — correspond to memory exhausted (2 GB)

2.6 Conclusion

In this section, we have laid the groundwork of this thesis. In particular, we have presented:

- The transition systems which are used to model real-life systems.
- The CTL and LTL temporal logics which are used to express in a formal way what a real-life system is expected to do.
- The partial-order reduction technique which given a transition system M computes a reduced transition system M_R that preserves deadlocks, LTL_X properties, or CTL^*_X properties.
- The backward and the forward symbolic model checking algorithms which allow one to check whether a CTL property is verified by a transition system M.

In the next parts of this thesis, we will use those concepts to combine symbolic model checking with partial-order reduction approaches and verify systems featuring asynchronous processes.

Chapter 3

Checking CTL Properties: a Set-Based Approach

This chapter presents the *evalCTLX* algorithm. It combines partial-order reduction (POR) and set-based model checking, which manipulates sets of states rather than individual states (c.f. 2.5.3). Those sets are generally encoded by means of Binary Decision Diagrams (BDD) or Multi-valued Decision Diagrams (MDD). The evalCTLX algorithm is designed to verify properties on models featuring asynchronous processes. More precisely, it checks whether a transition system with a safe and linear process model (c.f. Section 2.4.2) verifies a CTL_X property or not.

At the root of our work is the *Two-Phase* algorithm of N. Nalumasu al. [NG02]. It is an explicit POR algorithm, i.e. it does not manipulate sets of states, but solely individual states. It takes as input a transition system M. It generates a reduced transition system M_R wich preserves CTL^*_X properties. Besides, it is implemented inside the *PV model checker* [NG97b, NG98b]. For some real protocols, it is able to generate a reduced state space much faster than other POR tools, e.g. Spin [Hol97]. From the Two-Phase algorithm, F. Lerda et al. developed the *ImProviso* algorithm [LST03]. Using BDD-based symbolic model checking, It computes the reachable state space of a reduced transition system which preserves reachability properties [BK08].

In this chapter, we start by presenting and discussing both the Two-Phase algorithm and the ImProviso algorithm. Then, we construct the *PartialExploration* algorithm which is a variation around ImProviso.

Moreover, it is a POR version of the *FwdUntil* algorithm of H. Iwashita et al. [INH96] presented in Section 2.5.2. We recall that thanks to the FwdUntil algorithm a fragment of CTL can be verified in a forward way. Intuitively, given two sets of states H and F, the PartialExploration algorithm computes all the states which are reachable from a state of H within states of F. On the one hand, the FwdUntil algorithm computes those states on the original transition system. On the other hand, the PartialExploration algorithm computes those states on an equivalent reduced version of the original transition system.

Contrary to classical POR algorithms, the PartialExploration algorithm does not need to identify cycles. Instead, it guarantees that from each state of the reduced graph, only sequences of safe actions will start (c.f. Definition 2.16). These sequences ends by a fully expanded state as presented in Section 2.4.3. In consequence, no action is postponed forever. In others words, each state is a fully-forming state (c.f. Section 2.4.3). The PartialExploration algorithm can be used to verify a fragment of CTL_X. Intuitively, it is able to verify CTL_X properties having the form $\mathsf{E}[f\ \mathsf{U}\ g]$ and $\mathsf{EG}\ f$.

The last part of this chapter is devoted to the construction of the evalCTLX algorithm. To verify all CTL_X properties, it combines forward and backward symbolic model checking. The former applies partial-order reduction based on the PartialExploration algorithm, while the latter is the classical CTL model checking algorithm which is presented in Section 2.5.1.

To summarize, the main contributions of this chapter are the PartialExploration algorithm that combines POR and forward CTL model checking, the evalCTLX algorithm which verifies CTL_X properties, a proof of their correctness, and an original way to treat cycles in the context of POR.

The remainder of this chapter is structured as follows. Section 3.1 reviews the Two-Phase algorithm for POR, and its symbolic incarnation in ImProviso. Section 3.2 presents the PartialExploration algorithm which revisits the ImProviso algorithm. Section 3.3 combines together the PartialExploration algorithm and the forward model checking approach of H. Iwashita et al. Section 3.4 presents the evalCTLX algorithm. Section 3.5 provides a conclusion and observations.

3.1 The Two-Phase Approach

This section starts by presenting the *Two-Phase* algorithm. It is an explicit model-checking algorithm which computes the reachable state space of a reduced transition system which is visible-bisimilar to the original one. Then, the *ImProviso* algorithm is introduced. It is a symbolic version of the Two-Phase algorithm. In the sequel we adapt it to allow the verification of CTL_X rather than limiting its use to verification of reachability properties only.

3.1.1 The Two-Phase Algorithm

The *Two-Phase* algorithm was conceived at the end of 1995 in the context of verifying real distributed shared-memory protocols. It was initially presented by R. Nalumasu et al. in a series of articles [Nal98, NG96, NG97a, NG97b, NG98a, NG98b, NG02]. It was firstly proved that the algorithm preserves stutter-free safety properties [NG97a]. Then, it was demonstrated that LTL_X properties are also preserved [Nal98, NG97b, NG98a]. Finally, it was proved that all CTL^*_X are preserved [NG02]. The algorithm was also implemented inside the PV model checker [NG97b, NG98b]. Experimental results suggest that the PV model checker is able to compute a safe part of the reachable state space of very large systems, where in some cases, other tools such as Spin are not able to make that computation [NG97b, NG98b].

The *Two-Phase* algorithm takes as input a transition system with a safe and linear process model. It generates, depending on the versions, either a reduced transition system or the reachable state space of a reduced transition system which preserves CTL^*_X properties.

It can been seen as a variant of the classical DFS algorithm with POR of [God96, CGP99]. In other words, it performs a selective DFS. At each step the choice of the ample set respects the conditions C_0, C_1, C_2, C_4 of Section 2.4.1. Intuitively, it tries as much as possible to expand successively each transition relation R_i of a safe and linear process model $\{R_0, \ldots, R_{m-1}\}$. When either a cycle is closed or no safe actions remain to be expanded for the current transition relation R_i, the algorithm switches to the next safe transition relation R_{i+1}. When all the transition relations have been expanded, a full expansion is made. More precisely, the Two-Phase algorithm alternates between two distinct

phases:

- *Phase-1* (c.f. procedure `Phase1OfAllProcesses` of Listing 3.1, line 26) considers each transition relation R_i at a time, in a fixed order. As long as actions remain to be expanded for the current transition relation R_i and no cycles are closed, the single transition that is enabled for that process is executed. Otherwise, either the algorithm moves on to the next transition relation R_{i+1}, or, when all the processes are expanded, the last reached state is passed on to Phase-2.

- *Phase-2* (c.f. procedure `Phase2` of Listing 3.1) is simple. It performs a full expansion of the states resulting from Phase-1, then a Phase-1 is applied again to the reached states.

Algorithm 3.1 and Listing 3.1 introduces the specification and the implementation of the Two-Phase algorithm. We firstly make precise the roles of the main variables. To that end, we suppose that the algorithm has already performed a number of cycles Phase-1/Phase-2, and is now performing a Phase-1. The `Visited` variable (line 4) contains all the states which have been expanded since the beginning of the algorithm to the the end of the last Phase-2. The `Queue` variable (line 5) contains all the states which have been expanded since the beginning of the current Phase-1. The `Frontier` variable (line 6) contains the states which remain to expand, and one of its states is chosen to start each Phase-1. The `current` variable (line 7) is the state under consideration. The $\texttt{R}_\texttt{R}$ variable (line 8) represents the transition relation of M_R.

Algorithm 3.1: The Two-Phase Algorithm [NG02]

Global: A transition system $M = (S, R, i, L)$ with a process model $\{R_0, R_1, \ldots, R_{m-1}\}$.

Header: Two-Phase()

Precondition: $M = (S, R, i, L)$ is a deterministic transition system which has an unique initial state, and $\{R_0, R_1, \ldots, R_{m-1}\}$ is a safe and linear process model.

Result: A reduced transition system $M_R = (S_R, R_R, i_R, L_R)$ which is visible-bisimilar to M.

Implementation: The source code is given in Listing 3.1

Listing 3.1: The Two-Phase Algorithm [NG02]

```
1  global M = (A, AP, S, R, i, L): a transition system
2  global R_0, R_1, ..., R_{m-1} ⊆ R
3
4  global Visited ⊆ S
5  global Queue ⊆ S
6  global Frontier ⊆ S
7  global current ∈ S
8  global R_R ⊆ R
9
10 TwoPhase() {
11   R_R := ∅
12   Visited := ∅
13   Frontier := {i}
14
15   while (Frontier ≠ ∅) {
16     current := an element of Frontier
17     Queue := {current}
18
19     Frontier := Frontier \ {current}
20     Phase1OfAllProcesses()
```

```
21      Phase2()
22    }
23    return (A, AP, Visited, R_R, i, L)
24  }
25
26  Phase1OfAllProcesses() {
27    local i ∈ ℕ := 0;
28    while(i ≠ m) {
29      Phase1OfOneProcess(i)
30      i++;
31    }
32  }
33
34  Phase1OfOneProcess(i) {
35    local continue ∈ {true, false} := post(R_i, {current}) ≠ ∅
36
37    while(continue) {
38      local a ∈ A :=
39        the unique action of enabled(R_i, {current})
40
41      if (R_i(current, a) ∉ Queue) {
42        R_R := R_R ∪ {current --a--> R_i(current, a)}
43        current := R_i(current, a)
44        Queue := Queue ∪ {current}
45        continue := post(R_i, {current}) ≠ ∅
46      } else {
47        continue := false
48      }
49    }
50  }
51
52
53  Phase2() {
54    local isNew ∈ {true, false} := current ∉ Visited
55    Visited := Visited ∪ Queue
56    if (isNew) {
57      local Image ⊆ S := post(R, {current})
58      Frontier := (Frontier ∪ Image) \ Visited
59      R_R := R_R ∪ {current --a--> R(current, a) | a ∈ enabled(R, {current})}
60    }
61  }
```

During Phase-1, the `Queue` variable exclusively contains states which belong to a path of the reduced transition system $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n$ where all actions a_i are invisible and all $\{a_i\}$ respect the condition C_1. The goal of the first phase is to extend the paths π as much as possible. We point out that π might contain some cycles: $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{j-1}} \boxed{s_j} \xrightarrow{a_j} \ldots \xrightarrow{a_{p-1}} \boxed{s_j} \xrightarrow{a_p} \ldots \xrightarrow{a_{n-1}} s_n$. When such a cycle is closed, the current Phase-1 switches to the next transition relation of the process model. Thus, π can contain at most m cycles. At the end of Phase-1, the last state of π is given to Phase-2 to be fully expanded. Hence, all the states of π are fully-forming states an are added to the `Visited` set. The algorithm ends when `Frontier` is empty, as such no states remain to be treated. When the algorithm ends, the reduced transition system contains only fully-forming states (c.f. Section 2.4.3). As a consequence, it is visible-bisimilar to the original one and preserves CTL^*_X properties. It is interesting to note that the condition C_3 (or one of its variant) of Section 2.4.1 might not be fulfilled because the reduced transition system can contain a cycle $\boxed{s_j} \xrightarrow{a_j} \ldots \xrightarrow{a_{p-1}} \boxed{s_j}$ in which some action a_k is enabled with respect to the full transition relation, but is never included in $\mathrm{ample}(s_\ell)$ for any $j \leq \ell \leq p$ on the cycle.

This paragraph is devoted to the worst-case complexity of the Two-Phase algorithm. For that purpose, we firstly analyze when Phase-1 moves on the next transition relation. This happens when the variable `current` represents a dead state with respect to the transition relation R_i, or when the variable `current` belongs to `Queue`. We stress that Phase-1 does not take into consideration the states which belong to `Visited`. For this reason, during each Phase-1, all the states of `Visited` could reappear into `Queue`. As a direct consequence, it might happen that the Two-Phase algorithm visits on the order of $\#S^2$ state. This is for instance the case when each cycle Phase-1/Phase-2 discovers only one new state, and when each Phase-1 revisits the states of `Queue`. Nevertheless, experimental results do not report such bad behaviors. Actually, R. Nalumasu et al. intentionally choose this approach because it seems the most performant to verify real distributed shared-memory protocols [NG96, NG98a]. In practice, it seems more expensive to check at each step whether a state has already been visited since the beginning of the algorithm than risking to expand some states more than once.

3.1.2 The ImProviso Algorithm

In [LST03], F. Lerda et al. propose ImProviso, a symbolic adaptation of the Two-Phase algorithm. It combines POR and BDD-based methods. On the one hand, the Two-Phase algorithm takes as input a transition system with a safe and linear process model. It produces a reduced transition system which is visible-bisimilar to the original one. On the other hand, ImProviso takes as input a transition system M with a process model $\{R_0, R_1, \ldots, R_{m-1}\}$ which is safe but not necessary linear. It aims to produce a reduced transition system which is stuttering-equivalent to the original one (c.f. Section 2.3.1). Algorithm 3.2 presents the specification of ImProviso. Its implementation is given in Listing 3.2. We point out that the original ImProviso algorithm was modified to keep track of states that have no transition with the current process (cf. Listing 3.2, lines 39 and 47). Those states are passed on to the next process. If this computation was not done, we could have missed some states during the BFS. Intuitively, we could have violated the condition C_0 of Section 2.4.1. The need for this computation was apparently not addressed in [LST03].

Classical set-based model checking algorithms use a single transition relation to carry out the required computation on the state space. The ImProviso method supposes $m + 1$ transition relations, where m is the number of safe transition relations of the process model. One is $R_g = R \setminus \bigcup_{i=0}^{m-1} R_i$ which is used to expand the non-safe actions during Phase-2, and the others are the safe transition relations R_i which are used in Phase-1.

On the one hand, the Two-Phase algorithm considers each R_i only once during the same Phase-1. When the last process is reached, the current state is passed to the Phase-2. However, that state could still be safely expanded by another safe transition relation. On the other hand, the heuristic chosen by ImProviso is to include an additional loop during Phase-1 (line 20), which guarantees that a state is passed on from Phase-1 to Phase-2 only if it is not possible to expand it with any safe transition relation. In addition, this heuristic guarantees that no transition belonging to a safe transition relation will be expanded during Phase-2. Thus, the transition relation R_g which is used in Phase-2 contains only the transitions which are not safe. That is why the transition relation R_g is used instead of the whole one. This heuristic

could also have been added to the Two-Phase algorithm.

Contrary to the DFS preferred by classical POR methods, the set-based method amount to a *breadth-first* search (BFS). The DFS algorithm allows us to easily distinguish between states which close a cycle and states which have already been treated. This is possible because, at each step, the DFS algorithm keeps information on the currently visited path. The BFS algorithm does not offer that possibility because it does not keep any data about the paths of the graph. It is thus harder to detect cycles within Phase-1 in the symbolic case. ImProviso adopts a pessimistic approach: at each step during Phase-1, it is assumed pessimistically that any previously expanded state that is reached again might close a cycle, although these occurrences might actually be on different paths. This over-approximation guarantees that all cycles are correctly identified, but possibly needlessly reduces the number of states where Phase-1 can be applied. This is the key justification for basing ImProviso on the Two-Phase algorithm, as this limits the need for cycle detection to each single execution of Phase-1, as opposed to the whole exploration for more traditional POR approaches.

Algorithm 3.2: ImProviso [LST03]

Global: A transition system $M = (S, R, I, L)$ with a process model $\{R_0, R_1, \ldots, R_{m-1}\}$. $\{D_0, D_1, \ldots, D_{m-1}\} \subseteq 2^S$, $R_g \subseteq R$.

Header: ImProviso()

Precondition: M is a deterministic transition system with a safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$. Each $D_i = \text{dead}(R_i, S)$ contains the dead states with respect to R_i. $R_g = R \setminus \bigcup_{i=0}^{m-1} R_i$.

Result: The reachable state space of a reduced transition system which is stuttering-equivalent to M.

Source code: The source code is given in Listing 3.2

Listing 3.2: The ImProviso Algorithm [LST03]

```
1  global M = (A, AP, S, R, I, L): a transition system
2  global R_0, R_1, ..., R_{m-1} ⊆ R
3  global D_0, D_1, ..., D_{m-1} ∈ S
4  global R_g ⊆ R
5  global Visited ⊆ S
6  global Queue ⊆ S
7  global Frontier ⊆ S
8
9  ImProviso() {
10   Visited := ∅
11   Queue := ∅
12   Frontier := I
13
14   while (Frontier ≠ ∅) {
15     Phase1()
16     Phase2()
17   }
18 }
19
20 Phase1() {
21    local Old ⊆ S := Frontier
22    Phase1OfAllProcesses()
23
24    while (Old ≠ Frontier) {
25      Old := Frontier
26      Phase1OfAllProcesses()
27    }
28 }
29
30 Phase1OfAllProcesses() {
31    local i ∈ ℕ := 0
32    while (i ≠ m) {
33      Phase1OfOneProcess(i)
34      i := i + 1
35    }
36 }
```

```
37
38 Phase1OfOneProcess(i) {
39   local E ⊆ S := D_i ∩ Frontier
40
41   while (Frontier ≠ ∅) {
42     Queue := Queue ∪ Frontier
43     Frontier := post(R_i, Frontier)
44
45     E := E ∪ (Queue ∩ Frontier)
46     Frontier := Frontier \ Queue
47     E := E ∪ (D_i ∩ Frontier)
48   }
49   Frontier := E
50 }
51
52
53 Phase2() {
54   Visited := Visited ∪ Queue ∪ Frontier
55   Queue := ∅
56   Frontier := post(R_g, Frontier) \ Visited
57 }
```

As in the Two-Phase algorithm, ImProviso might visit a state more than once, for the same reasons. Intuitively, in the worst case, each state in `Visited` can be expanded again during Phase-1, and so reappears into `Queue`. In other words, in the worst case the algorithm might perform around $\#S^2$ operations on sets. However, experimental results do not exhibit such bad behaviors. On the contrary, they show that substantial improvements can be achieved by applying that technique.

3.2 The PartialExploration Algorithm

In this section, the *PartialExploration* algorithm is defined. It applies principles of ImProviso and extends them to support model checking of CTL_X properties with partial-order reduction. We start by a deep analysis of the problem and its interesting properties. Those properties allow us to construct a correct version of the PartialExploration algorithm. Afterwards, we construct the algorithm itself.

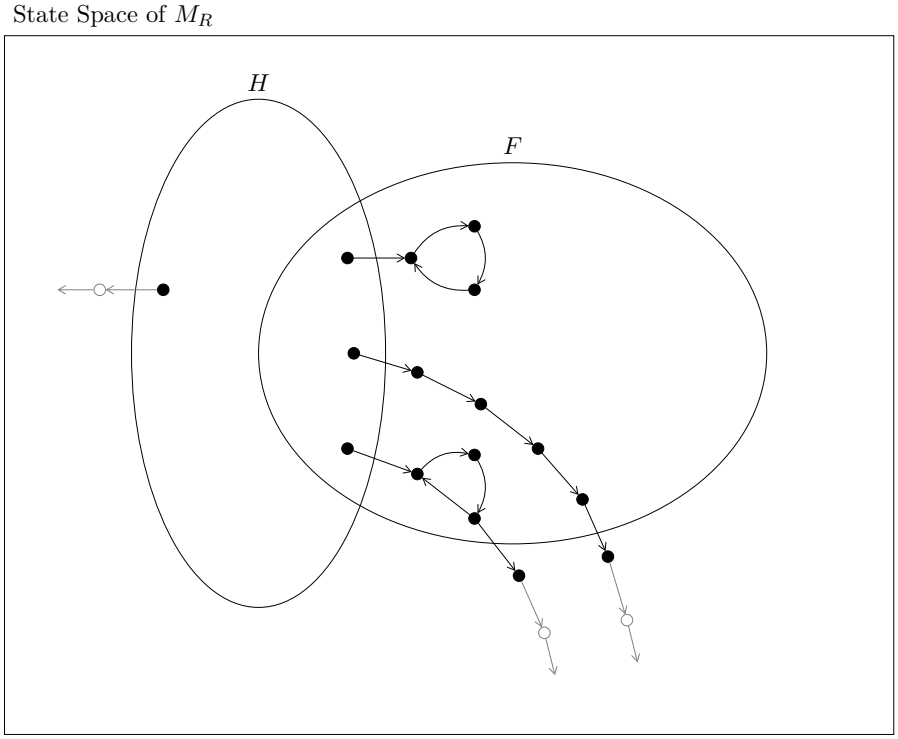


Figure 3.3: States which are Discovered by the PartialExploration Algorithm

3.2.1 Problem Theory

Given a transition system $M = (S, R, I, L)$ and a subset of H of S, the goal of the PartialExploration algorithm is to be an alternative to the the FwdUntil approach H. Iwashita et al [INH96] which is presented in Section 2.5.2. In the end, it will be used to answer the two following questions:

1. Is there a state in a set H which respects the CTL property $\mathsf{E}[f \,\mathsf{U}\, g]$?

2. Is there a state in a set H which respects the CTL property $\mathsf{EG}\, f$?

To that end, it takes as input a transition M and two set of states H and F, where F contains all states which respect f from the above formulae, i.e. $\mathcal{L}(f)$. It aims at discovering on a reduced transition system M_R, all the states which are reachable from a state of H through states of F. To rephrase, it looks for all the states s_n such that there exists a path of M_R: $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ where $s_0 \in H$ and $\forall i \in [0, \ldots, n-1] \cdot s_i \in F$. Figure 3.3 schematically represents M_R, H and F. The black states represent the states which will be found by the PartialExploration algorithms.

The key idea is to work with a transition system $M_H = (S, R, H, L)$ which is similar to $M = (S, R, I, L)$. However, its set of initials states is H instead of I. The PartialExploration algorithm performs the search not on the model M_H, but on a visible-bisimilar reduced transition model $M_R = (S, R_R, H, L)$. Note that only a part of the reduced transition system is constructed. We try to keep that part as small as possible.

To achieve its goal, the algorithm visits a reduced transition system M_R which contains only fully-forming states (c.f. Section 2.4.3). As such, we know that M_R is visible-bisimilar to M_H, and so it preserves CTL^*_X properties. At each step, the states which have already been visited are organized as follows. The `Visited` variable and the `Frontier` variable contain all the states which have already been reached. The `Frontier` variable contains the states which remain to be expanded. In addition, it contains at least one state s for each cycle which might not contain a state leading to a fully expanded state, s are both in `Visited` and in `Frontier`. The `Visited` variable contains all the states which have already been expanded.

F-arrangement

At each step, the algorithm arranges the states in such a way that it is possible to construct a transition system M_R which respects the two following conditions:

(1) M_R is visible-bisimilar to M_H.

(2) M_R contains at least all the states which have already been visited, that is, M_R contains at least all the states in $\texttt{Visited} \cup \texttt{Frontier}$.

More precisely, the `Visited` variable and the `Frontier` variable form an *F-arrangement* when it is possible to construct a transition system M_R which respects the three following conditions:

(1) M_R is visible-bisimilar to M_H (c.f. (1) of Definition 3.1).

(2) All initial states are in $\texttt{Visited} \cup \texttt{Frontier}$ (c.f. (2) of Definition 3.1)

(3) The successors of the states which are in $\texttt{Visited} \cap \texttt{F}$ but not in `Frontier` are necessarily in $\texttt{Visited} \cup \texttt{Frontier}$ (c.f. (3) of Definition 3.1).

(4) All states in $\texttt{Visited} \cup \texttt{Frontier}$ are reachable from H through states of F (c.f. (4) of Definition 3.1).

Definition 3.1 (F-arrangement)**.** *Let $M = (S, R, I, L)$ be a deterministic transition system with a safe and linear process model $\{R_0, R_1, \ldots, R_{m-1}\}$. Let H, F, `Visited`, and `Frontier` be four subsets of S. Let $M_H = (S, R, H, L)$ be a deterministic transition system derived from M and H.*

$(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement if and only if there exists a transition system $M_R = (S, R_R, H, L)$ such that the following conditions hold:

(1) M_R contains only fully-forming states with respect to M_H and $\{R_0, R_1, \ldots, R_{m-1}\}$, so it is visible-bisimilar to M_H, and

(2) $H \subseteq (\texttt{Visited} \cup \texttt{Frontier}) \subseteq S$, and

(3) $\mathrm{post}(R_R, F \cap \texttt{Visited} \cap \overline{\texttt{Frontier}}) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$.

(4) For all states $s \in (\texttt{Visited} \cup \texttt{Frontier})$ *there exists a path of* M_R*:* $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s$ *where* $s_0 \in H$ *and* $\forall i \in [0, \ldots, n-1] \cdot s_i \in F \wedge s_i \in (\texttt{Visited} \cup \texttt{Frontier})$.

According to the previous definition, when $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, we know that there exists at least one reduced transition system M_R which respects conditions (1), (2), and (3). In the sequel, any transitions systems M_R which respects conditions (1), (2), and (3) is referred to as a transition system induced by $(\texttt{Visited}, \texttt{Frontier})$. We notice that for any set of states H and F, and any transition system M, the two sets $\texttt{Visited} = \emptyset$ and $\texttt{Frontier} = H$ forms an F-arrangement because M_H is the required reduced transition system.

Performing a Partial Expansion

In this paragraph, given an F-arrangement $(\texttt{Visited}_0, \texttt{Frontier}_0)$ and a safe and linear transition relation R_i, we show how to perform a partial expansion of $\texttt{Frontier}_0$. Actually, any state $s \in \texttt{Frontier}_0$ which has a transition $s \longrightarrow_{R_i} s'$ can be safely expanded, and added to $\texttt{Visited}_0$. Moreover, s' has to be added to $\texttt{Frontier}_0$. To show that this construction is correct, we build from the F-arrangement $(\texttt{Visited}_0, \texttt{Frontier}_0)$ a new one: $(\texttt{Visited}, \texttt{Frontier})$.

Theorem 3.2. *Let* $M = (S, R, I, L)$ *be a deterministic transition system with a safe and linear process model* $PM = \{R_0, R_1, \ldots, R_{m-1}\}$*. Let* H*,* F*,* $\texttt{Visited}$*,* $\texttt{Visited}_0$*,* $\texttt{Frontier}$*, and* $\texttt{Frontier}_0$ *be six subsets of* S*. Let* $M_H = (S, R, H, L)$*. Let* $\texttt{E} \subseteq \texttt{Frontier}$ *be a set of states such that all* $s \in \texttt{E}$ *are not dead states with respect to* R_i*.*

If $(\texttt{Visited}_0, \texttt{Frontier}_0)$ *is an F-arrangement and the two following equalities hold then* $(\texttt{Visited}, \texttt{Frontier})$ *is an F-arrangement.*

(1) $\texttt{Visited} = (\texttt{Visited}_0 \cup \texttt{E})$

(2) $\texttt{Frontier} = (\texttt{Frontier}_0 \setminus \texttt{E}) \cup \text{post}(R_i, \texttt{E} \cap F)$

Proof. To show that $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, we construct a reduced transition $M_R = (S, R_R, H, L)$ which satisfies the three conditions of Definition 3.1.

Because $(\texttt{Visited}_0, \texttt{Frontier}_0)$ is an F-arrangement, we know that there exists a reduced transition system $M_0 = (S, R_0, H, L)$ which satisfies the conditions of Definition 3.1.

We now construct R_R as follows: $R_R = ((R_0 \setminus R_1) \cup R_2 \cup R_3)$ where:

- $R_1 = (E \cap F) \times \overline{(\texttt{Visited}_0 \cup \texttt{Frontier}_0)}$. We remove from R_0 all the transitions starting from a state $s \in E \cap F$ and leading to a state s' which is neither in $\texttt{Visited}_0$ nor in $\texttt{Frontier}_0$
- $R_2 = R_i \cap ((E \cap F) \times S)$. We add to R_0 all the transitions of R_i starting from a state $s \in E \cap F$.
- $R_3 = R \cap ((\overline{F} \cup \overline{\texttt{Visited}} \cup \texttt{Frontier}) \times S)$. We add to R_0 the transitions of R_3 to fully expand all the states s which do not belong to F or $\texttt{Visited}$, or which belong to $\texttt{Frontier}$.

Then, we show that the conditions of Definition 3.1 are met.

1. To prove the condition (1), we show that each state of M_R is a fully-forming state with respect to M_H and PM (c.f.Section 2.4.3). R_3 fully expands all the states s such that $s \in (\overline{F} \cup \overline{\texttt{Visited}} \cup \texttt{Frontier})$. All of those states are trivially fully-forming states. We now prove that the other states are also fully-forming states. We suppose a state s such that $s \in (F \cap \texttt{Visited} \cap \overline{\texttt{Frontier}})$. Hence $s \in \texttt{Visited}_0$. Since M_0 contains only fully-forming states with respect to M_H and PM, by expanding R_0, any path π: $s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ is a forming path such that s_n is fully expanded with respect to R. There are two cases:
   (a) The path π does not contain any state of $E \cap F$, and so π does not contain any transitions of R_1. Because R_R contains all the transitions of R_0 except those of R_1, all the transitions of π are also in R_R. Hence, π is also a forming path in M_R where s_n is fully expanded with respect to R. Therefore, the state s of M_R is a fully-forming state.
   (b) The path π contains a state of $(E \cap F)$. Suppose that e is the first state of π which belongs to $(E \cap F)$. By hypothesis, there is one transition $e \xrightarrow{a}_{R_i} e'$, and so there is a transition $e \xrightarrow{a}_{R_2} e'$ where $e' \in \texttt{Frontier}$. Thus, R_3 fully expands e'.

Moreover, π can be rewritten as follows $\pi = s \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{r-1}} e \xrightarrow{a_r} \ldots \xrightarrow{a_{n-1}} s_n$. Based on that, by expanding R_R, $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{r-1}} e \xrightarrow{a} e'$ is a forming path which ends by a full expansion. Therefore, the state s of M_R is a fully-forming state.

2. $H \subseteq (\texttt{Visited} \cup \texttt{Frontier}) = [(\texttt{Visited} \cup E) \cup (\texttt{Frontier} \setminus E)]$ $\subseteq (\texttt{Visited} \cup \texttt{Frontier}) \subseteq S$. Therefore, the condition (2) is verified.

3. To prove condition (3), we suppose a state $s \in (F \cap \texttt{Visited} \cap \overline{\texttt{Frontier}})$ such that $s \longrightarrow_{R_R} s'$. By hypothesis, $s \in (\texttt{Visited}_0 \cup E)$. There are two cases:

   (a) $s \in \texttt{Visited}_0 \wedge s \notin E$. In this case, $s \longrightarrow_{R_0} s'$ (c.f.definition of R_R). We recall that $(\texttt{Visited}_0, \texttt{Frontier}_0)$ is an F-arrangement. By condition (3), $s' \in (\texttt{Visited}_0 \cup \texttt{Frontier}_0) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$.

   (b) $s \in E$. The transition $s \xrightarrow{a_r}_{R_R} s'$ must belong to $(R_0 \setminus R_1)$ or to R_2 because it does not belong to R_3. There are two cases:

      i. $s \longrightarrow s' \in (R_0 \setminus R_1)$. By hypothesis, $s \in (E \cap F)$. Thus, $s' \in (\texttt{Visited}_0 \cup \texttt{Frontier}_0) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$ (c.f. definition of R_1).

      ii. $s \longrightarrow s' \in R_2$. Therefore, $s \longrightarrow_{R_R} s$ must belong to R_i, and $v' \in \text{post}(R_i, \texttt{E} \cap F)$. Consequently, $v' \in \texttt{Frontier}$

4. The proof of condition (4) proceeds in a similar way as the one of condition (1) and is left to the reader. □

Three Variants

The set E which is defined in the previous section can contain any state which is not a dead state with respect to R_i. Moreover, E can contain any number of such states. In other words, there exists more than one valid set E. For instance, the three following strategies can be easily implemented:

(1) $E = \texttt{Frontier} \setminus \text{dead}(\texttt{R}_\texttt{i}, \texttt{S})$ contains all states which are not dead states with respect to R_i. This corresponds to partially expanding all the states which allow such an expansion.

(2) $E = \texttt{Frontier} \setminus \texttt{Visited}$ contains all states which are not in `Visited`. This corresponds to fully expanding all the states which have been seen more than once during the search. Actually, this strategy is equivalent to the pessimistic one which considers that all such states close a cycle.

(3) $E = \texttt{Frontier} \setminus \texttt{FullyExpanded}$ where `FullyExpanded` is the set which contains all states which have already been fully expanded. It seems a good strategy not to partially expand the states of $(\texttt{FullyExpanded} \cap \texttt{Frontier})$, as those states have already been fully expanded. Even if states in $(\texttt{FullyExpanded} \cap \texttt{Frontier})$ will be fully expanded again during the next full expansion of Phase-2, all their successors will already be in `Visited` and thus not be explored again.

Performing a full expansion

In this paragraph, given an F-arrangement $(\texttt{Visited}_0, \texttt{Frontier}_0)$, we show how to perform a full expansion of $\texttt{Frontier}_0$. Actually, we merge $\texttt{Visited}_0$ and $\texttt{Frontier}_0$ to create the new `Visited`. The successors of $\texttt{Frontier}_0$ with respect of the full transition relation R constitute the new `Frontier`.

Theorem 3.3. *Let $M = (S, R, I, L)$ be a deterministic transition system with a safe and linear process model $PM = \{R_0, R_1, \ldots, R_{m-1}\}$. Let H, F, $\texttt{Visited}$, $\texttt{Visited}_0$, $\texttt{Frontier}$, and $\texttt{Frontier}_0$ be six subsets of S. Let $M_H = (S, R, H, L)$.*

If $(\texttt{Visited}_0, \texttt{Frontier}_0)$ is an F-arrangement and the two following equalities hold then $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement.

(1) $\texttt{Visited} = (\texttt{Visited}_0 \cup \texttt{Frontier}_0)$

(2) $\texttt{Frontier} = \text{post}(R, \texttt{Frontier}_0) \setminus \texttt{Visited}$

Proof. This proof is very similar to the proof of Theorem 3.2. It consists in constructing a reduced transition system M_R from a transition sys-

tem M_0 which is induced by the F-arrangement $(\texttt{Visited}_0, \texttt{Frontier}_0)$. Therefore, the complete proof is left to the reader. □

Properties of an F-arrangement

In this paragraph we present two useful properties of an F-arrangement. The first property will be used to show that at the end of the algorithm, an F-arrangement contains enough states to check a EU property. The second property states the same but for an EG property.

Theorem 3.4. *Let $M = (S, R, I, L)$ be a deterministic transition system with a safe and linear process model $PM = \{R_0, R_1, \ldots, R_{m-1}\}$. Let H, F, $\texttt{Visited}$, be three subsets of S. Let $M_H = (S, R, H, L)$. Let $(\texttt{Visited}_0, \emptyset)$ be an F-arrangement, and $M_R = (S, R_R, H, L)$ be any transition system induced by $(\texttt{Visited}, \emptyset)$.*

There exists a state $v \in S$ such that there is a path of M_R: $v_0 \xrightarrow{a_0} v_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-2}} v_{n-1} \xrightarrow{a_{n-1}} v$ such that $v_0 \in H$, and $\forall i \in [1, \ldots, n[\cdot v_i \in F$ if and only if $v \in \texttt{Visited}$.

Proof.

⇒ The proof proceeds by induction of the length n of π. The induction hypothesis is the theorem itself.

Base case When $n = 0$, by the theorem statement, $v = v_0$, and so v belongs to H. Because $H \subseteq (\texttt{Visited} \cup \emptyset) = \texttt{Visited}$, we deduce that v belongs to $\texttt{Visited}$ (c.f. condition (2) of Definition 3.1).

Inductive step We prove that when the theorem is valid for any $n \in \mathbb{N}$, it also holds for $n + 1$. We suppose a path $\pi = v_0 \xrightarrow{a_0} v_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} v_n \xrightarrow{a_n} v$ such that $v_0 \in H$, and $\forall i \in [1, \ldots, n] \cdot s_i \in F$. In this case, the induction hypothesis is applicable for v_n, and so, v_n belongs to $\texttt{Visited}$. By condition (3) of Definition 3.1, and because $v_n \in F$, we deduce that v also belongs to $\texttt{Visited}$.

⇐ The proof is trivial by condition (4) of Definition 3.1 □

The following corollary makes the link between an F-arrangement and a CTL property $\mathsf{E}\,[f\ \mathsf{U}\ g]$.

Corollary 3.5. *Let $M = (S, R, I, L)$ be a deterministic transition system with a safe and linear process model $PM = \{R_0, R_1, \ldots, R_{m-1}\}$. Let H be a subset of S. Let $M_H = (S, R, H, L)$. Let f and g be two CTL_X properties. Let F be the set of states which satisfy f, i.e. $F = \mathcal{L}(f)$. Let $(\texttt{Visited}, \emptyset)$ be an F-arrangement.*

There is a state $h \in H$ such that $M, h \models \mathsf{E}[f \,\mathsf{U}\, g]$ if and only if there is a state $v \in \texttt{Visited}$ such that $M, v \models g$.

Proof.

(1) There is a state $h \in H$ such that $M, h \models \mathsf{E}[f \,\mathsf{U}\, g]$.

(2) From the CTL semantics, (1) if and only if $M_H \not\models \neg\mathsf{E}[f \,\mathsf{U}\, g]$.

(3) M_R preserves CTL properties, so (2) if and only if $M_R \not\models \neg\mathsf{E}[f \,\mathsf{U}\, g]$.

(4) From the CTL semantics, (3) if and only if there exists a path of M_R: $v_0 \xrightarrow{a_0} v_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} v$ such that $v_0 \in H$, and $\forall i \in [1, \ldots, n[\cdot v_i \models f \wedge v_i \in F$, and $M, v \models g$.

(5) By Theorem 3.4, (4) if and only if $v \in \texttt{Visited}$. □

The following property is helpful to show that an F-arrangement can be exploited to verify a CTL property $\mathsf{EG}\, f$.

Theorem 3.6. *Let $M = (S, R, I, L)$ be a deterministic transition system with a safe and linear process model $PM = \{R_0, R_1, \ldots, R_{m-1}\}$. Let H, F, $\texttt{Visited}$, be three subsets of S. Let $M_H = (S, R, H, L)$. Let $(\texttt{Visited}_0, \emptyset)$ be an F-arrangement, and $M_R = (S, R_R, H, L)$ be a transition system induced by $(\texttt{Visited}, \emptyset)$.*

For all infinite path of M_R: $v_0 \xrightarrow{a_0} v_1 \xrightarrow{a_1} \ldots$ such that $v_0 \in H$, and $\forall i \in \mathbb{N} \cdot v_i \in F$, $\forall i \in \mathbb{N} \cdot v_i \in \texttt{Visited}$.

Proof. We prove the theorem by contradiction. We suppose an infinite path of M_R: $v_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} v_n \xrightarrow{a_n} v_{n+1} \ldots$ such that $\forall i \in \mathbb{N} \cdot v_i \in F$ and $\forall i \in [0, \ldots, n] \cdot v_i \in \texttt{Visited}$, and $v_{n+1} \notin \texttt{Visited}$. Thus, $v_n \in (\texttt{Visited} \setminus \emptyset) \cap F$. By condition (3) of Definition 3.1, $v_{n+1} \in \texttt{Visited}$. We obtain a contradiction. Therefore, the unique hypothesis $v_{n+1} \notin \texttt{Visited}$ must be false. □

The following corollary makes the link between an F-arrangement and a CTL property $\mathsf{EG}\, f$.

Corollary 3.7. *Let $M = (S, R, I, L)$ be a deterministic transition system with a safe and linear process model $PM = \{R_0, R_1, \ldots, R_{m-1}\}$. Let H be a subsets of S. Let $M_H = (S, R, H, L)$. Let f be a CTL_X properties. Let F be the set of states which satisfy f, i.e. $F = \mathcal{L}(f)$. Let $(\texttt{Visited}, \emptyset)$ be an F-arrangement.*

There is a state $h \in H$ such that $M, h \models \mathsf{EG}\, f$, if and only if there is a path in M: $v_0 \xrightarrow{a_0} v_1 \xrightarrow{a_1} \ldots$ such that $v_0 \in H$, and $\forall i \in \mathbb{N} \cdot v_i \in F \cap \texttt{Visited}$.

Proof. This proof is very similar to the proof of Corollary 3.5. It is left to the reader. □

3.2.2 Construction of the Algorithm

In this section, we construct the PartialExploration algorithm. This construction is based on the problem theory which was presented in the previous section. It supposes a deterministic transition system $M = (S, R, I, L)$ with a safe and linear process model $\{R_0, R_1, \ldots, R_{m-1}\}$, and two sets $H \subseteq S$ and $F \subseteq S$. It returns a relevant part of the reachable state space which corresponds to the set `Visited` such that $(\texttt{Visited}, \emptyset)$ forms an F-arrangement. In the following sections, it will be shown that the PartialExploration algorithm can be used instead of he FwdUntil algorithm (c.f. Section 2.5.2) to perform forward CTL_X model checking. The algorithm consists in sub-problems. We firstly define the environment of the algorithm. Then, we provide the specification of all the sub-problems. Concerning the specification of those sub-problems, when $\texttt{Visited}_0$ and $\texttt{Frontier}_0$ appear in a postcondition of a sub-problem, they respectively correspond to the sets `Visited` and `Frontier` before the call of that sub-problem. Figure 3.4 graphically represents the call tree of the PartialExploration Algorithm.

Environment

The global variables of the PartialExploration algorithm contains a finite and deterministic process model $\texttt{M} = (\texttt{S}, \texttt{R}, \texttt{I}, \texttt{L})$ with a safe and linear process model $\{\texttt{R}_0, \texttt{R}_1, \ldots, \texttt{R}_{\texttt{m}-1}\}$. For Each $\texttt{R}_\texttt{i}$, there is an associated

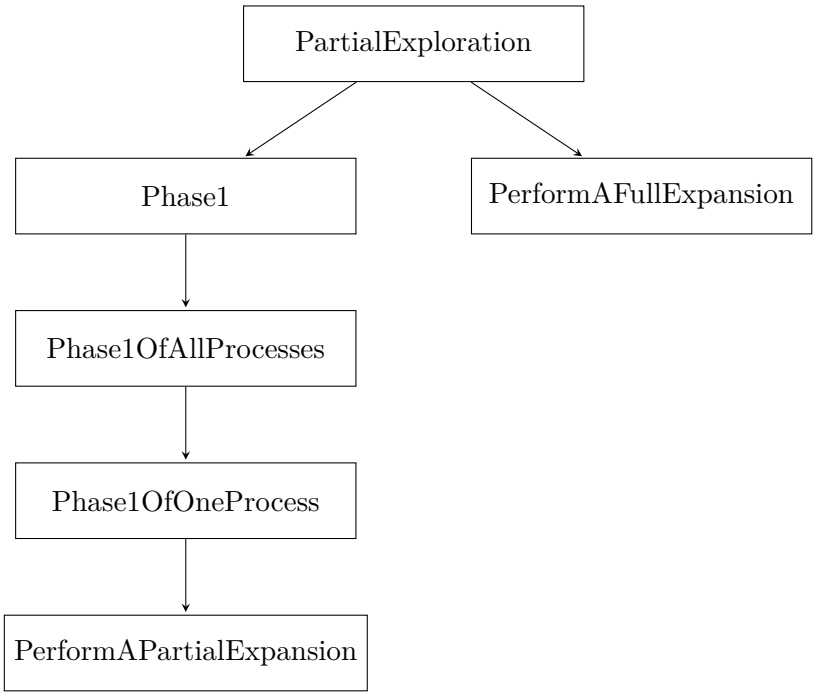


Figure 3.4: Call Tree of The PartialExploration Algorithm

set of states $\mathtt{D_i}$. It contains all the states which are dead states with respect to $\mathtt{R_i}$. The sets `Visited` and `Frontier` form an F-arrangement. Algorithm 3.5 precisely introduces that environment.

Algorithm 3.5: The global variables of the PartialExploration Algorithm

Global:

```
1 global M = (A, AP, S, R, I, L): a transition system
2 global R_0, R_1, ..., R_{m-1} ⊆ R
3 global D_0, D_1, ..., D_{m-1} ∈ S
4 global F ⊆ S
5
6 global Visited ⊆ S
7 global Frontier ⊆ S
```

Performing a Partial Expansion

Algorithm 3.6 shows how to concretely implement a one step of a partial expansion. It is guided by the Theorem 3.2. E contains all states which are not dead states with respect to R_i. We recall that other choices are possible (c.f. Section 3.2.1).

Algorithm 3.6: PerformAPartialExpansion

Header: PerformAPartialExpansion(i)

Precondition: $i \in \mathbb{N}$, and $0 \leq i < m$, and $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement.

Postcondition: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, and $\text{post}(\texttt{R}_\texttt{i}, \texttt{Frontier}_0 \cap \texttt{F}) \subseteq \texttt{Frontier}$, and $(\texttt{Visited}_0 \cup \texttt{Frontier}_0) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$.

Implementation:

```
1 PerformAPartialExpansion(i) {
2   local E := Frontier \ D_i
3   Visited := Visited ∪ E
4   Frontier:= (Frontier \ E) ∪ post(R_i, E ∩ F)
5 }
```

Performinging a Full Expansion

Algorithm 3.7 performs one full expansion. It is based on Theorem 3.3.

Algorithm 3.7: PerformAFullExpansion

Header: PerformAFullExpansion()

Precondition: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement.

Postcondition: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, and
$(\texttt{Visited} = \texttt{Visited}_0 \cup \texttt{Frontier}_0)$, and
$(\texttt{Visited} \cap \texttt{Frontier}) = \emptyset$.

Implementation:

```
1 PerformAFullExpansion() {
2   Visited := (Visited ∪ Frontier)
3   Frontier := post(R, Frontier ∩ F) \ Visited
4 }
```

Phase-1 of One Process

The Phase1OfOneProcess sub-problem expands transitions of a safe transition relation R_i. It is defined in Algorithm 3.8. It continues until no new states can be found. It is based on the specification of the PerformAPartialExpansion sub-problem.

Phase-1 of All Processes

The Phase1OfAllProcesses sub-problem expands the transitions of the various safe transition relations R_i. It is defined in Algorithm 3.9. It uses the Phase1OfOneProcess sub-problem.

Algorithm 3.8: Phase1OfOneProcess

Header: Phase1OfOneProcess(i)

Precondition: $i \in \mathbb{N}$, and $0 \leq i < m$, and $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement.

Postcondition: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, and $\text{post}^*(\texttt{R}_\texttt{i}, \texttt{Frontier}_0 \cap \texttt{F}) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$, and $(\texttt{Visited}_0 \cup \texttt{Frontier}_0) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$.

Loop Invariant: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, and $(\texttt{Visited}_0 \cup \texttt{Frontier}_0) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$. Moreover, there exists two sets $\texttt{Visited}_{-1} \subseteq \texttt{S}$ and $\texttt{Frontier}_{-1} \subseteq \texttt{S}$ which form an F-arrangement such that $\texttt{W}_\texttt{R} = (\texttt{Visited}_{-1} \cup \texttt{Frontier}_{-1}) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$.

Halting Condition: $\texttt{W}_\texttt{R} = (\texttt{Visited} \cup \texttt{Frontier})$

Loop Variant: $\#\texttt{S} - \#\texttt{W}_\texttt{R}$.

Implementation:

```
1  Phase1OfOneProcess(i) {
2    local W_R ⊆ S := (Visited ∪ Frontier)
3    PerformPartialExpansion(i);
4
5    while(W_R ≠ (Visited ∪ Frontier)) {
6      W_R := (Visited ∪ Frontier)
7      PerformPartialExpansion(i);
8    }
9  }
```

Algorithm 3.9: Phase1OffAllProcesses

Header: Phase1OffAllProcesses()

Precondition: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement.

Postcondition: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, and

$\text{post}^*(\texttt{R}_{\texttt{m}-1}, (\dots \text{post}^*(\texttt{R}_1, \text{post}^*(\texttt{R}_0, \texttt{Frontier}_0 \cap \texttt{F}) \cap \texttt{F}) \dots) \cap \texttt{F}) \subseteq$
$(\texttt{Visited} \cup \texttt{Frontier})$, and

$(\texttt{Visited}_0 \cup \texttt{Frontier}_0) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$.

Loop Invariant: $\texttt{0} \leq \texttt{i} < \texttt{m}$, and
$(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, and

$\text{post}^*(\texttt{R}_{\texttt{i}-1}, (\dots \text{post}^*(\texttt{R}_1, \text{post}^*(\texttt{R}_0, \texttt{Frontier}_0 \cap \texttt{F}) \cap \texttt{F}) \dots) \cap \texttt{F}) \subseteq$
$(\texttt{Visited} \cup \texttt{Frontier})$, and

$(\texttt{Visited}_0 \cup \texttt{Frontier}_0) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$.

Halting Condition: $i = m$

Loop Variant: $m - i$.

Implementation:

```
1 Phase1OfAllProcesses() {
2   local i ∈ ℕ := 0
3
4   while(i ≠ m) {
5     Phase1OfOneProcess(i);
6     i++
7   }
8 }
```

Phase-1

After the sequential expansion of the various safe transition relations, it might still be possible to discover new states with the help of those transition relations. For instance, a new state s which was discovered by a safe transition relation R_i can be a dead state with respect to R_i. But in the same time, s may well be expanded by an other safe transition relation R_j where $j < i$. The Phase1 sub-problem guarantees that a full expansion of a state is made only if it is not possible to expand it with any safe transition relations. It is defined by Algorithm 3.10. It is based on the specification of the Phase1OfAllProcesses sub-problem.

The PartialExploration Algorithm

We now construct the PartialExploration algorithm itself. It is defined by Algorithm 3.11. It is based on the specifications of the Phase1 sub-problem and the PerformAFullExpansion sub-problem.

We note that as the Two-Phase algorithm and the ImProviso algorithm, the PartialExploration algorithm might visit a state more than once. It happens during Phase-1 when a cycle is closed. In such a case, the algorithm might explore the same cycle of transitions more than one. At each step, `Frontier` contains at least a state s_c of the cycle. Our algorithm will stop looping around the cycle either during a Phase-2, or when a safe transition relation R_i allows to discover a state which is not on the cycle. When no POR are possible,the results of the PartialExploration algorithm and the ForwardUntil algorithm are the same.

This paragraph briefly analyzes the complexity of the PartialExploration algorithm in terms of set operations. One can see that in the worst case, the PartialExploration algorithm discovers only one new state during each Phase1/Phase2 cycle, except during the last one which does not discover any new state. When this happens, for each Phase1/Phase2 cycle, the algorithm performs on the order of m partial expansions plus one full expansion where m is the number of safe transition relations of the process model. Each partial or full expansion executes a constant number of set operations. Therefore, in the worst case, the PartialExploration algorithm performs on the order of $m \cdot \#S$ number of set operations.

Algorithm 3.10: Phase1

Header: Phase1()

Precondition: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement.

Postcondition: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, and $(\texttt{Visited}_0 \cup \texttt{Frontier}_0) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$, and $\text{post}(\bigcup_{i=0}^{m-1} R_i, \texttt{Frontier}) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$.

Loop Invariant: $(\texttt{Visited}, \texttt{Frontier})$ is an F-arrangement, and $(\texttt{Visited}_0, \texttt{Frontier}_0) \subseteq (\texttt{Visited}, \texttt{Frontier})$. Moreover, there exists two sets $\texttt{Visited}_{-1} \subseteq \texttt{S}$ and $\texttt{Frontier}_{-1} \subseteq \texttt{S}$ which form an F-arrangement such that $\texttt{W}_\texttt{R} = (\texttt{Visited}_{-1} \cup \texttt{Frontier}_{-1}) \subseteq (\texttt{Visited} \cup \texttt{Frontier})$.

Halting Condition: $\texttt{W}_\texttt{R} = (\texttt{Visited} \cup \texttt{Frontier})$

Loop Variant: $\#\texttt{S} - \#\texttt{W}_\texttt{R}$.

Implementation:

```
1  Phase1() {
2    local W_R ⊆ S := (Visited ∪ Frontier)
3    Phase1OfAllProcesses();
4
5    while(W_R ≠ (Visited ∪ Frontier)) {
6      W_R := (Visited ∪ Frontier)
7      Phase1OfAllProcesses();
8    }
9  }
```

Algorithm 3.11: PartialExploration

Header: PartialExploration($\mathtt{H}_0$, $\mathtt{Q}_0$)

Precondition: `M` is a deterministic transition system with a safe and linear process model $\{\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_{\mathtt{m}-1}\}$, and $\mathtt{F}_0 \subseteq \mathtt{S}$.

Postcondition: A set `Visited` such that $(Visited, \emptyset)$ is an F-arrangement.

Loop Invariant: (`Visited`, `Frontier`) is an F-arrangement.

Halting Condition: `Frontier` $= \emptyset$

Loop Variant: #`S` − #`Visited`

Implementation:

```
1  PartialExploration(H0, Q0) {
2    Q := Q0
3    Visited := ∅
4    Frontier := H0
5
6    while(Frontier ≠ ∅) {
7      Phase1();
8      PerformAFullExpansion();
9    }
10 }
```

Given a transition system M, it is easy to see that if the PartialExploration algorithm takes as input a safe but not linear process model, it induces a reduced transition system M_R which is stuttering-equivalent to M and so preserves LTL_X properties. This follows the fact that all states of M_R are fully-forming states (c.f. Theorem 2.21).

3.3 FwdUntil vs PartialExploration

In this section, we present the link between the FwdUntil algorithm and the PartialExploration algorithm of Section 3.2. Intuitively, the CTL properties $\mathsf{E}[f\ \mathsf{U}\ g]$ and $\mathsf{EG}\ f$ which can be verified with the FwdUntil approach can also be verified with the PartialExploration algorithm.

We start by a brief recall. In Section 2.5.2, we introduced:

- The FwdUntil(H, F) procedure which computes all the states which are reachable from a state of H through states of F.
- The EH(FG) procedure computes the states reachable from a cycle, all within FG

Moreover, the following equations have been presented (c.f. Equation (2.8) and (2.9)) .

$$H \cap \mathcal{L}(\mathsf{E}[q\ \mathsf{U}\ f]) = \emptyset \iff \text{FwdUntil}(H, \mathcal{L}(q)) \cap \mathcal{L}(f) = \emptyset \quad (3.1)$$

$$H \cap \mathcal{L}(\mathsf{EG}\ f) = \emptyset \iff \text{EH}\left(\text{FG}(H, \mathcal{L}(f))\right) = \emptyset \text{ where} \quad (3.2)$$

$$\text{FG}(H, F) = \text{FwdUntil}(H, F) \cap F$$

We now show how to use the PartialExploration algorithm instead of the FwdUntil algorithm. The idea is to replace the FwdUntil algorithm and keep EH as it is without partial-order reduction.

Theorem 3.8. *We suppose that $M = (S, R, I, L)$ is a deterministic transition system, $H \subseteq S$, and f, g are two CTL_X properties. There exists a state $h \in H$ which satisfies $\mathsf{E}\ [f\ \mathsf{U}\ g]$ (i.e. $M, h \models \mathsf{E}\ [f\ \mathsf{U}\ g]$) if and only if there exists a state v in the result set* `Visited` *of the PartialExploration$(H, \mathcal{L}(f))$ algorithm such that v satisfies the CTL_X property g (i.e. $M, v \models g$).*

Proof. The proof consists of the following four implications:

(1) $M, h \models \mathsf{E}[f \,\mathsf{U}\, g]$

(2) By definition, (1) if and only if there exists a path $h = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ in M, such that $\forall i \in [0, \ldots, n[\cdot s_i \models f$, and $s_n \models g$.

(3) By Corollary 3.5, (2) if and only if there exists a state $v \in \texttt{Visited}$ such that there exists a path $t_0 \xrightarrow{b_0} t_1 \xrightarrow{b_1} \ldots \xrightarrow{b_{m-1}} t_m = v$ in M, such that $\forall i \in [0, \ldots, m[\cdot t_i \models f$, $v \models g$. □

The PartialExploration procedure can also be used to check a property having the form $\mathsf{EG} f$. We point out that the proof of Equation (3.2) which was presented in [INH96] cannot be used to prove the following theorem. This is so because the sets returned by the algorithms FwdUntil(H, F) and PartialExploration(H, F) might be different. Thus, we are not able to make a proof which states that EH (FwdUntil($H, \mathcal{L}(f)) \cap \mathcal{L}(f)$) is equal to EH (PartialExploration($H, \mathcal{L}(f)) \cap \mathcal{L}(f)$).

Theorem 3.9. *We suppose that $M = (S, R, I, L)$ is a finite and deterministic transition system, $H \subseteq S$, f is a CTL_X property, and V is the result of* PartialExploration$(I, \mathcal{L}(f)) \cap \mathcal{L}(f)$. *There exists a state $h \in H$ which satisfies $\mathsf{EG}\, f$ (i.e. $M, h \models \mathsf{EG}\, f$) if and only if there exists a state v in the result set of the algorithm* EH(V).

Proof. The proof contains the following four equivalences:

(1) $M, h \models \mathsf{EG}\, f$.

(2) By Corollary 3.7, (2) and because M is finite, (1) if and only if there exists a path $h = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} \boxed{s_n} \xrightarrow{a_n} \ldots \xrightarrow{a_{p-1}} \boxed{s_n}$ in M, such that $\forall i \in [0, \ldots, n] \cdot s_i \in V$. We stress that the states of the path $\boxed{s_n} \xrightarrow{a_n} \ldots \xrightarrow{a_{p-1}} \boxed{s_n}$ forms a cycle within f.

(3) By definition of EH, (2) if and only if EH$(V) \neq \emptyset$. □

From Theorem 3.8 and Theorem 3.9, it follows that the PartialExploration algorithm can be used instead of the FwdUntil algorithm in the equation (3.1), and (3.2).

Corollary 3.10.

$$H \cap \mathcal{L}(\mathsf{E}[q \,\mathsf{U}\, f]) = \emptyset \iff \text{PartialExploration}(H, \mathcal{L}(q)) \cap \mathcal{L}(f) = \emptyset \quad (3.3)$$

$$H \cap \mathcal{L}(\mathsf{EG}\, f) = \emptyset \iff \text{EH}\,(\text{FPG}(H, \mathcal{L}(f))) = \emptyset \text{ } \textit{where} \quad (3.4)$$

$$\text{FPG}(H, F) = \text{PartialExploration}(H, F) \cap F$$

3.4 Forward CTL$_X$ Model Checking with POR

In this section, we construct the evalCTLX algorithm which combines partial-order reduction and the Iwashita's forward symbolic model checking of Section 2.5.2. Our algorithm is based on Equations (3.3) and (3.4). It supposes a deterministic transition system M with a safe and linear process model $\{R_0, R_1, \ldots R_{m-1}\}$. It takes as input a set of states H, and a CTL$_X$ property f. It returns a Boolean value which is true if and only if there exists a state $h \in H$ such that $M, h \models f$.

The evalCTLX algorithm which is defined by Algorithm 3.12 can be used to check whether a transition system $M = (S, R, I, L)$ satisfies a CTL$_X$ property f. In that case, it should be called with the two parameters I and $\neg f$. In this case, evalCTLX(I, f') returns false if and only if M verifies f.

The implementation of the evalCTLX algorithm is guided by the equivalences of Corollary 3.10. It tries to perform POR as deep as possible in the sub-formulae of f. For sub-formulae to which forward model checking does not apply, the standard, backward eval algorithm is called. It is defined by Algorithm 2.7 (c.f. Section 2.5.1). We stress that the algorithm always terminates because it proceeds by induction on the structure of the CTL formula.

3.5 Conclusion

This chapter is dedicated to the construction of the evalCTLX algorithm which verifies CTL$_X$ properties on asynchronous systems. To tackle the state space explosion problem, it combines partial-order reduction and forward model checking. For that purpose, we start by introducing the PartialExploration algorithm. It adapts and extends the ImProviso algorithm which merges POR and symbolic methods and extends it to

Algorithm 3.12: evalCTLX

Global: $\mathtt{M} = (\mathtt{A}, \mathtt{AP}, \mathtt{S}, \mathtt{R}, \mathtt{I}, \mathtt{L})$ a transition system with a process model $\{\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_{\mathtt{m}-1}\}$.

Header: evalCTLX(H, f)

Precondition: M is a deterministic transition, $\mathtt{H} \subseteq \mathtt{S}$, f is a CTL_X property, and $\{\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_{\mathtt{m}-1}\}$, is a safe and linear process model with respect to M and f.

Result: A Boolean value which is true if and only if there exists a state $h \in H$ such that $M, h \models f$

Induction Variant: The length of f.

Implementation:

```
 1  evalCTLX(H, f) {
 2    local r ∈ {true, false}
 3    if (f has the form E[f′ U g′]) {
 4      local T ⊆ S :=
 5          PartialExploration(H, eval(f′))
 6
 7      r := evalCTLX(T, g′)
 8    } else if (f has the form EG f′) {
 9      local T ⊆ S := PartialExploration(H, eval(f′))
10      r := EH(T ∩ eval(f′)) = ∅
11    } else  {
12      r := (H ∩ eval(f)) = ∅
13    }
14    return r
15  }
```

support CTL_X model checking. Then, the PartialExploration algorithm is used as a POR alternative to the forward symbolic model checking approach which allows to verify a subset of CTL properties. Finally, we construct the evalCTLX algorithm itself. To be able to check all CTL properties, it uses when possible our PartialExploration algorithm, otherwise it uses the classical backward model checking.

The reduced state set computed by the PartialExploration algorithm could as well be used in other BDD-based model-checking circumstances: as a filter during fixed-point computations in classical backward model-checking, or even to restrict the BDD of the transition relation before standard, non-POR techniques are applied. It would be interesting to compare the benefits of the reduction in the different approaches. However, the size of the BDD representing either the reduced state space or the reduced transition relation could become unmanageable due to the loss of some symmetry. Actually, in Chapter 7, which is devoted to experimental evaluations, we see that this indeed generally happens.

Chapter 4

Checking LTL Properties: a set-based Approach

In this chapter, we introduce an algorithm which verifies linear temporal logic properties without the next operator (LTL_X) properties. Its name is the *evalLTLX* algorithm. It combines symbolic model checking and partial-order reduction (POR) by using the PartialExploration algorithm which was presented in the previous chapter. Intuitively, it adapts and combines three methods: tableau-based symbolic LTL model checking [CGH97], the forward symbolic CTL model checking [INH96] and the PartialExploration algorithm. More precisely, we proceed as follows:

1. We start from the tableau-based reduction of LTL verification to fair-CTL of E. Clarke et al. [CGH97]. This approach starts by constructing a new transition system P from both the transition system and the property under verification. Then, it looks for a fair path in P.

2. We compute a safe subset V_R of the states of P. V_R is safe in the following sense: if P contains a fair path then P also contains a fair path which is exclusively composed of states of V_R.

3. By using the forward traversal approach of H. Iwashita et al. [INH96], we check whether P contains a fair path which is composed of states of V_R.

The main contribution of this chapter is the evalLTLX algorithm. To bring this construction off, we provide the required theoretical foundation. Moreover, we provide a proof of correctness.

The remainder of this section is structured as follows. Section 4.1 presents some definitions and properties that support the construction and the validation of a correct version of the evalLTLX algorithm. Among other things, it revisits the LTL symbolic model checking algorithm of [CGH97], and shows that partial-order reduction can be applied directly on the product P which was mentioned above. Section 4.2 presents our new approach for LTL model-checking with POR and discusses its correctness. Section 4.3 gives conclusions.

4.1 Problem Theory

In this section, we present the theoretical foundations which are exploited by the evalTLX algorithm. We present a set-based model checking approach which verifies whether a transition system satisfies an LTL property or not. We then present how POR approaches can be applied on a specific class of nondeterministic transition systems. We remind that generally POR approaches are only applicable on deterministic transition systems.

The LTL model-checking algorithm of this section is a slight variant of the one which was initially presented by E. Clarke et al. in [CGH97]. Intuitively, it takes as input a transition system M and an LTL property f. It computes if there exists a path of M which does not satisfy f. To rephrase, it computes if there exists a path of M which satisfies $\neg f$. To achieve this computation, M is rearranged to form a new transition system P where each state of M is created. Each copy is annotated with sub-formulae of f. More precisely, each state of P has the form (s, F) where F is a set of sub-formulae of $\neg f$. Finally, it is guaranteed that if there exists an infinite fair path from a state (s, F), then the paths of M which start from s satisfy all the formulae of F. Thus, all we need to look for is a fair path which starts from a state (i, F) such that i is an initial state and F contains $\neg f$.

4.1.1 Fairness

In the sequel, to verify if a transition system M satisfies an LTL property f, we will need to check if a new transition system P derived from M and $\neg f$ contains a special kind of path called fair path. Intuitively, given a set of states F_k which is called a fairness constraint, a fair path with respect to F_k is an infinite path which contains a state of F_k infinitely often. To deal with fair paths, we extend the concept of transition system.

Definition 4.1 (A Fair Transition System)**.** *A* fair transition system *is a structure* $M = (A, AP, S, R, I, L, F)$ *where:*

- *(A, AP, S, R, I, L) is a transition system, and*
- $F = \{F_0, F_1, \ldots, F_{n-1}\} \subseteq 2^S$ *is a set of fairness constraints.*

The set of states that appear infinitely often on a infinite path π is noted $\inf(\pi)$. A trace is *fair* if and only if each *fairness constraint* $F_k \in F$ is met infinitely often on this path. To rephrase, a path π is said to be fair if and only if for every $F_k \in F$, $\inf(\pi) \cap F_k \neq \emptyset$. A *fair state* s is a state from which a fair path starts.

It is shown in [CGP99] that the set of fair states is the largest set FS which respects the following property:

> For all fairness constraints $F_k \in F$ and all states $s \in FS$, there is a sequence of states of length one or greater from s to a state s' in $FS \cap F_k$.

This characterization can be expressed by means of the two following fixed-point equations. Equation (4.1) has been presented in Section 2.5.1. It is used to compute the set of all states which satisfy a property $\mathsf{E}[f \,\mathsf{U}\, g]$. Equation (4.2) can be used to compute the set of fair states in a backward way [CGP99].

$$\text{evalEU}(F, G) = \mu Z \cdot [F \cap \text{pre}(Z)) \cup G] \tag{4.1}$$

$$FS = \nu Z \cdot [\bigwedge_{k=1}^{n} \text{pre}(\text{evalEU}(S, Z \wedge F_k))] \tag{4.2}$$

To check whether a transition system contains at least one fair state, the forward approach can also be applied (Equation (4.5)) [INH96].

Equation (4.3) has been introduced in Section 2.5.2. Intuitively, it is used to compute whether a transition system verifies a property $\mathsf{E}[f \,\mathsf{U}\, g]$. Equation (4.4) computes the fair states reachable from a cycle, all within F.

$$\mathrm{FwdUntil}(H, F) = \mu Z.[H \cup \mathrm{post}(Z \cap F)] \tag{4.3}$$

$$\mathrm{EHF}(H) = \nu Z \cdot [F \wedge \mathrm{pre}(\bigwedge_{k=1}^{n} \mathrm{pre}(\mathrm{FwdUntil}(F_k, Z) \cap Z)] \tag{4.4}$$

$$I \cap FS = \emptyset \Leftrightarrow \mathrm{EHF}(\mathrm{FwdUntil}(I, S)) = \emptyset \tag{4.5}$$

4.1.2 Product of Two Transition Systems

We now introduce a simple way to annotate the states of a transition system M. It consists in creating a new transition system T which contains the annotations. This method is a variant of the one presented [CGH97]. In the sequel, we show how to encode an LTL property into this new transition system T. Afterwards, T is combined with M to produce a third transition system P which contains the annotated states of M. This is done in such a way that P preserves the structure of both M and T. It means that each path of P corresponds to a path of M and to a path T. To be precise, the opposite is not always true: not all paths of M or T have a corresponding path in P.

Definition 4.2 (Product of M and T)**.** *Let $M = (A, AP, S, R, I, L)$ be a transition system and $T = (A_T, AP_T, S_T, R_T, I_T, L_T, F_T)$ be a fair transition system. The asymmetric product of M and T, denoted $M \times T$, is a fair transition system $P = (A, AP, S_P, R_P, I_P, L_P, F_P)$ such that:*

- $S_P = S \times S_T$
- $R_P = \left\{ ((s,t), a, (s',t')) \mid s \xrightarrow{a}_R s' \wedge t \longrightarrow_{R_T} t' \right\}$
- $I_P = I_T \times I$
- $L_P((s,t)) = L(s)$
- $F_P = \{\{(s,t) \in S_P \mid t \in P_k\} \mid P_k \in F_T\}$

This product is asymmetric because it ignores the labeling of T. We would like to point out that there are plenty of variants of such products in the literature [Hoa85]. Each variant is designed to preserve some specific properties. Besides, contrary to the product defined in [CGH97], the previous definition does not require the two components of each state (s,t) to share the same labeling, i.e. $L(s) = L_T(t)$. As so, it preserves the following useful property: if both M and T have a total relation transition, then P is stuttering-equivalent to M (c.f. Section 2.3.1). Moreover, each fair path of T has a corresponding fair path in P.

Theorem 4.3. *Let M and T be two transition systems such that their transition relations are total and T has at least one initial state. Let P be the product $M \times T$. In such a case, M is stuttering-equivalent to P.*

Proof. The proof consists in showing that for any path of M there is a corresponding path of P, and vice versa.

- We consider any path of M from an initial state s_0: $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$. We suppose that t_0 is an initial state of T. Because T has a total transition relation, there exists a path of T: $t_0 \xrightarrow{b_0} t_1 \xrightarrow{b_1} \ldots$. Thus, there exists by construction a a path of P: $(s_0,t_0) \xrightarrow{a_0} (s_1,t_1) \xrightarrow{a_1} \ldots$. Furthermore, the two paths are equivalent because for all s_i, $L(s_i) = L_P((s_i,t))$.
- We suppose a path of P: $(s_0,t_0) \xrightarrow{a_0} (s_1,t_1) \xrightarrow{a_1} \ldots$. By construction, the following path of M exists: $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ .Furthermore, the two paths are equivalent because for all s_i, $L_P((s_i,t_i)) = L(s_i)$. □

4.1.3 LTL Set-Based Model Checking

This section starts by presenting a variant of the LTL model checking algorithm of E. Clarke et al. [CGH97]. Afterwards, we explain the differences between our variant and the original one. Given a transition system M and an LTL property f, the *tableau* of $\neg f$ is constructed. It is a fair transition system T over the set of propositions which appear in f. Roughly, the construction is based on the set of sub-formulae of f. Each state of the tableau is a set of such sub-formulae . They characterize the sub-formulae of f that are satisfied on fair traces from that state. Initial

states are those that entail f. The fairness constraints ensure that all eventualities occurring in f are fulfilled. The fair traces of the tableau correspond to the traces that satisfy f.

More precisely, the function $\mathrm{el}(f)$ from an LTL formula to its *elementary formulae* is defined:

Definition 4.4 (Function el)**.** *The function* el *from an LTL formula f to a set of elementary sub-formulae of f is recursively defined as follows:*

- *If f is an element $p \in AP$,* $\mathrm{el}(f) = \{p\}$.
- *If f has the form $\neg g$,* $\mathrm{el}(f) = \mathrm{el}(g)$.
- *If f has the form $g \vee h$,* $\mathrm{el}(f) = \mathrm{el}(g) \cup \mathrm{el}(h)$.
- *If f has the form $\mathsf{X}g$,* $\mathrm{el}(f) = \{\mathsf{X}g\} \cup \mathrm{el}(g)$.
- *If f has the form $g \,\mathsf{U}\, h$,* $\mathrm{el}(f) = \{\mathsf{X}(g \,\mathsf{U}\, h)\} \cup \mathrm{el}(h) \cup \mathrm{el}(g)$.

are finite. The function $\mathrm{sat}(g)$ from LTL formulae to subsets of $2^{\mathrm{el}(f)}$ is also needed.

Definition 4.5 (Function sat)**.** *Given an LTL formula f, let g be a sub-formula of f. The function* sat *from LTL formulae to subsets of $2^{\mathrm{el}(f)}$ is defined recursively as follows:*

- *if $g \in \mathrm{el}(f)$,* $\mathrm{sat}(g) = \{s \in 2^{\mathrm{el}(f)} \mid g \in s\}$
- *if g is the form $\neg h$,* $\mathrm{sat}(g) = \{s \mid s \notin sat(h)\}$
- *if g is the form $h \vee i$,* $\mathrm{sat}(g) = \mathrm{sat}(h) \cup \mathrm{sat}(i)$
- *if g is the form $h \,\mathsf{U}\, i$,* $\mathrm{sat}(g) = [\mathrm{sat}(h) \cap \mathrm{sat}(\mathsf{X}\,(h \,\mathsf{U}\, i))] \cup \mathrm{sat}(i)$

Based on the previous definitions, the *tableau* T is constructed.

Definition 4.6 (Tableau T)**.** *Given an LTL property f, a tableau is constructed. It is a fair transition system $T = (A_T, AP_T, S_T, R_T, I_T, L_T, F_T)$ over $A_T = \{\top\}$ and the set AP_T of propositions which appear in f. It is constructed as follows.*

- $S_T = 2^{\mathrm{el}(f)}$

- $R_T \subseteq (S_T \times \{\top\} \times S_T)$ *such that* $(s_t, \top, s'_t) \in R_T$ *if and only if*

$$\bigwedge_{\mathsf{X} g \in \mathrm{el}(f)} [s_t \in \mathrm{sat}\,(\mathsf{X}\, g) \Leftrightarrow s'_t \in \mathrm{sat}\,(g)]$$

- $I_T = \mathrm{sat}(f)$
- $L_T : S_T \to 2^{AP_T} : L_T(s_t) = s_t \cap AP_T$
- $F_T = \{sat\,((f\ \mathsf{U}\ g) \Rightarrow g) \mid (f\ \mathsf{U}\ g)$ *is a sub-formula of* $f\}$

Then, the product P of M and T which was defined in the previous section is computed. It is shown in [CGH97] that M contains a path which satisfies $\neg f$ if and only if there is an infinite fair path π in P such that the two components of all the states of π share the same atomic propositions. Furthermore, the existence of fair traces is captured by the CTL formula $\mathsf{EG}\,\mathrm{true}$ under fairness conditions, to be read as "there exists a fair path. The interest is that this CTL formula can be verified with set-based forward symbolic model checking.

Definition 4.7 (Acceptable Path of P)**.** *Let M be a transition system with a total transition relation, f be an LTL property, T be the tableau of $\neg f$, and P be the product $M \times T$. A path π of P is acceptable if and only if π is an infinite fair path such that the two components of all states (s,t) which appear on π share the same labeling, i.e. $L(s) \cap AP_T = L_T(t)$.*

Theorem 4.8. *Let M be a transition system with a total transition relation, f be an LTL property, T be the tableau of $\neg f$, and P be the product $M \times T$. $M \not\models f$ if and only if P contains an acceptable path π.*

Proof. The proof is the same as the proof of the Theorem 5 of the original approach of Clarke et al. [CGP99, CGH97]. □

This paragraph explains the main difference between our approach and the one of E. Clarke et al. The product of the original approach and the product of this section are not the same. Actually the two components of the states (s,t) of the original product share the same labeling. It means that the set of states of the original product is equal to $\{(s,t) \in S \times S_T \mid L(s) \cap AP_T = L_T(t)\}$. We do enforce such a restriction because we would like to produce a product P which is

stuttering equivalent to M by ensuring that Theorem 4.3 is applicable. As a consequence, the original approach looks for any infinite trace on the constrained product, while our approach looks for an acceptable infinite fair trace on the constrained product.

4.1.4 Total Tableau Construction

We outline here a minor variant of the algorithm which is presented in the previous section. We modify the construction of the tableau to ensure that its transition relation is total. As such by Theorem 4.3, we are sure that the product $P = M \times T$ is stuttering-equivalent to M.

As in the original approach, given a transition system M and an LTL property f, a total tableau $T_\top$ is constructed from $\neg f$. It is the same as the one of the previous section but with one more state $\top$ which a unique transition which is a self-loop. Roughly speaking, when state $\top$ is reached, it is impossible to leave it. While the original tableau T might contain some deadlocks, thanks to state $\top$, $T_\top$ is deadlock-free. Then, the product $P_\top$ of M and $T_\top$ is constructed. Like P, $P_\top$ contains an infinite fair path sharing the same label if and only if M does not satisfy f

Before defining the new construction of the total tableau $T_\top$, we identify the deadlock states of the tableau T.

Definition 4.9 (Deadlock States of T)**.** *Let f be an LTL property f. Let s be an element of $2^{\mathrm{el}(f)}$. The result of the predicate $\mathrm{dead}_T(s)$ is true if and only if*

$$\neg\exists s' \in 2^{\mathrm{el}(f)} \cdot \bigwedge_{\mathsf{X} g \in \mathrm{el}(f)} \left[s \in \mathrm{sat}\left(\mathsf{X}\, g\right) \Leftrightarrow s' \in \mathrm{sat}\left(g\right) \right].$$

Based on that, the total total tableau $T_\top$ is constructed from the LTL property f as follows.

Definition 4.10 (Total Tableau $T_\top$)**.** *Given an LTL property f, a fair transition system $T_\top = (A_\top, AP_\top, S_\top, R_\top, I_\top, L_\top, F_\top)$ is defined as follows*

- $A_\top = \{\top\}$
- $AP_\top = \mathrm{var}(f) \cup \{\top\}$ *where* $\top \notin \mathrm{var}(f)$

- $S_\top = (2^{\mathrm{el}(f)} \cup \{\top\})$
- $R_\top \subseteq (S_\top \times \{\top\} \times S_\top)$ *such that* $(s, \top, s') \in R_\top$ *if and only if*

  (1) $s \neq \top \wedge \neg \operatorname{dead}_T(s) \implies$

  $$\bigwedge_{\mathsf{X} g \in \mathrm{el}(f)} \left[s \in \operatorname{sat}(\mathsf{X}\, g) \Leftrightarrow s' \in \operatorname{sat}(g) \right], \textit{ and}$$

  (2) $s = \top \vee [s \neq \top \wedge \operatorname{dead}_T(s)] \implies s' = \top$
- $I_\top = \operatorname{sat}(f) \cup \{\top\}$
- $L_\top : S_\top \to 2^{AP} \cup \{\top\}$:

  (1) $s_t = \top \implies L_\top(s_t) = \top$

  (2) $s_t \neq \top \implies L_\top(s_t) = s_t \cap AP_\top$
- $F_\top = \{\operatorname{sat}((g \,\mathsf{U}\, h) \Rightarrow h) \mid g \,\mathsf{U}\, h$ *is a subformula of* $f\}$

Given an LTL property f, the tableau T which is defined in the previous section is a sub-transition system of the total tableau $T_\top$. Actually, $A_T = A_\top$, $AP_\top = (AP_T \cup \{\top\})$, $S_\top = (S_T \cup \{\top\})$, $I_\top = (I_T \cup \{\top\})$, and $L_\top = (L_T \cup \{(\top, \top)\})$. Moreover, $R_\top$ is a sub-transition relation of R_T. Actually, the condition (1) includes all the transition of R_T. The condition (2) adds a transition from deadlock states with respect to R_T to the $\top$ state. Moreover, the condition (2) also adds a self loop to the $\top$ state. Thus, $R_\top$ is a total transition relation. As in the original approach, $T_\top$ is composed with the initial system M to produce a new fair transition system $P_\top$.

Theorem 4.11. *Let M be a transition system with a total transition relation. Let $T_\top$ be the total tableau of $\neg f$. Let $P_\top$ be the product of M and $T_\top$. $M \not\models f$ if and only if $P_\top$ contains an acceptable path π*

Proof. The proof is the same as the proof of the Theorem 5 of the original approach of Clarke et al. [CGH97, CGP99]. □

4.1.5 POR and Nondeterministic Transition Systems

In this part, we present the theorem which allows us to apply partial-order reduction on the product $P_\top$ defined in the previous section rather than on the transition system M under verification itself. Furthermore, thanks to this theorem, we are allowed to make use of a process model of M to reduce the product $P_\top$.

In the sequel, we suppose a deterministic transition system $M = (A, AP, S, R, I, L)$ with safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$. We also suppose any fair transition system with a total relation transition $T_T = (A_T, AP_T, S_T, R_T, I_T, L_T, F_T)$.

Theorem 4.12 (c.f [Pel96])**.** *Let M be a transition system with a total transition relation, f be an LTL property, $T_\top$ be the total tableau of $\neg f$, and $P_\top$ be the product $M \times T_\top$. Let $P_R = (S_R, R_R, I_R, L_R)$ be a sub-transition system of $P_\top$, i.e. $P_R \sqsubseteq P$. If each state of P_R is a fully forming state with respect to the process model $\{R_0, R_1, \ldots, R_{m-1}\}$ and the transition relation R_P, then $P_R = M_R \times T$ is the product of a transition system M_R which is stuttering equivalent to M and $T_\top$.*

Proof. The proof is exactly the same as the proof of the Theorem 2.20 which was initially presented in [GKPP99]. □

The product P can be seen as a non-deterministic variant of M. For each transition (s, a, s') of M, there is one or more transitions $((t, s), a, (t', s'))$ in P. We are sure that there is at least one transition because by definition $T_\top$ is deadlock-free. The above theorem shows that partial-order reduction can be easily applied on P based on the safe actions of M. Given a transition system M and a total tableau $T_\top$, Theorem 4.12 can be used to construct an algorithm which produces a stuttering-equivalent reduced version of P. This leads to the following corollary:

Corollary 4.13. *Let $P_\top = (S_\top, R_\top, I_\top, L_\top)$ be the product of M and $T_\top$. Let $P_R = (S_R, R_R, I_R, L_R)$ be a sub-transition of $P_\top$, i.e. $P_R \sqsubseteq P_\top$. If each state of P_R is a fully forming state with respect to the process model $\{R_0, R_1, \ldots, R_{m-1}\}$ and the transition relation $R_\top$, then*

P_R contains an acceptable path if and only if $P_\top$ contains an acceptable path.

Actually, a theorem similar to Theorem 4.12 was firstly introduced in [Pel96] (c.f. Theorem 4.2). We outline here the context in which this theorem was introduced. Given a transition system G and an LTL property f, a Büchi automaton B which accepts the language $\mathcal{L}(\neg f)$ is constructed. The product A of G and B is computed, i.e. $A = G \times B$. Then, a reduced version A' of A is constructed by performing a modified DFS (c.f. Section 2.4.1). At each step a valid ample set is chosen. In practice, it means that the conditions C_1 and C_2 are checked on G alone, while C_0 and C_3 are checked on the whole product. It is shown that A' corresponds to a product of a reduced system G_R and B such that G_R is a property-preserving reduction of G, i.e. $(G, i) \models \mathsf{E}\ \neg f$ iff $(G_R, i) \models \mathsf{E}\ \neg f$.

4.2 Symbolic LTL Model Checking with POR

In this section, we bring together the symbolic LTL model checking approach and the partial-order reduction (POR) technique to create the evalLTLX algorithm. We merge the computation of the reachable state space by means of POR and the fair cycle detection.

Given a transition system $M = (S, R, I, L)$ with a safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$ and an LTL_X property f, our algorithm verifies whether M satisfies f by building a total tableau $T_\top$ from $\neg f$. Then, it checks for the absence of infinite fair traces in the product $P_\top = (S_\top, R_\top, I_\top, L_\top, F_\top)$ of M and $T_\top$. This check is performed symbolically. We proceed as follows.

1. We start by computing the subset $S'_\top$ of $S_\top$ which contains all the states (s, t) such that $L(s) \cap AP_T = L_T(t)$.

2. Then, thanks to the partial exploration algorithm, we compute a sufficiently large subset V_R of $S'_\top$, in the sense that we are certain that the two following sentences are equivalent:

   - P contains an acceptable path π.
   - P contains an acceptable path π which is exclusively composed of states of V_R.

3. Finally, thanks to the forward model checking algorithm, we check whether P contains a fair cycle within the reduced state space

V_R (c.f. Section 2.5.2). It takes place if and only if M does not verify the property f (c.f. Theorem 4.11).

In the rest of this section, we suppose that M is a deterministic transition system with a total relation transition.

Theorem 4.14. *Given an LTL$_X$ property f, $T_\top$ is the total tableau of $\neg f$. $P_\top = (S_\top, R_\top, I_\top, L_\top, F_\top)$ is the product of M and $T_\top$. $Q_\top$ is the set of all states of P which have two components sharing the same labeling. Finally V_R is the result of the algorithm PartialExploration(I_P, $Q_\top$) (c.f. Section 3.2.2). M does not satisfy f if and only if $P_\top$ contains an acceptable fair path π such that all the states of π belongs to $V_R \cap Q_T$.*

Proof. By the specification of the PartialExploration algorithm, we know there exists a transition system $P_R = (S_R, R_R, I_R, L_R) \sqsubseteq P_\top$ which contains only fully forming states with respect to the process model $\{R_0, R_1, \ldots, R_{m-1}\}$ and the transition relation $R_\top$. By Corollary 4.13, we know that P_R is stutteringly-equivalent to P. We now perform the following deductions:

(1) M does not satisfy f.

(2) By LTL semantics, (1) if and only if there is a path π of M which starts from an initial state such that $M, \pi \models \neg f$

(3) By Theorem 4.11, (2) if and only if there is an acceptable path π' in $P_\top$.

(4) By Corrolary 4.13, (3) if and only if there is an acceptable fair path π'' of P_R. Moreover, because $P_R \sqsubseteq P_\top$, $P_\top, \pi'' \models \neg f$, and π'' is an acceptable fair path. Thus, all the states of π'' belong to $Q_\top$.

Besides, we know from the specification of the PartialExploration algorithm that because the path π'' of P_R exclusively contains state of $Q_\top$, all the states of π'' belong to V_R. □

Algorithm 4.1 constructs the evalLTLX algorithm. Both Theorem 4.14 and Equation (4.4), which is relative to fair path detection, help us to deduce that the evalLTLX algorithm is correct.

Algorithm 4.1: evalLTLX

Global: A transition system $\mathtt{M} = (\mathtt{S}, \mathtt{R}, \mathtt{I}, \mathtt{L})$ with a process model $\{\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_{\mathtt{m}-1}\}$, and $\{\mathtt{D}_0, \mathtt{D}_1, \ldots, \mathtt{D}_{\mathtt{m}-1}\} \subseteq 2^{\mathtt{S}}$.

Header: evalLTLX(`f`)

Precondition: `M` is a deterministic transition system, $\{\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_{\mathtt{m}-1}\}$ is a safe process model. Each $\mathtt{D}_\mathtt{i}$ contains the dead states with respect to $\mathtt{R}_\mathtt{i}$. `f` is an LTL_X property.

Result: A boolean value which is true if and only if `M` satisfies the property `f`.

Implementation:

```
1 evalLTLX(f) {
2   local T⊤ := create the total tableau of ¬f
3   local (S⊤, R⊤, I⊤, L⊤, F⊤) := M × T⊤
4   local Q⊤ ⊆ S⊤ := {(s,t) ∈ S⊤ | L(s) ∩ AP_T = L_T(t)}
5   local V_R ⊆ S⊤ := PartialExploration(I⊤, Q⊤)
6   return EHF(V_R) = ∅
7 }
```

4.3 Conclusion

In this chapter, we presented an improved set-based model checking algorithm for verifying LTL_X properties on asynchronous models. Our approach combines the tableau-based reduction of LTL model-checking to fair-CTL from [CGH97], forward state-traversal of fair-CTL formulae from [INH96] used to detect fair cycles, and the PartialExploration algorithm to reduce the forward state traversal.

We note that other symbolic partial-order reduction algorithms which construct a valid POR-reduced reachable state set $\mathit{reduced}(M)$ of a transition system M could have been used instead of our approach. In the same way, other algorithms can be used to detect fair cycles, for instance the classical backward fair CTL model-checking algorithm can be used [CGP99]. Actually, these approaches are valid, and the proof of Theorem 4.14 remains the same. For comparison purposes, we implemented the classical backward as a basis for comparison, though experimental results show the forward approach to be more efficient than the backward approach in all performed experiments.

Chapter 5

Checking LTL Properties: a Bounded Approach

BDD-based symbolic model checking, as exploited in Chapter 3 and Chapter 4, can handle systems with a very large number of states [BCM+92]. Nevertheless, the size of the BDD structures themselves can become unmanageable for some systems. To tackle this problem, *bounded model checking* (BMC) was developed [BCC+03]. Intuitively, it constructs a set $\mathrm{tr}(k, M, \neg f)$ of traces of the system M under verification. Given a predetermined bound k, this set contains the error paths of length k which allow one to discover an error. Then, it checks whether $\mathrm{tr}(k, M, \neg f)$ is empty. If this is the case, no errors are found. In practice, the set $\mathrm{tr}(k, M, \neg f)$ is represented by a propositional formula. A SAT-solver is used to verify that $\mathrm{tr}(k, M, \neg f)$ is empty. BMC offers the advantage of polynomial space complexity in the length of the generated propositional formula. Moreover, it has proven to provide competitive execution times in practice. Nevertheless, it is well-known that the SAT problem is NP complete, and thus hard to solve.

In this chapter, we present an approach which attempts to improve BMC for asynchronous systems by applying partial-order reduction to bounded model checking for verifying linear temporal logic (LTL) properties. Given a system and a property to check, we try to generate a propositional formula such that its satisfiability is easier to check than the one generated by classical BMC. We start from the PartialExploration algorithm of Chapter 3 . We merge a variant of this algorithm with the

BMC procedure. The standard BMC method constructs a propositional formula which represents a finite unfolding of the transition relation and the property. To some extant, our method proceeds in the same way, but instead of using the entire transition relation during the unfolding of the model, we only use a safe subset based on POR considerations. This produces a propositional formula which is well suited for most modern SAT-solvers because it contains less nondeterminism, and therefore reduces backtracking in the SAT search.

To assess the validity of our approach, we reason on the computation tree which is induced by a given transition system (c.f. Section 2.1.1). This offers the advantage that states of the original transition system that can be reached through different paths in the original model, and thus be expanded in different ways, become different states in the computation tree, each with its unique expansion. It matches with the bounded model-checking approach naturally. Actually BMC does not attempt to prevent exploring the same state several times on the same path, as opposed to conventional enumerative model-checkers. We start by introducing two methods which can be combined together for transforming computation trees. The POR method captures partial-order reduction criteria [God96, CGP99, GKPP99]. The idle-extension shows how a finite number of transitions can be added while also preserving temporal logic properties. Then, the concept of *bounded partial exploration* (BPE) is introduced, as a particular instance of a combination of these two methods. The bounded partial exploration approach is inspired from the PartialExploration algorithm. Finally, we present a finite prefix of the tree which is generated by the bounded partial exploration. It is encoded as a propositional formula suitable for BMC.

The remainder of this chapter is structured as follows. Section 5.1 introduces bounded model checking. Besides, it characterizes a broad class of derived computation trees reduced according to partial-order criteria and extended with (finite chains of) idle transitions, and shows that they are visible-bisimilar to the computation tree they derive from. Section 5.2 defines the computation tree corresponding to the BPE method we outlined in the previous section, as a particular instance of this class of derived computation trees. Section 5.3 expresses a constraint system whose solutions are bounded execution paths of the computation tree produced by BPE. Section 5.4 shows how to handle traces which

contain a cycle. Section 5.5 gives conclusions.

5.1 Problem Theory

5.1.1 Bounded Model Checking

This section presents Bounded Model Checking (BMC), which uses SAT-solvers as the underlying computational device [BCC+03]. Given a transition system M and a property f, the idea of BMC is to characterize the set $\mathrm{tr}(k, M, \neg f)$ of error paths of length k. The set is represented as a propositional formula. Then, the emptiness check of $\mathrm{tr}(k, M, \neg f)$ is entrusted to a SAT-solver.

This formula is obtained by combining a finite unfolding of the system's transition relation and an unfolding of the negation of the property being verified. The latter is obtained on the basis of expansion equivalences, such as $p\mathsf{U}q \Leftrightarrow q \vee (p \wedge \mathsf{X}(p\mathsf{U}q))$, which allow us to propagate across successive states the constraints corresponding to the violation of the LTL property. If no counterexample is found, k is incremented and a new execution path is searched. This process is continued until a counterexample is found or a fixed limit is reached.

BMC allows us to check LTL properties on a transition system. Since BMC works on finite paths, an approximate bounded semantics of LTL is defined. Intuitively, the bounded semantics treats differently paths with a back-loop (c.f. Figure 5.1(a)) and paths without such a back-loop (c.f. Figure 5.1(b)). The former is an infinite path formed by a finite number of states. In this case the classical semantic of LTL can be applied. In contrast, the latter is a finite prefix of an infinite path. In some safety properties, such a prefix π is sufficient to show that a path violates a property f. For instance, let f be the property $\mathsf{G}p$. If π contains a state which does not satisfy p then all paths which start with the prefix π violate $\mathsf{G}p$.

Let M be a transition system. Let f be an LTL property. Let k be a natural number. The set $\mathrm{tr}(k, M, \neg f)$ contains all the paths π of M which respect one of the two following conditions:

(1) π is a path of length k, and all infinite paths of M which have π as a prefix necessarily violate property f.

(a) A path with a back-loop (b) A path without a back-loop

Figure 5.1: The two cases for a bounded path [BCC+03].

(2) π is an infinite looping path which does not respect the LTL property f. It can be decomposed into a prefix p of length k, and a back-loop from the last state of p to any state of p.

Definition 5.1 ($\mathrm{tr}(k, M, \neg f)$)**.** *Given a transition system $M = (S, R, i, L)$, an LTL formula f, and a bound $k \in \mathbb{N}$, $\mathrm{tr}(k, M, \neg f) \subseteq tr(k, M)$ is a set of paths such that all paths $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{l-1}} s_\ell \xrightarrow{a_\ell} \ldots \xrightarrow{a_{k-1}} s_k \in \mathrm{tr}(k, M, \neg f)$ satisfy one of the following conditions:*

(1) all infinite paths of M which have π as a prefix necessarily violate the property f.

(2) There exists a $\ell \in [0, \ldots, k]$ such that $s_k \xrightarrow{a_k} s_\ell$. Moreover, the infinite path $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{l-1}} \boxed{s_\ell} \xrightarrow{a_\ell} \ldots \xrightarrow{a_{k-1}} s_k \xrightarrow{a_k} \boxed{s_\ell} \ldots$ does not respect the LTL property f.

The set $\mathrm{tr}(k,M,\neg f)$ is represented by the propositional formula $[\![k, M, \neg f]\!]$ which is constructed as follows:

Definition 5.2 (BMC encoding)**.** *Given a transition system $M = (S, R, I, L)$, an LTL formula f, and a bound $k \in \mathbb{N}$, the set $\mathrm{tr}(k, M, \neg f)$ is encoded by the following propositional formula:*

$$[\![k, M, \neg f]\!] = I(0) \wedge \bigwedge_{i=0}^{i=k-1} R(s_i, s_{i+1}) \wedge \Big([\![k, \neg f]\!] \vee \bigvee_{\ell=0}^{\ell=k} ([\![\ell, k, M]\!] \wedge [\![\ell, k, \neg f]\!]) \Big)$$

where:

- $I(0) \wedge \bigwedge_{i=0}^{i=k-1} R(s_i, s_{i+1})$ *represents the path prefixes π of M of length k.*

- $[\![k, \neg f]\!]$ *represents the paths* $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{k-1}} s_k$ *of* M *of length* k *which respect the following property: all infinite paths* $\pi' = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{k-1}} s_k \xrightarrow{a_k} s_{k+1} \xrightarrow{a_{k+1}} s_{k+1} \ldots$ *which have* π *as a prefix violate the property* f*.*

- $[\![\ell, k, M]\!]$ *is true if and only if there is a transition from* s_k *to* s_ℓ*, i.e.* $s_k \xrightarrow{a_k} s_\ell$*.*

- $[\![\ell, k, \neg f]\!]$ *represents infinite paths* $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{l-1}} \boxed{s_l} \xrightarrow{a_\ell} \ldots \xrightarrow{a_{k-1}} s_k \xrightarrow{a_k} \boxed{s_\ell}$ *which do not respect the LTL property* f*.*

It is shown in [BCC+03] that $M \not\models f$ if and ony if there is a $k \geq 0$ such that $[\![k, M, \neg f]\!]$ is satisfiable. Actually, A. Biere et al. [BCC+03] show the following result. For an upper bound K which depends on f and M, if for all $k \leq K$ $\mathrm{tr}(k, M, \neg f)$ is empty then M satisfies the LTL property f.

Given a propositional formula p produced by the BMC encoding, a SAT-solver decides if p is satisfiable or not. If it is, a satisfying assignment is given that describes the path violating the property. Most of the SAT-solvers apply a variant of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DLL62] to resolve this NP-complete problem. Intuitively, DPLL performs alternatively two phases. The first one proceeds by case-splitting, i.e. it chooses a value for some variable. The second one propagates the implications of this decision that are easy to infer. This method is known as unit propagation. The algorithm backtracks when a conflict is reached. The performance of the DPLL algorithm essentially depends on the number of the propositional variables, but also on the amount of case-splitting and backtracking.

In this chapter, the proposed BMC method builds a propositional formula which is well-suited for the DPLL algorithm, in the sense that less case-splitting occurs than with the classical BMC method. Suppose that we want to find a satisfying assignment for the path $s_0 \xrightarrow{a_0} s_1 \ldots$ and that s_0 is a deterministic state. Once the values of variables representing s_0 are completely decided, the values of variable representing s_1 can be completely decided by the unit propagation phase. Because s_0 is a deterministic state, there is exactly one possibility for the value of variable representing s_1.

5.1.2 Computation Tree Revisited

This section presents a variant of the PartialExploration approach (c.f. Section 3.2.2), called the *BoundedPartialExploration* (BPE) method. In contrast to the original PartialExploration, which performs partial expansions as long as possible, our BPE method imposes a fixed number n of partial expansions for each safe transition relation. If less than n successive steps are enabled for some process, invisible *idle* transitions are performed instead. Figure 5.2 illustrates the resulting computation tree, for two processes with $n = 3$ transitions each.

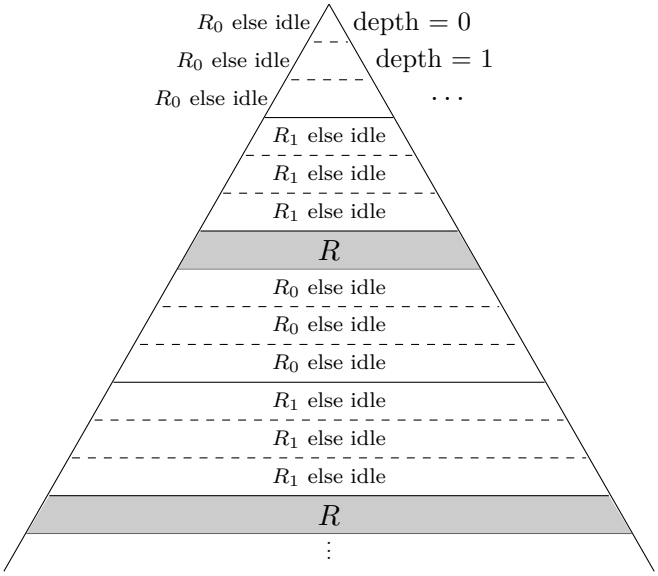


Figure 5.2: BPE$(M, 3)$ with two processes and $n = 3$.

This approach ensures that, at a given depth in the execution, the same (partial or global) transition relation is applied to all states, which greatly simplifies the encoding and resolution of this exploration like a bounded model checking problem using SAT-solvers.

Partial Order Reduction of Computation Trees

We recall that a computation tree is also a transition system. Hence, Theorem 2.20 relative to fully forming states remains applicable to compu-

tation trees. Let $T = (A, AP, S, R, i, L)$ be a computation tree with a safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$. Let $T_R = (A_R, AP_R, S_R, R_R, i, L_R)$ be a computation tree such that T_R is a sub-transition system of T. Let AP' be a subset of AP. If all the states s_r of S_R are fully forming states with respect to AP', the safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$ and the relation transition R then T_R is stuttering-equivalent to T.

Idle-Extension of Computation Trees

Given a computation tree T, the idle-extension consists in adding a finite (possibly null) number of idle transitions on states of T, giving T_{idle}. Intuitively, an idle transition is a transition which does nothing and so does not modify the current state.

Definition 5.3 (Idle-Extension)**.** *Given a computation tree $T = (A, AP, S, R, i, L)$, an* idle-extension *of T is a computation tree $T_{idle} = (A \cup \{idle\}, AP, S_{idle}, R_{idle}, i, L_{idle})$ where $idle \notin A$, $S \subseteq S_{idle}$, $L \subseteq L_{idle}$ and the following conditions hold:*

(1) For all states s, t of S, There is a transition $s \xrightarrow{a} t$ in T if and only if there is a path $s \xrightarrow{idle} s_1 \xrightarrow{idle} s_2 \xrightarrow{idle} \cdots \xrightarrow{idle} s_n \xrightarrow{a} t$ in T_{idle} such that $L(s) = L_{idle}(s) = L_{idle}(s_1) = \cdots = L_{idle}(s_n)$ in T_{idle}.

(2) For all states s, t of S_{idle} where there is no transition $s' \xrightarrow{idle}_{R_{idle}} s$, there is a path $s \xrightarrow{idle} s_1 \xrightarrow{idle} \cdots \xrightarrow{idle} s_n \xrightarrow{a} t$ in T_{idle} such that $a \neq idle$ if and only if $s \in S$, $t \in S$, and $s \xrightarrow{a} t$ in T.

(3) There is no finite or infinite maximal path in T_{idle}: $s_0 \xrightarrow{idle} s_1 \xrightarrow{idle} s_2 \ldots$

The first condition defines in which context an idle transition can be added. The second and third condition ensure that no more than the necessary states are added. In the sequel, we prove that an *idle-extension* of a computation tree T is visible-bisimilar to T and so respects the same CTL_X properties.

Given a state $s \in S$, we write $s \xrightarrow{idle*} s_i$ when a sequence $s \xrightarrow{idle} s_1 \xrightarrow{idle} \ldots \xrightarrow{idle} s_i$ exists in T_{idle}. We call s_i an *idle-successor* of

s. Moreover, s is the *idle-origin* of s_i. Note that the *idle* transition is invisible according to this definition. Since the idle-extension is a tree, two different idle-successors s_i are never shared between multiple idle-origins.

Theorem 5.4. *If the computation tree T_{idle} is an idle-extension of the computation tree T, then T and T_{idle} are visible-bisimilar.*

Proof. To prove the theorem, we define $B \subseteq S \times S_{idle}$ such that $(s, s_i) \in B$ if and only if s_i is an idle-successor of s. We will prove that B is a visible bisimulation between T and T_{idle}. Firstly, we notice that for all $s \in S$, $(s, s) \in B$, so $(i, i) \in B$.

Secondly, we suppose two states $s \in S$ and $s_i \in S_{idle}$ such that $(s, s_i) \in B$ and check that the three conditions of Definition 2.9 are satisfied for B and B^{-1}. By definition of B, s_i is an idle-successor of s.

(1) $L(s) = L_{idle}(s_i)$ by Definition 5.3.

(2) We separately treat the two cases:

B: If $s \xrightarrow{a} t$ in T, then by definition there is a path $s \xrightarrow{idle*} s_n \xrightarrow{a} t$ in T_{idle} such that $B(s, s)$, $B(t, t)$, and $\forall j \in [1, \ldots, n] \cdot (s, s_j) \in B$.

B^{-1}: If $s_i \xrightarrow{a} t$ in T_{idle} then either

$a = idle$: so a is invisible, and t is another idle-successor of s so $B(s, t)$, or

$a \neq idle$: so s_i is the last idle-successor of s. By definition, $s \xrightarrow{a} t$ in T, with $(t, t) \in B$.

(3) We separately treat the two cases:

B: Suppose there exists an infinite path $t_0 \xrightarrow{a_0} t_1 \xrightarrow{a_1} t_2 \cdots$ in T, where $s = t_0$, all a_i are invisible, and $\forall j \in \mathbb{N} \cdot (t_j, s_i) \in B$. Therefore, $\forall j \in \mathbb{N} \cdot t_j \xrightarrow{idle^*} s_i$ and all t_j are idle-origins of s_i. Because T_{idle} is a tree, s_i has only one idle-origin, otherwise there would be more than one path that reach s_i. Hence, $\forall j \in \mathbb{N} \cdot t_j = s$, and so there is a cycle $s \xrightarrow{a_0} s$. According to Definition 5.3, it is impossible, because by definition computation trees do not permit such cycles. In consequence, this case never happens.

B^{-1}**:** Suppose there exists an infinite path $t_0 \xrightarrow{a_1} t_1 \xrightarrow{a_2} t_2 \cdots$ in T_{idle}, where $s_i = t_0$, all a_i are invisible and $\forall j \in \mathbb{N} \cdot (s, t_j) \in B$. Therefore, $\forall j \in \mathbb{N} \cdot s \xrightarrow{idle^*} t_j$ and all t_j are idle successors of s. From that and because T_{idle} is a tree, there is an infinite sequence of idle actions: $s = t_0 \xrightarrow{idle} t_1 \xrightarrow{idle} t_2 \cdots$. According to Definition 5.3, it is impossible. In consequence, this case never happens. □

5.2 The Bounded Partial Exploration Method

In order to accelerate the SAT procedure, we want to consider a modified computation tree T_{idle} derived from a given transition system M. The computation tree T_{idle} has the following particularity: the same (possibly partial) transition relations are applied to all states at a given depth across the tree (see Figure 5.2 of page 98).

This result can be obtained by applying the BPE function, which is a variant of the PartialExploration algorithm (c.f. Section 3.2.2). For the simplicity of the arguments, this section and Section 5.3 considers only the case of finite traces without back-loops (c.f. Section 5.1.1). Section 5.4 explains how to reason about back-loops.

We consider a transition system $M = (S, R, i, L)$ with a safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$ (c.f. Section 2.4.2). For a predetermined natural number n, our BPE function expands exactly n safe transitions of R_0, then n safe transitions of R_1, ..., then n safe transitions of R_{m-1}. If less than n safe transitions are allowed, then idle transitions are performed instead. Then, a full expansion occurs even if there are safe transitions remaining. The computation tree produced by $\mathrm{BPE}(M, n)$ is defined in Listing 5.1, where $\mathrm{BCT}(s, t, d)$ computes the transition relation from state t at depth d using transitions from state s.

Given a transition system M and its computation tree T, the computation tree produced by the BPE function is an idle-extension of a partial order reduction of T. It is therefore visible-bisimilar to T. Actually, it jointly applies both methods of Section 5.1.2. (c.f. Figure 5.3).

We notice that an infinite computation tree is created. However, we point out that this is not a problem because our bounded model checking algorithm will only explore this tree down to a bounded depth k.

Moreover, we also notice that when n equals 0 no partial-order reduction is performed and the resulting computation tree is the same as the original computation tree. Figure 5.3(b) illustrates the result of applying one full cycle of BPE to the computation tree of Figure 5.3(a), with two processes and $n = 3$. The gray arrows of Figure 5.3(a) represent transitions which are ignored by the partial expansion.

Listing 5.1: BPE Function

```
1  BPE(M=(S, R, i, L), {R_0, R_1, ..., R_{m-1}}, n) = (S', R', i', L')
2  where
3
4  S' is an infinite set of states,
5  R' = BCT(i, i', 0),
6  i' is a fresh state,
7
8  BCT(s, t, d) {
9    i := d div (m · n + 1)
10
11   if i < m ∧ s --a--> s' ∈ R_i then
12
13       ⋃_{(s,a,s')∈R_i} {{t --a--> t'} ∪ BCT(s', t', d + 1)} such that
14                   t' is a fresh state
15
16   else if i < m ∧ enabled(s, R_i) = ∅ then
17
18     {t --idle--> t'} ∪ BCT(s, t', d + 1) where
19                   t' is a fresh state
20
21   else
22       ⋃_{(s,a,s')∈R} {{t --a--> t'} ∪ BCT(s', t', i + 1)} such that
23                   t' is a fresh state.
24 }
```

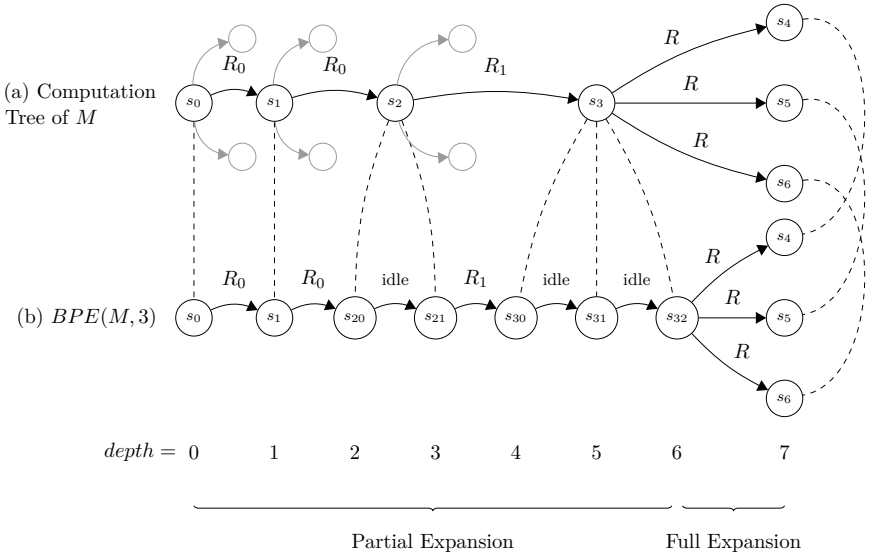


Figure 5.3: Computation tree of M vs $BPE(M,3)$, if s and s' are linked by a dashed line then $(s,s') \in B$ and $(s,s') \in B^{-1}$

5.3 Applying BMC With Partial Order Reduction

This section describes the actual bounded model-checking problem used in our approach. This problem encodes bounded executions of a computation tree resulting from a bounded partial exploration. This kind of exploration is defined in the previous section.

Let $M = (S, R, i, L)$ be a transition system with a safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$. Let f be an LTL_X property. Let n and k be two natural numbers. Our approach uses a variant of the BMC method presented in [BCC+03]. Intuitively, it constructs the set $\mathrm{tr}(k, BPE(M, n), \neg f)$. This set contains the "faulty" paths of length k which result from a bounded partial exploration of the computation tree of M. This set is represented by the propositional formula $[\![n, k, M, \neg f]\!]^{BPE}$. Contrary to the classical bounded model checking methods which use a single transition relation to carry out the required computation on the state space, we define $m + 1$ transition relations. One is the full transition relation R used in Phase-2. The others used to perform partial expansion are derived from the safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$. Given a relation transition R_i, a R_i^{idle} transition relation is created. It contains all the safe transitions of R_i. Moreover, it contains idle transitions on states where no such safe transitions are enabled. Given two states s, t and an action a, $(s, a, t) \in R_i^{idle}$ if and only if either $\{a\} \in enable(s, R_i)$ and $s \xrightarrow{a} t$, or $enable(s, R_i) = \emptyset$, $a = idle$, and $L(s) = L(t)$. Given the number of transition relations m in the process model and the parameter n, we know that each Phase-1/Phase-2 cycle performs $m \cdot n + 1$ steps: n partial expansions for each of the m safe transition relation plus one full expansion. We know which phase is used in the unfolding process at each depth i . Furthermore, if Phase-1 is performed at i, we also know which transition relation is being unfolded (c.f. Figure 5.2). The transition relation $R^{BPE}(i, s, a, s')$ expanded at level i is defined as follows:

Definition 5.5 ($R^{BPE}(i, s, a, s')$)**.** *Given $M = (S, R, s_0, L)$ with a safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$. Let $c = m \cdot n + 1$, $i \in \mathbb{N}$, $s, s' \in S$, and $a \in A$:*

$$R^{BPE}(i, s, a, s') := \begin{cases} R(s, a, s') & \text{if } i \bmod c = m \cdot n \text{ (Phase-2)} \\ R^{idle}_{(i \bmod c) \operatorname{div} n}(s, a, s') & \text{otherwise (Phase-1)} \end{cases}$$

Definition 5.6 (BPE encoding)**.** *Let M be a process model which contains m processes, f be an LTL_X property and $n, k \in \mathbb{N}$:*

$$[\![n, k, M, \neg f]\!]^{BPE} := I(0) \wedge \bigwedge_{i=0}^{i=k-1} R^{BPE}(i, s_i, a, s_{i+1}) \wedge [\![k, \neg f]\!]$$

When the propositional formula $[\![n, k, M, \neg f]\!]^{BPE}$ is built, a decision procedure is used to check its satisfiability. An error is found if $[\![M, \neg f]\!]^{BPE}_{k,n}$ is satisfiable. The validity of this method stems from the following observations. By comparing the construction of BPE(M, n) and $[\![n, k, M, \neg f]\!]^{BPE}$ we show that the latter is the BMC encoding of the former i.e. $[\![n, k, M, \neg f]\!]^{BPE} = [\![BPE(M, n), \neg f]\!]_k$ (restricted to finite traces). The rest derives from the validity of BMC and BPE, as follows:

Theorem 5.7. *Let M be a transition system with a safe process model $\{R_0, R_1, \ldots, R_{m-1}\}$, CT(M) be the computation tree of M, f be an LTL_X formula, and $n \in \mathbb{N}$. There exists $k \geq 0$ such that $[\![k, n, M, \neg f]\!]^{BPE}$ if and only if $M \not\models f$*

Proof.

1. There exists a k such that $[\![k, \text{BPE}(M, n), \neg f]\!]$ is satisfiable if and only if $\text{BPE}(M, n) \not\models_k f$ (c.f. Theorem 2 of [BCC+03]).
2. There exists a k such that $\text{BPE}(M, n) \not\models_k f$ if and only if $BPE(M, n) \not\models f$ (c.f. Theorem 1 of [BCC+03]).
3. $BPE(M, n) \not\models f$ if and only if $CT(M) \not\models f$ because $BPE(M, n)$ is visible bisimilar to $CT(M)$ (c.f. Section 5.2).
4. $CT(M) \not\models f$ if and only if $M \not\models f$ by construction of computation trees. □

5.4 BMC with Back-loops

This section explains how to extend the BPE approach when back-loops are taken into account. Figure 5.4 represents a path π which contains a back-loop. We see that this finite loop induces an infinite path which does not belong to the set resulting from $\text{BPE}(M, 2)$.

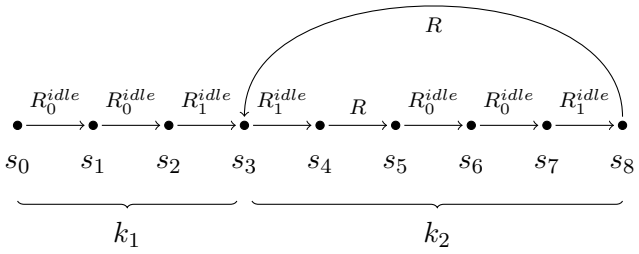


Figure 5.4: A finite path with a back-loop.

This path belongs to the computation tree presented in Figure 5.5. All the execution paths start with a prefix of length $k_1 = 3$ of the form $s_0 \xrightarrow{R_0^{idle}} s_1 \xrightarrow{R_0^{idle}} s_2 \xrightarrow{R_1^{idle}}$, followed by an infinite expansion of the loop of length $k_2 = 6$: $s_3 \xrightarrow{R_1^{idle}} s_4 \xrightarrow{R} s_5 \xrightarrow{R_0^{idle}} s_6 \xrightarrow{R_0^{idle}} s_7 \xrightarrow{R_1^{idle}} s_8 \xrightarrow{R} s_3$.

Given a process model $M = (S, T, i, L)$ with m processes, and lengths k_1 and k_2, we can build variants of BPE(M, n) that correspond to the computation tree of Figure 5.5. Given a computation tree T, these variants are still idle-extensions of partial-order reductions of T, hence stuttering-equivalent to M. We can then construct a complete version of $[\![n, k, M, \neg f]\!]^{BPE}$ with back-loops, similar to Definition 5.2 of Section 5.1.1.

In order to satisfy the LTL bounded semantics presented in Section 5.1.1, the full transition relation R must be used to check whether there exists a back loop or not. If a transition relation R_j^{idle} was used instead, we could have a loop that does not contain a Phase-2 expansion, thus postponing some transitions indefinitely.

5.5 Conclusion

In this chapter, we constructed the BPE bounded model checking algorithm. It checks whether a transition system M contains any traces of a fixed length k which violate an LTL$_X$ properties f. It translates this problem into a propositional formula $[\![n, k, M, \neg f]\!]^{BPE}$ which is passed

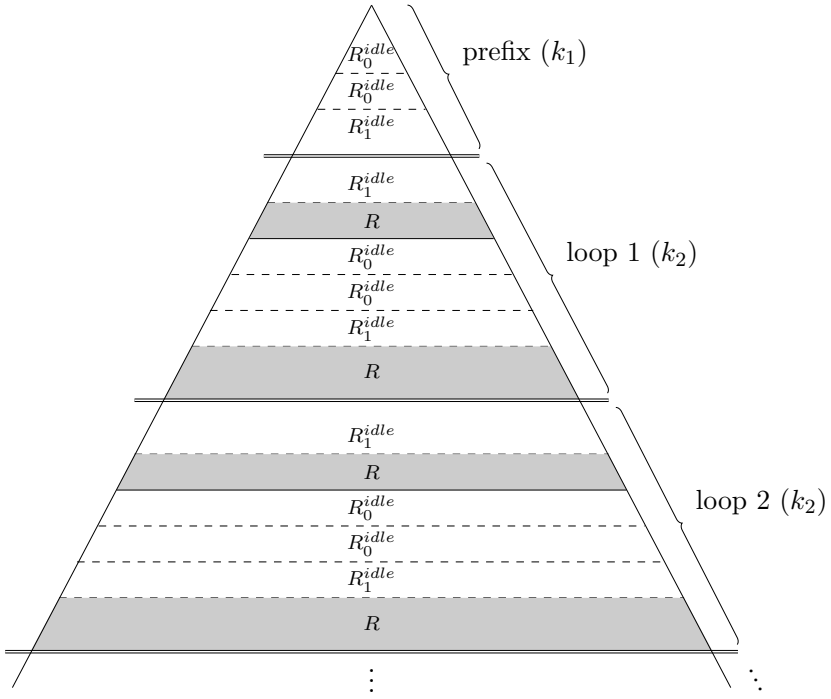


Figure 5.5: variant of $BPE(M, n)$.

on to a SAT-solver. If $[\![n, k, M, \neg f]\!]^{BPE}$ is satisfiable, an evidence that M does not verify f is found. The encoding of $[\![n, k, M, \neg f]\!]^{BPE}$ is is inspired by our PartialExploration algorithm. As shown from experimental results detailed in Chapter 7, this encoding is well suited for checking asynchronous models because it does not encode all the possible interleavings. By doing so, it reduces the search space which the SAT-solver has to deal with.

The ideas behind the BPE algorithm are intuitive. They consist in successively applying a fixed number n of partial expansions followed by a full expansion. On the other hand, our proof of correctness of the BPE algorithm is quite technical. For instance, it reasons about modified computation trees instead of transitions system. By consequence, an important part of this chapter was devoted to show that the BPE algorithm is correct.

With hindsight, we have the intuition that other approaches are possible to show the correctness of the BPE algorithm. For instance, one of them consists in showing that the propositional formula $[\![n, k, M, \neg f]\!]^{BPE}$ encodes the set errors paths of length k of a reduced graph M_R which is stuttering-equivalent to M. This opens the doors for future, challenging theoretical research.

Chapter 6

The Milestones Symbolic Model Checker

This chapter is devoted to the Milestones model checker [VP11b]. Its goal is to concretely implement the algorithms presented in the previous chapters. It defines a language for describing transition systems. It combines all three approaches presented in the chapters 3, 4, and 5.

Milestones is a model checker which was developed from scratch with the Scala programming language [OSV08]. It uses two third-party tools: the Buddy BDD library [LN08], and the Yices SMT solver [DdM06]. We mainly chose to restart from scratch to control all the numerous design choices, from the chosen programming language to the way the tools are invoked. We took great care to produce an implementation which contains as few bugs as possible. To achieve that goal, we applied a rigorous development process. In particular, we divided the implementation into well-defined sub-problems. Each sub-problem was specified precisely before it was implemented, including defining invariant for all non-trivial loops. The main limitation of our approach comes from the fact that Milestones defines it own language, so that we cannot take advantage of the huge number of already developed model in other languages. Finally, the current version of Milestones does not yet provide counter-example when such properties are violated. It is not essential for assessing and comparing the new algorithms constructed in this thesis. However, the counter-example generation is mandatory for any fully-featured model checkers. To be added, the counter-example generation

requires some engineering efforts, but it does not present any technical novelties [CGMZ95].

We designed the Milestones language in such a way that it has a simple to understand semantics. A Milestones program defines a system composed of several processes which manipulates bounded integer variables. Processes can communicate through global variables, and synchronize by rendez-vous.

The language of Milestones and NuSMV are similar in the sense that both define transition systems, constructed with the help of a finite number of variables, and using labeled transition relations. Actually, the NuSMV's input variables (IVAR) essentially amount to transitions labels. Both languages offer the possibility to define more than one initial state. However, the NuSMV language is richer than the Milestones one. It provides a richer data type support, a more expressive expression syntax, and a possibility to define a hierarchy of processes. Nevertheless, the Milestones language offers a built-in mechanism of synchronization by rendez-vous which is not present in NuSMV.

We now make a brief comparison between the Promela language of Spin and the one of Milestones. Spin is an explicit model checker. Its language is inspired by the guarded command language which was introduced by E. Dijkstra [Dij76]. It is procedural. Its syntax defines rich control flow statements such as loop statements, conditional statements, and dynamic creation of processes. None of these control flow statements are defined in the syntax of Milestones. Therefore, Promela is higher-level than the Milestone language. Conversely, Milestones language is simpler than Promela and it has a simpler semantics.

In order to evaluate our approaches, Milestones can translate one of its program P into a NuSMV program [CCGR99] or into a Promela program [Hol97]. To make the comparison as fair as possible, the translation has been turned to ensure that when the program P contains a unique initial state, the resulting state machines are exactly the same as the one generated by Milestones. In the case of NuSMV, the generated BDDs are the same as well.

The remainder of this section is structured as follows. Section 6.1 describes the feature provides by Milestones. Section 6.2 presents the Scala programming language, the Buddy BDD library and the Yices SMT solver that are used by Milestones. Section 6.3 describes the syntax

and the semantics of the Milestones language. It also explains how BDDs can be constructed from Milestones programs. Section 6.4 describes the Milestones part which verifies temporal properties. Section 6.5 describes the Milestones part which translates Milestones systems into a NuSMV systems or into a Spin systems. Section 6.6 gives conclusions.

6.1 Features of Milestones

This section introduces the Milestones model checker. We start by presenting its features. Then, we explain how it is organized. Figure 6.1 graphically represents in a single picture these two points.

Milestones defines its own language for describing systems. We explain this language in Section 6.3. To verify such a system, Milestones provides a set of command-line tools. Twelve commands are provided: two for checking CTL properties, three for checking LTL properties, two for checking deadlocks, two for computing the reachable state space, and finally two for translating a Milestones system into a NuSMV program or into a Spin program.

All the command-line tools start by parsing a provided program. Then, various syntactic checks are performed. For instance, it is checked that all variables which are referred are declared. Afterwards, a flattened Milestone program is created. It contains the instantiation of the variables and the processes. It is a compact representation of the transition system M which will be verified. The next step depends of the command-line which is executed. Here are the possible choices:

- When set-based techniques are applied, M is translated into BDDs. Then, it is possible to compute the reachable state space of M, to verify if M contains some deadlocks, to check whether M respects some CTL properties, and finally to check whether M respects some LTL properties. There exist two command-line tools for each of these operations, one with POR and one without.

- BMC techniques can be applied to check an LTL property f. In this case, M and $\neg f$ are translated into a propositional formula. The satisfiability of this formula is then checked. According to the result of this check, the tool reports an error or not. As with the set-based approach, there exist two command-line tools which

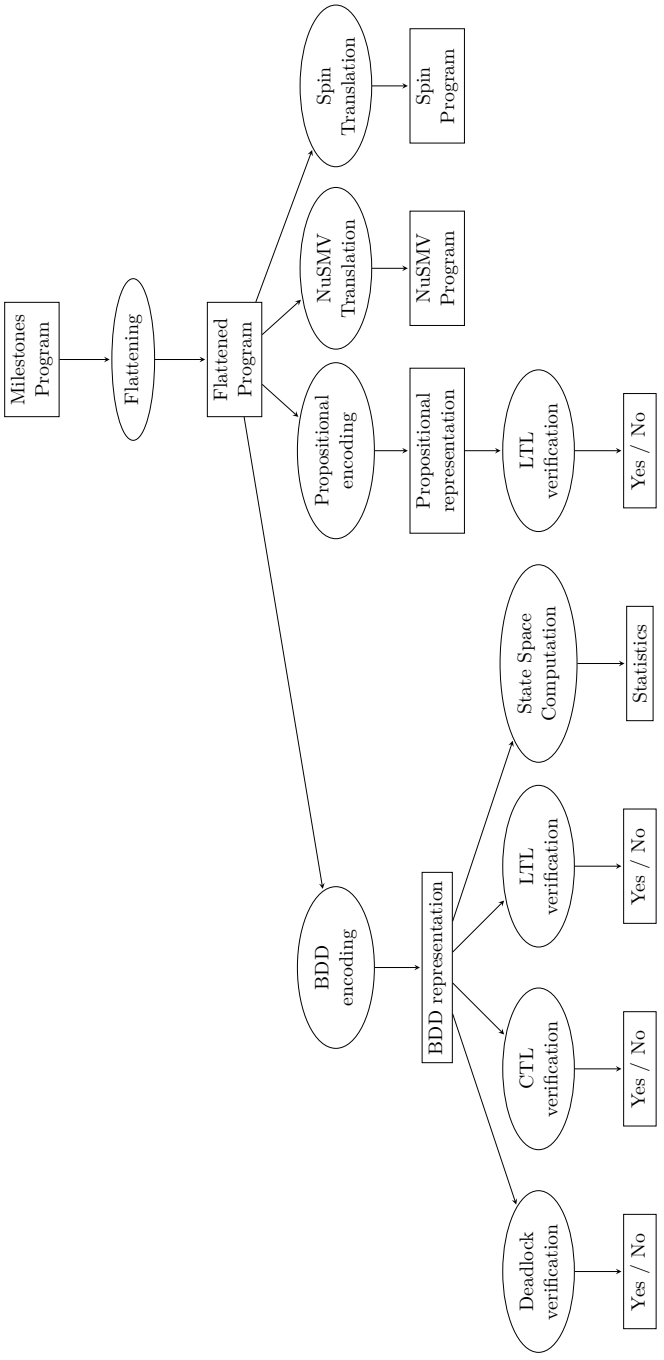


Figure 6.1: Organization of Milestones

perform bounded model checking, with or without applying POR methods.

- The last two tools allow M to be translated into a NuSMV system [CCGR99] or into a Promela system [Hol97].

6.2 Development Environment

In this section, we present the Scala programming language [OSV08] which is used to develop Milestones, as well as the two third-party components which are used by Milestones: the Buddy BDD library to represent sets and the Yices solver to check if a propositional formula is satisfiable.

The programming language which was used to develop Milestones is Scala. We transcribe here a description of the Scala language which is taken from the Scala website [EPF12]. This description summarizes in a few lines why we make this choice.

> Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages. Code sizes are typically reduced by a factor of two to three when compared to an equivalent Java application. Existing Java code and programmer skills are fully re-usable. Scala programs run on the Java VM, are byte code compatible with Java so you can make full use of existing Java libraries or existing application code. You can call Scala from Java and you can call Java from Scala, the integration is seamless.

With hindsight, we experience that working with such a rich language has both advantages and drawbacks. On the one hand, this richness was exploited to write a concise source code. For instance, Scala offers a mechanism of class pattern matching which is very helpful to decompose CTL and LTL expressions. On the other hand, this richness makes the semantic of Scala not easy to master. Therefore, the time saved on one side was lost on the other.

Milestones represents the sets which are manipulated by means of BDDs. For that, it uses the Buddy BDD library [LN08]. It is a well-known BDD library which was developed by J. Lind-Nielsen, as part of his Ph.D. about model checking of finite state machines [LN00]. The package has evolved from a simple introduction to BDDs to a full-blown BDD package with all the standard BDD operations. In practice, Buddy is simple to use and provides an efficient BDD package implementation [LNAH$^+$01].

In the context of BMC, we use the Yices SMT solver [DdM06]. Yices is both an SMT solver and a SAT solver that decides the satisfiability of arbitrary formulas containing uninterpreted function symbols with equality, integer arithmetic, fixed-size bit-vectors, etc. We decide to use Yices because it has a simple input language. This language allows one to easily encode the bounded integer variables manipulated by Milestones as fixed-size bit-vectors. In practice, Yices is an efficient SMT solver. For instance, Yices took first place in seven of the twelve divisions of SMT-COMP'07 competition, and second place in three more [BDOS08].

We now provide some statistics about the source code of Milestones. It is organized in seven packages. A package is dedicated for each of the following concepts: the parser, the flattened model, the temporal expressions, the BDDs, the set-based verification, the BMC verification, and the translation into Spin and NuSMV. Those packages are organized into 66 files, around 150 methods, and around 5000 lines of code.

6.3 Input Language

Milestones defines a language for describing transition systems. It is designed to model systems which are composed of a finite number of processes. This section explains the syntax and the semantics of a Milestones program. Moreover, we explain how BDDs are generated from such a system.

6.3.1 Syntax of Milestones Systems

A Milestones system is composed of three parts:

1. The declaration of the global actions and the global variables.

2. The definition of several process types. Each process type is used as a blueprint to create instances of itself. Each process type defines local actions and local variables. Besides, it specifies local guarded commands which define the behavior of processes of this type.

3. The main part creates particular instances of the process types. The main part is also in charge of defining global guarded commands which will be used during rendez-vous synchronizations.

To make an analogy with the object-oriented programming languages, § 2 plays the role of class definition, and § 3 corresponds to object creations.

We now provide the grammar which is used to define a Milestones system. Listing 6.1 shows the Milestones system of the running example presented in Chapter 2 (c.f. page 12). This grammar is a simplified version of the real one. Actually, for clarity, we hide some details. For instance, we suppose that all expressions are well parenthesized. In reality, some usual priority rules between operators have been fixed. Moreover, we do not specify the syntax of the CTL and LTL properties, but in the sequel we discuss it.

```
// PROGRAM
SYSTEM          →   ACTIONS
                    VARIABLES
                    (PROC_TYPE)*
                    MAIN
                    CTL
                    LTL

// PROCESS TYPE
PROC_TYPE       →   "LOCAL" ID
                      ACTIONS
                      VARIABLES
                      LCASE
                      SYNC
                    "END"

// MAIN PART
MAIN            →   "MAIN"
                       PROC_INST
                       GCASE
                    "END"

PROC_INST       →   "PROC"
```

(ID ":=" ID[CONST])*
"END"

// G. COMMANDS
LCASE → "CASE"
("[" CONST "]" "[" ID "]" BEXPR ":" ASS)*
"END"

GCASE → "CASE"
("[" ID "]" BEXPR ":" ASS)*
"END"

ASS → (ID ":=" MEXPR)*

// ACTIONS
ACTIONS → "ACTIONS" (ID)* "END"

SYNC → "SYNC" (ID)* "END"

// VARIABLES
VARIABLES → "VARIABLES"
(ID ":" CONST ":" [":=" CEXPR])*
"END"

// EXPR
ID → an identifier
CONST → a natural number

CEXPR → CONST |
"(" CEXPR MOP CEXPR ")"

MEXPR → CONST |
ID | ID "." ID | ID "[" CEXPR "]" "." ID
"(" MEXPR MOP MEXPR ")"

BEXPR → "true" | "false" |
"¬" "(" BEXPR ")" |
"(" BEXPR BOP BEXPR ")"
"(" MEXPR EOP MEXPR ")"

MOP → + | − | × | / | %
EOP → = | ≠ | < | > | ≤ | ≥
BOP → ∧ | ∨ | ⟹ | ⇔

// PROPERTIES
CTL → CTL properties

LTL → LTL properties

Listing 6.1: Running example in the Milestones Language

```
1  ACTION
2    Gpp; GE0;  xExMy;
3  END
4
5  VARIABLE
6    g : 1 := 0;
7  END
8
9  LOCAL A
10   ACTION
11     Xpp; Zpp;
12   END
13
14   VARIABLE
15     pc: 2 := 0;
16     x:  1 := 0;
17     z:  1 := 0;
18   END
19
20   CASE
21     [0] [Xpp] true : pc := 1; x :=  x + 1;
22     [1] [Zpp] true : pc := 2; z := z + 1;
23     [2] [xExMy] true :
24   END
25
26   SYNC
27     xExMy;
28   END
29 END
30
31 LOCAL B
32   ACTION
33     Ypp;
34     YE0;
35   END
36
```

```
37    VARIABLE
38      pc: 1 := 0;
39      y:  1 := 0;
40    END
41
42    CASE
43      [0] [Ypp] g == 0 : pc := 1; y :=  y + 1;
44      [1] [YE0] g == 0 : pc := 0; y :=  0;
45      [1] [xExMy] true :
46    END
47
48    SYNC
49      xExMy;
50    END
51  END
52
53  GLOBAL
54    PROC
55      a: A[1];
56      b: B[1];
57    END
58
59    CASE
60      [Gpp] a[0].pc == 2 & b[0].pc == 1 & g == 0:
61        g := g + 1;
62      [GE0] a[0].pc == 2 & b[0].pc == 1 & g == 1:
63        g := 0;
64      [xExMy] a[0].pc == 2 & b[0].pc == 1:
65    END
66  END
67
68  CTL
69    AG EF (g == 1);
70  END
71
72  LTL
73    F(G g == 1);
74  END
```

Listing 6.2: Milestones program T

```
1  ACTION a_0, ..., a_a END
2
3  VARIABLES
4   VAR_DECL_0
5   ...
6   VAR_DECL_v
7  END
8
9  PROC_TYPE_0
10 ...
11 PROC_TYPE_p
12
13 MAIN
14   PROC_INST_0
15   ...
16   PROC_INST_j
17
18   CASE
19     ggc_0
20     ...
21     ggc_h
22   END
23 END
```

6.3.2 Semantic of Milestones Systems

From a Milestones program T defined in Listing 6.2, we construct the following 6-tuple $(A_g, V_g, P_g, G_g, C_g, L_g)$ where:

- $A_g = \{a_0, a_1, \ldots, a_a\}$ is a set of *global actions* (c.f. Listing 6.2, line 1).
- $V_g = (v_0, v_1, \ldots, v_v)$ is a sequence of *global variables*. They are defined by the $\texttt{VAR\_DECL}_\texttt{i}$ declarations (line 3). For each $\texttt{VAR\_DECL}_\texttt{i}$ = '$\texttt{v}_\texttt{i} : \texttt{nbr}_\texttt{i}\,[:= \texttt{Cexpr}_\texttt{i}]$' a global variable is defined, where $\texttt{v}_\texttt{i}$, $\texttt{nbr}_\texttt{i}$, and $\texttt{Cexpr}_\texttt{i}$ are respectively the identifier $\texttt{v}_\texttt{i}$ of the variable, the

Listing 6.3: Milestones Process Type $\texttt{PType}_\texttt{i}$

```
1  ACTION a_0,...,a_a' END
2
3  VARIABLES
4     VAR_DECL_0
5     ...
6     VAR_DECL_v'
7  END
8
9  CASE
10   lgc_0
11   ...
12   lgc_h'
13 END
14
15 SYNC s_0,...,s_s' END
```

number of bits on which $\texttt{v}_\texttt{i}$ is encoded, and a mathematical expression which contains only constants but not variables. Each variable $\texttt{v}_\texttt{i} \in V_g$ has a finite domain $\text{domain}(\texttt{v}_\texttt{i}) = [0 \dots 2^{\texttt{nbr}_\texttt{i}}[$. Besides, a default set $\text{default}(\texttt{v}_\texttt{i})$ is also defined. If a default expression $\texttt{Cexpr}_\texttt{i}$ is provided, $\text{default}(\texttt{v}_\texttt{i}) = \{\text{value}(\texttt{Cexpr}_\texttt{i})\}$ where $\text{value}(\texttt{Cexpr}_\texttt{i})$ is the evaluation of $\texttt{Cexpr}_\texttt{i}$, otherwise $\text{default}(\texttt{v}_\texttt{i}) = \text{domain}(\texttt{v}_\texttt{i})$.

- A sequence of *processes* P_g is defined by the list of declarations: $\texttt{PROC\_INST}_\texttt{i}$ = '$\texttt{name}_\texttt{i} := \texttt{PType}_\texttt{i}\ \texttt{id}_\texttt{i}\ [\texttt{nbr}_\texttt{i}]$' (c.f. Listing 6.2, line 14). P_g contains $\texttt{nbr}_0$ process of type $\texttt{PType}_0$, followed by $\texttt{nbr}_1$ process of type $\texttt{PType}_1$, ... Each process is a 5-tuple $P_i = (A_i, V_i, G_i, Y_I, pc_i)$. The set $A_i = \{\texttt{a}_0, \dots, \texttt{a}_{\texttt{a}'}\}$ (c.f. Listing 6.3, line 1) contains the local action of P_i. The set $V_i = \{\texttt{v}_0, \dots, \texttt{v}_{\texttt{v}'}\}$ (line 3) contains the local variables of P_i. It is created in a similar way as the global variables. The set $Y_i = \{a_0, \dots, a_{a'}, s_0, \dots, s_{s'}\}$ contains the local actions on which P_i is synchronized. On the one hand, Y_i has local actions that are not synchronized with any other processes. On the other hand, it is also synchronized on some global actions $\{s_0, \dots, s_{s'}\}$ (line 15). The variable $pc_i \in V_i$

is a distinguished local program counter variable. Its identifier is necessarily 'pc'. Each process contains its own instances of the local actions and variables. The guarded commands of G_i refer to those instances, as well as the global actions and variables.

The sequence of processes P_g is required to generate both a transition relation and a process model of a system T. To construct them, we also define the sequence var(T) of all the global and local variables. It starts by the global variables of the system, followed by the local variables V_0 of P_0, followed by the local variables V_1 of $P_1, \ldots$. The set act(T) which contains the local and global actions of T is also defined.

- The *global guarded commands* $G_g = \{\texttt{ggc}_0, \ldots, \texttt{ggc}_\texttt{h}\}$ (Listing 6.2, line 18) and the *local guarded commands* $G_i = \{\texttt{lgc}_0, \ldots, \texttt{lgc}_{\texttt{h}'}\}$ of each process P_i (c.f.Listing 6.3, line 9) are used to generate a transition relation. Each local guarded command has the following form [`pc`] [`action`] `condition` : `assignments` where:
  - `pc` $\in$ domain(pc_i).
  - `action` is an action $a \in (A_g \cup A_i)$.
  - `condition` is a BEXPR expression constructed from the variables belonging to $(V_g \cup V_i)$.
  - `assignments` is a list of assignments $\texttt{u}_0 := \texttt{e}_0, \texttt{u}_1 := \texttt{e}_1, \ldots, \texttt{u}_\texttt{u} := \texttt{e}_\texttt{u}$. Each $\texttt{u}_\texttt{i}$ is a distinct variable of $(V_g \cup V_i)$. Each $\texttt{e}_\texttt{i}$ is a MEXPR expression which only refers to the variables of $(V_g \cup V_i)$.

  On the one hand, the local actions generate transitions which are executed by only one process. On the other hand, the global actions generate transitions which are executed synchronously by several processes. To avoid that a global variable is assigned more than once during a rendez-vous, a local guarded command which refers to a global action can only refer to the local variables of its process Each global guarded has the form [`action`]`condition` : `assignments` where:
  - `action` is an action $a \in A_g$,

  - `condition` is a BEXPR expression constructed from the variables of the system.
  - `assignements` is a list of assignments $\mathtt{u}_0 := \mathtt{e}_0, \mathtt{u}_1 := \mathtt{e}_1, \ldots, \mathtt{u}_u := \mathtt{e}_u$. Each $\mathtt{u}_i$ is a distinct variable of V_g. Each $\mathtt{e}_i$ is a MEXPR expression which uniquely refers to the variables of the system.

- C_g is a sequence CTL formulae and L_g is a sequence LTL formulae. The syntaxes of LTL and CTL properties correspond to their mathematical definition presented in Section 2.2.1. All the propositions which appear in a Milestones formulae are also BEXPR. Moreover, they only refer to global variables. For instance, the LTL Milestones formula $\mathsf{FG}\,(g == 1)$ contains the Boolean expression $g == 1$ which is also considered as an atomic proposition.

The object V_g and P_g are sequences because the induced order of these sequences is useful to create the variable order of the generated BDDs. C_g, L_g are sequences so as to check those properties in the same order they appear in the system.

Given a valuation s of the variables, the MEXPR (or BEXPR) expression e can be evaluated to a natural number (or to true or false). In the sequel, the states of the generated transition system will naturally provide such a valuation which is noted value(s, e). To deal with finite domains, the expressions are evaluated with the help of modular arithmetic. For instance, if the domain of x is $\{0, \ldots, 3\}$, the value of x is 3, the domain of y is $\{0, \ldots, 15\}$, and the value of y is 12, the evaluation of $2x + y$ proceed as follows. The domain of x is coerced to $\{0, \ldots, 15\}$. Then, $2x$ is evaluated to 6. The resulting expression $6 + y$ evaluates to 18, which is again coerced to $\{0, \ldots, 15\}$, giving 2.

6.3.3 Transition System Construction

Given a 6-tuple $T = (A_g, V_g, P_g, G_g, C_g, L_g)$ generated from a Milestones program, we now describe how a transition system $M = (A, AP, S, R, I, L)$ is generated:

- $A = \mathrm{act}(T)$.

- AP is the infinite set of Boolean expressions generated by the BEXPR rule.

- S is the cross product of the domains of the variables of $\text{var}(T) = (v_0, v_1, \ldots, v_w)$, i.e.$S = \{\text{domain}(v_0) \times \cdots \times \text{domain}(v_w)\}$. Because each domain is finite, S is also finite. We notice that each state s defines a valuation of the variables of $\text{var}(T)$.

- R is defined by means of the global and local guarded commands. A local guarded command $[\texttt{pc}]\,[\texttt{action}]\,\texttt{cond} : \texttt{assignments}$ of a process P_i can be rewritten as follows $[\texttt{a}]\,(\texttt{pc}_\texttt{i} = \texttt{pc}) \wedge \texttt{cond} : \texttt{assignments}$. Actually, the [pc] part is helpful to create the process model but not the transition relation, so we consider reduced local guarded command of the form $[\texttt{a}]\,\texttt{cond} : \texttt{assignments}$.

  For each action $a \in \text{act}(T)$, a set of transitions $R_a \subseteq \{s \xrightarrow{a} t\}$ is generated. R_a contains all the transitions from s to t which satisfy the following conditions, and only those.

  (1) For all processes P_i synchronized on a, there exists a local guarded command $[\texttt{a}]\,\texttt{cond:}\ \texttt{u}_0 := \texttt{e}_0, \texttt{u}_1 := \texttt{e}_1, \ldots, \texttt{u}_\texttt{u} := \texttt{e}_\texttt{u}$ in G_i such that $\text{value}(s, \texttt{cond}) = \text{true}$. For each variable $\texttt{u}_\texttt{k}$ which appears in the left side of an affectation, $\text{value}(t, \texttt{u}_\texttt{k}) = \text{value}(s, \texttt{e}_\texttt{k})$. For each variable v of the systems which does not appear in the left side of an affectation: $\text{value}(s, v) = \text{value}(t, v)$.

  (2) For all processes P_i which are not synchronized on a, and for each local variable $v_i \in V_i$: $\text{value}(s, v_i) = \text{value}(t, v_i)$.

  (3) If a is a global action, there exists a global guarded command $[\texttt{a}]\,\texttt{cond:}\ \texttt{u}_0 := \texttt{e0}, \ldots, \texttt{u}_\texttt{u} := \texttt{e}_\texttt{u}$ in G such that $\text{value}(s, \texttt{cond}) = \text{true}$. For each variable $\texttt{u}_\texttt{k}$ which appears in the left side of an affectation, $\text{value}(t, \texttt{u}_\texttt{k}) = \text{value}(s, \texttt{e}_\texttt{k})$. For each global variable $v \in V_g$ which does not appear in the left side of an affectation, $\text{value}(s, v) = \text{value}(t, v)$.

  The whole relation transition R is the union of the R_a, i.e.$R = \bigcup_{a \in \text{act}(T)} R_a$.

- I is the cross product of the default set of $\text{var}(T) = (v_0, v_1, \ldots, v_w)$, i.e.$I = \{\text{default}(v_0) \times \text{default}(v_1) \times \cdots \times \text{default}(v_w)\}$.

- $L : S \rightarrow 2^{AP} : s \rightarrow \{ap \in AP \mid \text{value}(s, ap) = true\}$.

6.3.4 Process model Construction

In the previous section, a transition system is constructed from a Milestones program. This section explains how a process model (c.f. Section 2.4.2) is extracted from such a program. We firstly define the concept of *safe guarded command*. A safe guarded command is a local guarded command which only refers to a local action and local variables. Intuitively, a safe guarded command of any process generates transitions which are independent of the transitions generated from other processes.

Definition 6.1 (Safe Guarded Command)**.** *Given a process P_i, each of its local guarded command* $[\mathtt{a}]\ \mathtt{cond{:}u_0} := \mathtt{e_0}, \ldots, \mathtt{u_u} := \mathtt{e_u}$ *is safe if and only if it satisfies the following three conditions:*

(1) The `cond` *expression only refers to local variables of P_i.*

(2) Each variable $\mathtt{u_k}$ *which appears in a left part of an assignment is a local variable.*

(3) Each expression $\mathtt{e_k}$ *which appears in a right part of an assignment refers only to local variables.*

Then, the concept of *safe* action is defined. Intuitively, all safe actions respect the partial order condition C_1 (c.f. Section 2.4.1). We notice that the inverse is not necessarily true. Furthermore, all safe actions are invisible (c.f. Definition 2.6) with respect to all CTL and LTL properties because they only refer to global variables.

Definition 6.2 (Safe action)**.** *Given a process $P_i = (A_i, V_i, G_i, Y_i, pc_i)$, and a local action* $\mathtt{a_i} \in A_i$, $\mathtt{a_i}$ *is a safe action if and only if the following condition is satisfied.*

If $\mathtt{a_i}$ *is referred by a local guarded command $g =$ '*$[\mathtt{pc}][\mathtt{a_i}]\ \mathtt{cond{:}\ ass}$*' of G_i, then all the local guarded commands '*$[\mathtt{pc}]\,[\mathtt{a}]\ \mathtt{cond} : \mathtt{ass}$*' of G_i with the same program counter value (*`pc`*) than g_i are safe guards.*

Then, the concept of *linear* action is defined. Intuitively, all linear and safe actions a, will allow us to construct an ample set $\{a\}$ which respects the two POR conditions C_1 and C_4. We notice that the inverse is not necessarily true.

Definition 6.3 (Linear Action)**.** *Given a process* $P_i = (A_i, V_i, G_i, Y_i, pc_i)$*, and an action* $\mathtt{a_i} \in A_i$*,* $\mathtt{a_i}$ *is a linear action if and only if the following condition is satisfied.*

If $\mathtt{a_i}$ *is referred by a local guarded command* $g =$ *'*$[\mathtt{pc}][\mathtt{a_i}]\,\mathtt{cond}\colon \mathtt{ass}$*' of* G_i*, then it is impossible to find a state* $s \in S$ *and two guarded commands* $g_1 =$ *'*$[\mathtt{pc}][\mathtt{a_1}]\,\mathtt{cond_1}\colon \mathtt{ass_1}$*', and '*$[\mathtt{pc}][\mathtt{a_2}]\,\mathtt{cond_2}\colon \mathtt{ass_2}$*' of* G_i *with the same program counter value (*$\mathtt{pc}$*) such that* $\mathtt{cond_1}$ *and* $\mathtt{cond_2}$ *are both evaluated on* s *to* true*.*

To determine if an action is linear, on can verify that a set of guards are disjoint. Milestones uses BDDs to make that check. Actually, the guard of a guarded command can be represented by a BDD. To verify whether two guards are disjoint, we check that the intersection of the two corresponding BDDs is empty. It is well-known that in general this check is an NP-complete problem, but in practice, this check is performed fast because the conditions are small and refer to a small number of variables. Nevertheless, Milestones can disable this check and assume that all safe guarded commands are also linear.

For each process P_i, a set $A_{s_i} = \{a_i \in A_i \mid a_i \text{ is safe}\}$ is constructed. One can see that $A_s = \{A_{s_0}, A_{s_1}, \ldots, A_{s_{n_0}}\}$ is a safe process model because all the action sets A_{s_i} contain only actions which are invisible and respect the POR condition C_1 (c.f. Section 2.4.1).

For each process P_i, a set $A_{l_i} = \{a_i \in A_i \mid a_i \text{ is safe} \wedge a_i \text{ is linear}\}$ is constructed. One can see that the set $A_l = \{A_{l_0}, A_{l_1}, \ldots, A_{l_{n_0}}\}$ is a safe and linear process model because all the actions set A_{l_i} contain only transitions a_i which are invisible and such that $\{a_i\}$ respects the POR condition C_1.

6.3.5 From Transition systems to BDDs

In this section we suppose a transition system $M = (A, AP, S, R, I, L)$ generated from a Milestones system T. In order to perform the set-based verification of M, it is encoded into BDDs (c.f. Section 2.5.3).

To encode the transition relation R into a BDD, a second set V' of variables is needed. V and V' are disjoint. The variables V will be used to represent the source states. The variables set V' will be used to represent the target states. Based on V and V', for each action $a_i \in A$ a Boolean formula F_{a_i} over $(V \cup V')$ is constructed. It is a BEXPR formula

(c.f. Section 6.3.1). It is used to characterize the set of transitions labeled with a_i, i.e. R_{a_i}. The formula F_{a_i} is constructed in such a way that given two states s and t, it is evaluated to true if and only if $s \xrightarrow{a} t$ belongs to R_{a_i}. Based on that, a Boolean formula F which represents R is constructed. To keep track of which action is executed, one more variable $v_{action} \notin (V \cup V')$ is required. The domain of v_{action} is A. The formula F is constructed as follows $F = \bigvee_{a_i \in A} (v_{action} = a_i \wedge F_{a_i})$. Then F is expanded into a Boolean representation F_p. Finally F_p, is converted into a BDD which represents R.

To encode the initial states I into a BDD, for each variable $v_i \in \text{var}(T)$ a BEXPR expression H_{v_i} is constructed. It is used to characterize the set of initial value of v_i. The formula H_{v_i} is constructed in such a way that given a value v, it is evaluated to true if and only if $v \in \text{default}(v_i)$. Based on that, the Boolean formula H which represents I is constructed, i.e. $H = \bigwedge_{v_i \in V} H_{v_i}$. Then H is expanded into a Boolean representation H_p which is converted into a BDD representing I.

L is not encoded as such into BDDs. But, given a valid BEXPR formula f, BDDs allows us to easily compute the set of states which satisfies the formula f, i.e. $\{s \in S \mid \text{value}(s, f) = true\}$

S, A and AP are not explicitly encoded into BDDs because the model checking algorithms do not use them.

We now present the order of the variables which is generated from a Milestones system T. This system defines two sequences of variables $\text{var}(T) = (v_0, v_1, \ldots, v_w)$, and its primed version $\text{var}'(T) = (v'_0, v'_1, \ldots, v'_w)$ which is used to encode the relation transition. It also defines the v_{action} variable which is used to encode the performed actions. According of the size of the variables, those two sequences are transformed into two sequences of Boolean variables $\text{bvar}(T) = (b_0, b_1, \ldots, b_b)$ and $\text{bvar}'(T) = (b'_0, b'_1, \ldots, b'_b)$. The Boolean sequence $\text{bvar}(T)$ firstly starts by the Boolean variables which encode v_0, followed by the Boolean variables which encode v_1, ... The sequence $\text{bvar}'(T)$ is constructed in the same way but with the primed variables. The v_{action} variable is encoded by the sequence of Boolean variables $(b^a_0, b^a_1, \ldots, b^a_c)$ where c depends on the size of $\text{domain}(v_{action})$. The order of the variables applies the interlaced method of Section 2.5.4: $b^a_0 \prec b^a_1 \prec \cdots \prec b^c_a \prec b_0 \prec b'_0 \prec b_1 \prec b'_1 \cdots \prec b_b \prec b'_b$.

6.4 Verification

Given a transition system M produced from a Milestones system $T = (A_g, V_g, P_g, G_g, C_g, L_g)$, Milestones provides the following command-line tools:

1. The state space of M is computed by the command `computeReachable`. It has two modes. During the search, the first mode ensures that all the reached states are expanded only once by keeping a frontier. The second mode does not keep any frontier, but instead, at each step, it computes the post-image of all the states which has already been reached. It stops when a fixed-point which characterizes the reachable state space is reached.

2. The command `checkMDeadlock` has the same modes as the command `computeReachable`. It verifies that M does not contains any deadlocks.

3. The state space of a reduced version M_R of M is computed by the command `computeReducedReachable`. It applies the PartialExploration algorithm. Its main options concern the properties that M_R preserves (deadlocks, CTL, or LTL properties), and the version of the PartialExploration algorithm which is considered: (1), (2), or (3). (c.f. Section 3.2.1).

4. The command `checkMRDeadlock` verifies that the M_R produced by the `computeReducedReachable` command does not contain any deadlocks. It has the same options as the command `computeReducedReachable`.

5. The command `checkCTL` verifies that M satisfies the CTL properties of sequence C_g. It applies the classical CTL symbolic model checking algorithm presented in Section 2.5.1

6. The command `checkCTLWithPOR` verifies that M satisfies the CTL properties of sequence C_g. Its main option allows us to choose the version of the PartialExploration algorithm which is considered: (1), (2), or (3).

7. The command `checkLTL` verifies that M satisfies the LTL properties of the sequence L_g. For achieving its goal, it applies the

LTL symbolic model checking of [CGH97] which is presented in Section 4.1.3.

8. The command `checkLTLWithPOR` verifies that M satisfies the LTL properties of the sequence L_g. It performs the evalCTLX algorithms presented in Chapter 4. Its main option allows us to choose the version of the PartialExploration algorithm which is considered: (1), (2), or (3).

9. The command `boundedCheckLTL` looks for a trace of some length k of M which violates the LTL properties of the sequence L_g. For achieving its goal, it applies the classical LTL bounded model checking algorithm of a. Biere et al [BCC$^+$03] which is presented in Section 5.1.1.

10. The command `boundedCheckLTLWithPOR` looks for a trace of some length k of M which violates the LTL properties of the sequence L_g. For achieving its goal, it applies our LTL bounded model checking algorithm BPE (c.f. Chapter 5).

Listing 6.4 shows the output of the reachable command on our running example. We make the printed information as clear as possible.

6.5 Exporting to Promela and NuSMV

Milestones can translate its models into NuSMV [CCGR99] (command `translateIntoNuSMV`) or into Promela, the language used by Spin [Hol97] (command `translateIntoPromela`). It is important to compare the new approaches presented in this thesis to state-of-the-art implementations of POR and symbolic model checkers. Spin and NuSMV are the most widely used model checkers. NuSMV is a symbolic model checker. It can verify for example both CTL and LTL properties. As for Spin, it is an explicit model checker featuring POR methods which can check LTL properties among other things.

6.5.1 NuSMV translation

Milestones is able to translate a Milestones program T into an equivalent NuSMV system. The two systems are equivalent because they define the

Listing 6.4: Verbose output of the `ComputeReachable` command

```
1  Milestones started
2
3  ./thesis/running_ex.mil system parsed
4  flattened system created
5
6  # of available nodes = 2000000
7  cache size = 80000 (# of nodes)
8  transition relation creation takes 9 milliseconds
9  transition relation 79 (#nodes)
10 bdd representation created
11
12 Reachable computation started
13 LFP iter: 0,  reach size: 7,  #states: 1.0, #nodes used: 341
14 LFP iter: 1,  reach size: 12, #states: 3.0, #nodes used: 372
15 LFP iter: 2,  reach size: 13, #states: 5.0, #nodes used: 404
16 LFP iter: 3,  reach size: 12, #states: 6.0, #nodes used: 428
17 LFP iter: 4,  reach size: 17, #states: 7.0, #nodes used: 441
18 LFP iter: 5,  reach size: 17, #states: 7.0, #nodes used: 443
19 # LFP iterations = 6
20 Reachable computation ended after 19 milliseconds
21
22 The full state space contains 7 states
```

same transition system. Before presenting the translation mechanism, we recall that T is transformed into a flattened model. Then, a BEXPR expression H which encodes the initial states induced by T is constructed. Moreover, a BEXPR expression F which encodes the transition relation induced by T is produced.

The NuSMV language defines the "INIT Boolean_expression" statement, and the "TRANS Boolean_expression" statement. They are respectively used to define the initial states and the relation transition of a system. Actually, the Boolean expressions which are associated to the INIT statement and the TRANS statement correspond to H and F. Roughly speaking, the Milestones BEXPR is a sub-language of the NuSMV Boolean expressions. Thus,, it is easy to translate the Milestones BEXPR into a NuSMV Boolean expression. Finally, the resulting NuSMV system is composed of four distinct parts:

1. The IVAR part which declares the actions of the system.
2. The VAR part which declares the variables of the system.
3. The INIT part which encodes the initial states of the system with a Boolean expression.
4. The TRANS part which encodes de transition relation of the system with a Boolean expression. This Boolean expression contains two versions of each variable x. The source state is represented by x, and the target state is represented by next(x) .

Listing 6.5 shows the NuSMV system generated from our running example.

Because BDD variable ordering can considerably influence the size of BDDs (c.f. Section 2.5.4), and thus the performance of the algorithm, a file which represents this order is generated. This file can be used by NuSMV to construct its BDDs. Together, these allow a close and fair comparison between Milestones and NuSMV.

6.5.2 Spin translation

Milestones can translate a program T into a Promela system. Listing 6.2 shows a part of the translation of our running example. The Promela model can be decomposed into three parts: the declarations of the

Listing 6.5: Form of the Generated NuSMV File

```
1  MODULE main
2
3  IVAR
4    action : {Gpp, Gmm, xExMy, Xpp, Zpp, Ypp, Ymm};
5
6  VAR
7    g : 0..1;
8    x : 0..1;
9    y : 0..1;
10   z : 0..1;
11   a_pc : 0..4;
12   b_pc : 0..2;
13
14 INIT g = 0 & x = 0 & y = 0 & z = 0 & a_pc = 0 & b_pc = 0;
15
16 TRANS
17   action = Xpp & a_pc = 0 &
18   next(g) = g & next(a_pc) = 1 & next(x) = 1 & next(z) = z &
19   next(b_pc) = b_pc & next(y) = y
20
21 | action = Zpp & a_pc = 1 &
22   next(g) = g & next(a_pc) = 2 &  next(x) = x & next(z) = 1 &
23   next(b_pc) = b_pc & next(y) = y
24
25 | action = Ypp & b_pc = 0 &
26   next(g) = g & next(a_pc) = a_pc &  next(x) = x & next(z) = z &
27   next(b_pc) = 1 & next(y) = 1
28
29 | action = Ymm & b_pc = 1 & g = 0 &
30   next(g) = g & next(a_pc) = a_pc &  next(x) = x & next(z) = z &
31   next(b_pc) = 0 & next(y) = 0
32
33 | action = Gpp & a_pc = 2 & b_pc = 1 & g = 0 &
34   next(g) = 1 & next(a_pc) = a_pc &  next(x) = x & next(z) = z &
35   next(b_pc) = b_pc & next(y) = y
36
37 | action = Gmm & a_pc = 2 & b_pc = 1 & g = 1 &
38   next(g) = 0 & next(a_pc) = a_pc &  next(x) = x & next(z) = z &
39   next(b_pc) = b_pc & next(y) = y
40
41 | action = xExMy & a_pc = 2 & b_pc = 1 &
42   next(g) = g & next(a_pc) = a_pc &  next(x) = x & next(z) = z &
43   next(b_pc) = b_pc & next(y) = y
```

variables (line 12), the initialization (line 27), and the transition relation (line 36).

Because Milestones allows us to perform *parallel assignments*, while Promela does not, for each Milestones variable v of $\mathrm{var}(T)$, two Promela variables v and v' are declared. Each new variable v' is initialized to 0. Each original variable v is either initialized to 0 or to another value default v when its default value is provided. The original variables v and their copies simulate parallel assignments. Given the sequence of variable $var(T) = (v_1, \ldots v_v)$, the parallel assignments $\mathtt{u_0} := \mathtt{e_0}, \ldots, \mathtt{u_u} := \mathtt{e_u}$ can be simulated as follows $\mathtt{u'_0} := \mathtt{e_0}, \ldots, \mathtt{u'_u} := \mathtt{e_u}, \mathtt{u_0} := \mathtt{u'_0}, \ldots, \mathtt{u'_u} := \mathtt{u_u}, \mathtt{u'_0} := \mathtt{0}, \ldots, \mathtt{u'_u} := \mathtt{0}$. The variables of V' are initialized and always reset to 0, to avoid producing spurious additional states.

Milestones allows us do declare more than one initial state, while Promela does not. Given a Milestones system, we distinguish two cases:

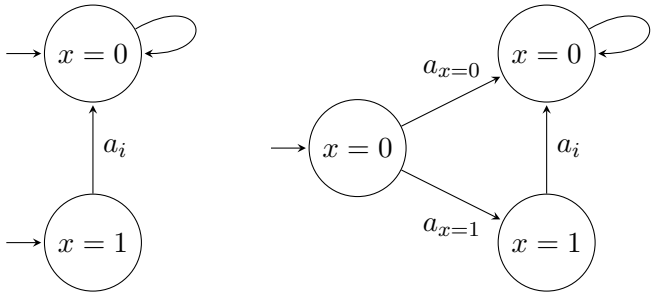
1. If the induced transition system M contains only one initial state, the Promela program produced by Milestones induces a transition system which is isomorphic to M.

2. If the induced transition system M contains more than one initial state, the Promela program generates a transition system M_P which contains one more state than M, and so M and M_P are not isomorphic. As a consequence, M and M_P might not respect the same LTL properties. Actually, the generated Promela program contains an additional initial state i_0^P, and the Promela transition relation contains a transition $i_0^P \longrightarrow i$ for each initial state i of the Milestones program. As a consequence, that the reachable state space of the Promela system contains one more state than the one of the Milestones program. Another consequence is that the generated transition system might not be visible-bisimilar of the original transition system. For instance, the two systems which are presented at Figure 6.3 are not visible-bisimilar. Indeed, the Milestones system does not satisfy the property "$\mathsf{E}[x = 0\,\mathsf{U}\,x = 1]$" while the Promela model does.

Before presenting the translation of the relation, we recall that a program T is composed of a set of processes P_g. For each action $a \in \mathrm{act}(T)$ a subset of processes $P'_g \subseteq P_g$ are synchronized on a. Each

Listing 6.2: Form of the Generated Promela File

```
1  bool TURN_POR_Off = true;
2
3  inline ___copy___() {
4           g = g_0;
5           a[0].pc = a[0].pc_0;
6           a[0].x = a[0].x_0;
7           a[0].z = a[0].z_0;
8           b[0].pc = b[0].pc_0;
9           b[0].y = b[0].y_0;
10 }
11 ...
12 local unsigned g_0 : 1 = 0;
13 local unsigned g : 1 =  0;
14
15 typedef A___TYPE {
16           unsigned pc : 2 = 0;
17           unsigned pc_0 : 2 =  0;
18           unsigned x : 1 = 0;
19           unsigned x_0 : 1 =  0;
20           unsigned z : 1 = 0;
21           unsigned z_0 : 1 =  0;
22 }
23 local A___TYPE a[1];
24 ...
25
26 init {
27   atomic {
28     g = 0;
29     a[0].pc = 0;
30     a[0].x = 0;
31     a[0].z = 0;
32     b[0].pc = 0;
33     b[0].y = 0;
34   }
35
36   do
37   ...
38   /****Gpp****/
39   :: atomic {
40   a[0].pc == 2 & b[0].pc == 1 && g == 0 && TURN_POR_Off ->
41          if :: b[0].pc == 1 -> fi;
42          if :: a[0].pc == 2 -> fi;
43          if :: a[0].pc == 2 & b[0].pc == 1 && g == 0 ->
44            g_0 = 1
45          fi;
46          ___copy___()
47          ___reset___()
48     }
49     ...
50   od;
51 }
```



(a) Milestones System (b) Generated Promela System

Figure 6.3: Two non Visible-bisimilar Systems

process $P_i \in P'_g$ defines a set of guarded commands G_i. A subset $G'_i = \{\text{`[action][pc]cond : assign'} \in G_i \mid \texttt{action} = a\}$ of G_i contains guarded commands which are synchronized on a. The transition relation is constructed based from those subsets G_i's. Actually, for each a of act(T), a guarded command which has the following form is produced:

```
1  ::  atomic { COND_a ->
2    /* Translation of the guarded commands of process P_i          */
3    /* which are synchronized on a, and so which belong to        */
4    /* G'_i = {[a][pc_0]cond_0 : assign_0, [a][pc_1]cond_1 : assign_1, ...}  */
5    if
6    :: pc_i = pc_0 & cond_0 -> assign_0
7    :: pc_i = pc_1 & cond_1 -> assign_1
8    ...
9    fi;
10
11   /* Translation of the guarded commands of process P_j          */
12   if ... fi;
13
14   /*If a is a global action*/
15   if ... fi;
16 }
```

This guarded command is atomic in order to simulate the parallel assignment which is performed by the transitions labeled a. The guard COND_a is true if and only if all the processes which are synchronized on a are able to perform a. Then, for each process P_i which is synchronized on a, an "if ... fi" statement is created. It naturally encodes the non-deterministic actions performed by P_i when a is fired. Actually, those actions are defined by means of guarded command in both Milestones

and Spin. Hence, the guarded commands of P_i which are labeled by a, are translated into guarded commands of Spin. If a is a global action, a final "if ... fi" statement is needed to simulate the non-deterministic assignment of the global variables. Finally, all the guarded commands which are generated from all the F_a formulae are put into a "do ... od" Promela loop to simulate the whole transition relation.

To summarize, given a Milestones program which induces a transition system $M = (A, AP, S, R, I, L)$, Milestones encodes M into a Promela program which induces a system $M_P = (S_P, R_P, i_p, L)$. For each $s \in S$, there is a corresponding $s_p \in S_P$. For each action $a \in A$, $s \xrightarrow{a}_R s'$ if and only if $s_p \longrightarrow_{R_P} s'_p$. Moreover, if $I = \{i\}$ then i corresponds to i_p. Therefore, we claim that the generated Promela model defines almost the same state machine as the Milestones system, except for one more state which is necessary to correctly initialize the variables. As long as the two systems are visible-bisimilar, or as long as there in only one initial state, we have good support for fair comparison between Spin and Milestones.

6.6 Conclusion

In this chapter, we introduced the Milestones model checker which combines POR methods and symbolic model checking approaches. It implements all the algorithms presented in Chapter 3, 4, and 5, and their different variants, as well as other existing approaches for comparison. It provides us with a suitable environment to experiment and compare the algorithms presented in the previous chapters. Milestones defines its own simple language to model real systems into Milestones programs. The language is simple in the sense that it is low level and it has a simple to understand semantics. On such systems, the following features are provided:

- Computation (with or without POR) of the reachable state space by applying set-based approaches.
- Deadlock detection (with or without POR) by applying set-based approaches.
- CTL verification (with or without POR) by applying set-based approaches.

- LTL verification (with or without POR) by applying set-based approaches or BMC approaches.
- Translation into a NuSMV systems or a Spin systems.

Milestones was developed from scratch in Scala. It uses the Buddy BDD library and the Yices SMT solver. Doing so gives us the ability to make all design and development choices. Another approach could have been to extend another existing tool. In this case, all these design and development choices would have been limited. On the other hand, we could have benefited of all the already developed features. As an example, although Milestones is able to check temporal properties, it needs to be extended by adding generation of counter-examples for failed properties.

Chapter 7

Experimental Evaluation

To assess the effectiveness and the scalability of the algorithms evalCTLX of Chapter 3, evalLTLX of Chapter 4, and BPE of Chapter 5 as implemented in Milestones, we applied them to four examples. For each of the examples, we start by giving some statistics on its reachable state space. Then, we verify on it a number of CTL or LTL properties with one or more of the different methods developed in this thesis, as well as comparable methods from other authors. All the tests have been run on a 2,16 GHz Intel Core 2 Duo with 2 GB of RAM memory. Here is a brief description of the presented examples:

- the *TT system* which models a turntable,
- the *EL system* which manages the elevators of a twelve-floor building,
- the *CP system* which models a small cash machine network,
- the *PC system* which models a simple producer consumer system.

Table 7.1 gives, for each example, the number of processes which are involved and the size of its reachable state space of this system. When a number n is provided, it means that the system is scalable. In that case, the number of processes which compose the system depends on n.

The remainder of this section is structured as follows. In Section 7.1, we clarify the conventions used through this chapter. In Section 7.2, we verify a variant of the turntable system which is described in [BTW+05,

Table 7.1: Summary Of the four presented examples

System	n	# of processes	# states
TT	40	$n + 3$	$\approx 10 \cdot 10^{37}$
EL	—	8	$\approx 10 \cdot 10^{22}$
CP	—	6	$\approx 10 \cdot 10^{10}$
PC	20	$2\ n$	$\approx 10 \cdot 10^{39}$

Mat06]. In Section 7.3, we check a variant of the elevator system which is described in [RAO92]. In Section 7.4, we test a cash point system which is described in [DOP00]. In Section 7.5, we model and verify a producer-consumer system which is modeled from scratch. Section 7.6 provides conclusions.

7.1 Naming Conventions

For each of the four systems, we compute the reachable state space with three different methods:

- The *BFS* method performs a symbolic BFS. At each step, it expands a frontier which contains the states which have not yet been processed.
- The *PE* method computes the reachable state space with the PartialExploration approach. It produces a reduced state space which preserves CTL^*_X properties.
- The *PE(dead)* is a variant of PartialExploration approach. It computes the reachable state of a reduced state space which preserves deadlocks. This approach is similar to the one of the PartialExploration algorithm, but at each step its frontier contains only the states which were not explored before. Therefore, it might violate the cycling POR condition C_3 (c.f. Section 2.4.1). It takes as input a transition system with a safe and linear process model. It computes the reachable state of a reduced state space which preserves deadlocks. By doing so, we hope a bigger reduction that one obtained with the normal PartialExploration approach.

We mainly perform those computations for two reasons. On the one hand, it gives us some precious information about the system, e.g. size of the system, or whether the system contains any deadlocks. On the other hand, we recall that the evalCTLX algorithm is composed of two phases. The first one performs forward model checking, then the second one performs backward model checking. Hence, the worst-case time complexity of the evalCTLX algorithm is the addition of the worst-case time complexity of the first and the second phase. Concerning the worst-case time complexity of the first phase, it is easy to see that it is equivalent to visit the whole reduced state space. Therefore, computing the whole reduced state correspond to this case.

Here, we enumerate the information we have collected:

- the computation time in seconds (*time*)
- the number of states (*#states*)
- the number of post-image computations (*#post*)
- the number of BDD nodes needed to represent the reachable state space (*reach #nodes*)
- the average number of BDD nodes needed to represent the successive frontiers (*avg #nodes*)
- the maximum number of BDD nodes needed to represent the successive frontiers (*max #nodes*)
- the number of BDD nodes needed to represent the whole relation transition R ($\#R$).
- the maximum number of BDD nodes needed to represent a relation transition R_i of the generated process model (max $\#R_i$).

The properties are verified with either Milestones (c.f. Chapter 6), NuSMV 2.4.3 [CCGR99] specifying the two options -dcx and -df, or Spin 5.2.2 [Hol97]. For each property, if nothing is mentioned, it is verified with Milestones. We use one or more of the following BDD-based methods to verify the CTL properties:

- The evalCTLX method corresponds to our evalCTLX algorithm of Section 3.2.1).

- The BWD method corresponds to the classical backward CTL model checking algorithm of [CGP99] as implemented in NuSMV but executed in Milestones.
- The FWD method corresponds to the forward CTL model checking algorithm of Iwashita et al [INH96] executed in Milestones.
- NuSMV is equivalent to BWD but executed in NuSMV.

We use one or more of the following BDD-based methods to verify the LTL properties:

- The evalLTLX method corresponds to our evalLTLX algorithm of Section 4.2.
- The evalLTLX(BWD) method corresponds to a variant of the evalLTLX algorithm which uses the backward CTL model checking to perform the fair cycle detection.
- The BWD method corresponds to the LTL model checking algorithm of [CGH97] which looks for fair cycles with the classical backward CTL model checking algorithm in Milestones.
- The FWD method corresponds to the LTL model checking algorithm of [CGH97] but the fair cycles are checked with the forward CTL model checking algorithm of [INH96] in Milestones.
- NuSMV is equivalent to BWD but executed in NuSMV.
- SPIN means that the LTL properties are verified with Spin which applies POR principles.

We use one or more of the following bounded model checking methods to verify the LTL properties:

- The BMC+BPE method corresponds to the BoundedPartialExploration algorithm of Chapter 5.
- The BMC method corresponds to the bounded model checking algorithm of [BCC$^+$03] in Milestones.
- BMC+NuSMV is equivalent to BMC but executed in NuSMV.

7.2 The Turntable System

This section applies model checking verification to an adaptation of the turntable system which is described in [BTW+05, Mat06]. We call this system the TT system. We start by computing its reachable state space and giving the statistics which are described in the previous section. Then, we check some CTL properties with Milestones and NuSMV. Afterwards, we check some LTL properties with Milestones, NuSMV, and Spin. Note that we have tried to apply bounded model checking on this example, but it failed to find any errors. Actually, the properties we checked require a long trace to be discovered. Due to performance, BMC are not able to deal which such long traces on the TT system.

The TT system consists of a round turntable which has $n + 3$ slots: an input place, an output place, n drills and a testing device. Each of these slots can hold a single product. The turntable transports products between the input position, the drills, the testing device and the output position. The drills drill holes in the products. After being drilled, the products are delivered to the tester, where the depth of the holes is measured, there is a possibility for the drilling to have gone wrong. The original model was described in LOTOS, a formal specification technique based on process algebras [ISO88]. We manually translated the LOTOS system into a Milestones program. Moreover, it had only one drill; we extended it to represent an arbitrary number of drills. Figure 7.1 graphically represents the turntable system.

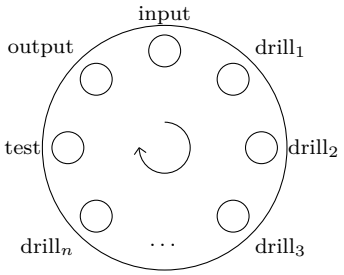


Figure 7.1: The Turntable System

The Milestones program is 442 lines long. Its is composed of $n + 4$

processes which are used to model the slots and the turntable itself. We define five process types, one for each type of component: input place, output place, drill, testing device, and turntable. They communicate together via global variables, and they synchronize with each other which global actions. Statistics about the state space computation of the TT system are presented in Table 7.2. We discuss the instantiation of the system which contains 40 drills. First of all, we observe that the PE approach and the PE(dead) approach have similar performance. This is surprising because the PE(dead) approach uses fewer restrictive POR criteria and was therefore expected to perform better. While the BFS approach takes around 34 minutes to visit around 10^{37} states, the PE approach needs about 24 seconds to visit more or less $1{,}6 \cdot 10^{08}$ states. The BFS approach approximately performs 5 times fewer post-image computations, but they involve BDDs which are in average 7 times bigger that the ones which are manipulated by the PE approach. In the end, the reduced state space is encoded with a BDD 7 times larger than the state space in its entirety. This suggests that, at least for this model, it would be counter productive to firstly compute a reduced state space, and then verify a property within this reduced state space.

7.2.1 CTL Verification

We have verified thirteen properties from [Mat06] on the TT system. [Mat06] expresses these properties in regular alternation-free μ-calculus [MS03], and verifies them with the CADP toolset: Evaluator [FGK$^+$96, GJM$^+$97]. We manually translate the regular alternation-free μ-calculus formulæ properties into CTL. Afterwards, we checked that the TT system verifies those CTL properties.

For instance, property $p6$ states that if a piece is well drilled, no alarm will be raised before the piece is removed. Property $p11$ states that each piece will be removed from the turntable after it is tested, $p6$

Table 7.2: Statistics for the reachable state space computation of the TT system . "–" indicates that the computation did not end within 5 hours.

# drills	time (sec)			#states			#post			max R_i
	BFS	**PE(dead)**	**PE**	BFS	**PE(dead)**	**PE**	BFS	**PE(dead)**	**PE**	
1	0,938	**0,771**	**0,530**	18 168	**5 572**	**5 572**	120	**722**	**722**	29
2	0,891	**0,755**	**0,571**	109 644	**31 792**	**31 792**	174	**999**	**999**	29
3	1,076	**0,845**	**0,674**	749 976	**92 916**	**92 916**	228	**1 268**	**1 268**	29
4	1,287	**1,095**	**0,776**	5 232 300	**203 200**	**203 200**	282	**1 535**	**1 535**	29
10	5,122	**3,046**	**2,448**	$6{,}15 \cdot 10^{11}$	**2 695 600**	**2 695 600**	606	**3 147**	**3 147**	29
20	51,659	**5,385**	**5,022**	$1{,}74 \cdot 10^{20}$	$\mathbf{2{,}03 \cdot 10^{07}}$	$\mathbf{2{,}03 \cdot 10^{07}}$	1 146	**5 823**	**5 823**	30
40	2 082,468	**44,915**	**24,129**	$1{,}39 \cdot 10^{37}$	$\mathbf{1{,}57 \cdot 10^{08}}$	$\mathbf{1{,}57 \cdot 10^{08}}$	2 226	**11 187**	**11 187**	30
60	—	**43,444**	**37,991**	—	$\mathbf{5{,}25 \cdot 10^{08}}$	$\mathbf{5{,}25 \cdot 10^{08}}$	—	**16 543**	**16 543**	31

# drills	reach #nodes			avg #nodes			max #nodes			R
	BFS	**PE(dead)**	**PE**	BFS	**PE(dead)**	**PE**	BFS	**PE(dead)**	**PE**	
1	131	**196**	**196**	111	**90**	**90**	287	**307**	**307**	1 543
2	274	**478**	**478**	215	**149**	**149**	547	**611**	**611**	1 913
3	428	**872**	**872**	332	**210**	**210**	795	**919**	**919**	2 307
4	582	**1 346**	**1 346**	470	**276**	**276**	1 043	**1 227**	**1 227**	2 719
10	1 506	**4 310**	**4 310**	1 787	**722**	**722**	3 991	**3 075**	**3 075**	5 636
20	3 046	**13 025**	**13 025**	5 888	**1 367**	**1 367**	13 301	**6 145**	**6 145**	12 184
40	6 126	**44 504**	**44 504**	21 284	**2 967**	**2 967**	48 121	**12 305**	**12 305**	31 572
60	—	**75 560**	**75 560**	—	**4 217**	**4 217**	—	**18 475**	**18 475**	59 365

and $p11$ are expressed in CTL as follows.

$$wd \equiv \text{the piece which is in the testing slot is well drilled} \quad (7.1)$$
$$turn \equiv \text{the turntable has just made a turn} \quad (7.2)$$
$$rem \equiv \text{the piece which was in the output slot was just removed} \quad (7.3)$$
$$rea \equiv \text{the alarm is resonating} \quad (7.4)$$
$$test \equiv \text{the piece which in on the test slot is tested} \quad (7.5)$$
$$p6 \equiv \neg \mathsf{EF}\,(wd \wedge \mathsf{E}\,[\neg turn\ \mathsf{U}\ (turn \wedge \mathsf{E}\,[\neg rem\ \mathsf{U}\ rea])]) \quad (7.6)$$
$$p11 \equiv \neg\,(\mathsf{EF}\,(test \wedge \neg \mathsf{EF} rem)) \wedge \neg\,(\mathsf{EF}\,(test \wedge \mathsf{E}\,[\neg rem\ \mathsf{U}\ turn])) \quad (7.7)$$

For all properties, we obtain similar results with NuSMV and the BWD approach of Milestones, and NuSMV tends to be a bit faster than Milestones. For 11 of the 13 properties, the evalCTLX algorithm outperforms the classical backward CTL algorithm. However, for $p1$ and $p2$ which have a similar shape to $p6$, the classical method is approximately 30 times faster than the evalCTLX algorithm. Unfortunately, we do not have an explanation for such a difference. We also notice that, on our Turntable system, the forward approach of [INH96] is less efficient than the classical method, taking exception from the general observation reported in [INH96].

Table 7.3 and Table 7.4 respectively show the times for the verification of properties $p6$ and $p11$. When the turntable has 40 drills, $p6$ property is checked approximately 13 times faster and $p11$ is checked approximately 8 times faster with the PartialExploration method than with the backward method. On the other hand, if there are only a few drills, the backward method runs faster.

7.2.2 LTL Verification

We have verified six properties: four properties that the system satisfies, and two properties which are not fulfilled. We obtained similar curves for the six properties. For instance, the property T_3 states that if in the future there is a piece which is not well drilled, the alarm will necessarily go off. Here is the translation of this property in LTL:

$$\mathsf{G}\ [\mathsf{F}\ \neg wd \implies \mathsf{F}\ rea]. \quad (7.8)$$

Table 7.3: Verification times (in seconds) for properties p6 of the TT model, using NuSMV (NuSMV) and Milestones using standard backward exploration (BWD), FwdUntil (FWD) and evalCTLX (evalCTLX).

# drill	property p6			
	NuSMV	BWD	FWD	**evalCTLX**
1	0,04	0,04	0,02	**0,06**
2	0,06	0,08	0,05	**0,09**
3	0,12	0,13	0,08	**0,12**
4	0,15	0,18	0,11	**0,16**
10	0,63	0,72	1,00	**0,38**
20	4,20	4,86	9,78	**0,87**
30	12,01	14,08	36,04	**1,48**
40	33,17	35,10	93,63	**2,61**

Table 7.4: Verification times (in seconds) for properties p11 of the TT model, using NuSMV (NuSMV) and Milestones using standard backward exploration (BWD), FwdUntil (FWD) and evalCTLX (evalCTLX).

# drill	property p11			
	NuSMV	BWD	FWD	**evalCTLX**
1	0,04	0,04	0,08	**0,10**
2	0,07	0,07	0,16	**0,15**
3	0,09	0,10	0,19	**0,19**
4	0,12	0,13	0,28	**0,26**
10	0,65	0,71	1,13	**0,64**
20	6,22	7,26	10,41	**1,64**
30	18,55	22,65	28,32	**2,81**
40	35,23	40,73	72,46	**4,97**

Figure 7.2 compares the times for the verification of property T_3. We observe that within a limit of 1000 seconds, the various approaches are able to verify a TT system which contains between 6 (NuSMV) and 61 drills (evaLTLX). We also observe that the FWD approach which does not apply any partial order reduction is able to check a system composed on 35 drills, and the evalLTLX(BWD) which applies partial order reduction is only able to check a system composed of 10 drills.

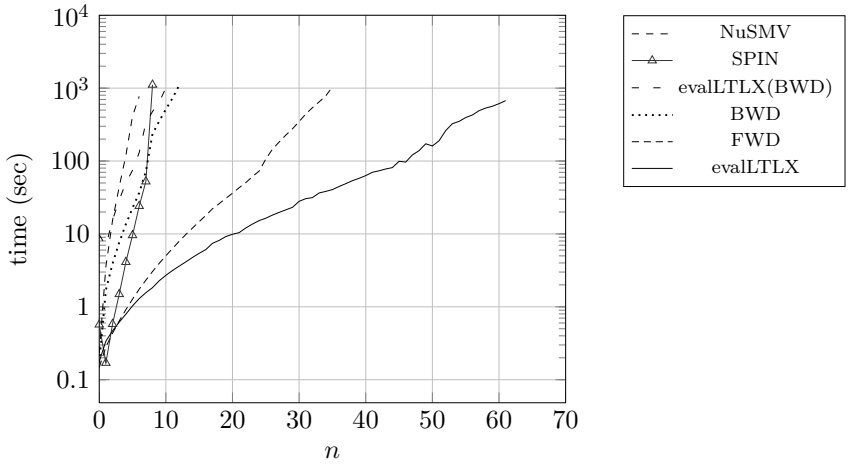


Figure 7.2: Verification times for the Turntable property T_3

7.3 The Elevator System

In this section, we model and verify an elevator system which is inspired from the one presented in [RAO92]. Hereafter, we name this system the EL system. We choose the EL system because its initial states are very broadly unconstrained, yielding $4^{12} \approx 1{,}5 \cdot 10^7$ initial states. We do not model the original system of [RAO92] because it contains real time constraints which are not easily encodable with the Milestones language. Moreover, as opposed to [RAO92], we model the behavior of the users, resulting in very different models overall.

The EL system is in charge of managing a set of elevators which are all located in a unique building which has x floors. In this section, we

consider that x is equal to 12. This system is composed of 8 asynchronous components: three elevators, four users, and one scheduler. The initial positions of the users are not determined: they might be on any floor. To get up or down, a user pushes a button at level L. Afterwards, the scheduler looks for an elevator which will move towards level L. The scheduler chooses the idle elevator E which is the closest to level L. If there is more than one such elevator, one is nondeterministically picked up. On the other hand, if no elevator is idle, the scheduler delays this task until an elevator is free. When E reaches the floor L, the user gets into the elevator E and goes to a floor L' that he wants to reach.

The Milestones program of the EL system consists in a file of 443 lines of code which contains 8 asynchronous components. Table 7.5 shows statistics about the reachable state space of this model. Concerning the PE approach, this table confirms the observations of Table 7.2. As for the PE(dead) method, it performs three times less post-image operations than the PE approach. On the other hand, it executes around 1.5 times more post-image computations than the BFS method but with much smaller BDD encoding the successive frontiers.

Table 7.5: Statistics about the Reachable State Space of the EL System, $\#R = 8507$, max $\#R_i = 293$

	BFS	**PE(dead)**	**PE**
time (s)	912,43	**0,706**	**1,558**
#states	$3{,}19 \cdot 10^{22}$	$\mathbf{6{,}39 \cdot 10^{10}}$	$\mathbf{3{,}17 \cdot 10^{20}}$
#post	171	**291**	**776**
#nodes	3 670	**2 929**	**41 002**
avg #nodes	88 413	**474**	**408**
max #nodes	170 167	**1 494**	**1 494**

We verified the three following similarly shaped properties on this system both in CTL and LTL:

- Whenever a user U_0 pushes the elevator button at level L, an elevator will eventually arrive at level L. This property is not met

by our system. Hence, it contains a starvation problem.

$$\neg \mathsf{EF}\,\mathsf{EG}\, wait_{u0} \qquad \text{(CTL)} \qquad (7.9)$$
$$\neg \mathsf{FG}\, wait_{u0} \qquad \text{(LTL)} \qquad (7.10)$$

- Whenever an elevator e_0 moves forward a level L, it will eventually arrive at level L. This property is valid on the EL system.

$$\neg \mathsf{EF}\,\mathsf{EG}\, dest_{e0} \neq level_{e0} \qquad \text{(CTL)} \qquad (7.11)$$
$$\neg \mathsf{FG}\, dest_{e0} \neq level_{e0} \qquad \text{(LTL)} \qquad (7.12)$$

- It is impossible that at some point, all the elevators remain busy forever. This property is also valid on the EL system.

$$\neg \mathsf{EF}\,\mathsf{EG}\, (\neg free_{e0} \wedge \neg free_{e1} \wedge \neg free_{e2}) \qquad \text{(CTL)} \qquad (7.13)$$
$$\neg \mathsf{FG}\, (\neg free_{e0} \wedge \neg free_{e1} \wedge \neg free_{e2}) \qquad \text{(LTL)} \qquad (7.14)$$

Table 7.6 reports the time for the verification of the previous properties with various approaches. Concerning the verification of the CTL and LTL properties with the BDD-based approaches, we notice that there is a great difference between the running times reported by forward and backward model checking. Concerning the verification of the property (7.10) with bounded model checking, we observe that the BPE approach finds a counter-example faster than the other two approaches, but slower than the evalLTLX one.

7.4 The Cash-Point System

In this section, we show that contrary to what happens with the EL system, in some cases the backward model checking approach behaves much better than the forward model checking approach. Hence, there is no uniform better choice among those two approaches. As far as we know, given a problem, there is no general method to predict which of the backward or the forward approach will perform better on this problem, but this choice could be made on the basis of some heuristics. For instance, imagine a transition system $M = (S, R, i, L)$ having only one initial state, and a CTL property $\mathsf{EF}\, p$ where p is a propositional

Table 7.6: Running time about the Verification of the EL System. "–" indicates that the computation did not end within 5 hours.

	(7.9)	(7.11)	(7.13)
BWD	9 228,45	8 064,23	8741,96
FWD	1 332,04	1 186,40	1 536,24
PE	**2,46**	**2,88**	**1,48**

	(7.10)	(7.12)	(7.14)
BWD	—	—	—
FWD	4 440,23	3 240,79	120,51
evalLTLX	**10,75**	**8,32**	**4,75**
BMC	1 425,22		
BMC+BPE	**63,34**		
BMC+NuSMV	642,23		

formula which is satisfied by a great majority of the states in S. In this case, it seems a good strategy to perform a backward search. Indeed, the forward approach starts its search from the singleton $\{i\}$, while the backward approach starts its search from $\mathcal{L}(p)$ which contains a great majority of the states. Hence, before even starting, the backward approach has already discovered a great majority of the states.

To illustrate our point, we model and verify a system of cash points, hereafter referred as the CP system. It is inspired from the one which was presented by T. Devir et al. in [DOP00]. It is composed of six concurrent components: a database server which manages three bank accounts, four tills, and a fault detector. Each ATM stands idle until a customer wants to do an operation. When this happens, the customer puts its bank card which is related to one of the three accounts inside the ATM. Notice that they might be more than one card per account. Then, the ATM asks for the pin code which is stored on the card. After a successful identification, the ATM contacts the database server to lock the corresponding bank account. Then, the customer can perform one or more of the following operations.

- view the balance of his accounts;

- make a withdrawal of cash;
- ask for a statement of its account to be sent by post;
- ask to the database server to unlock the account and then to return the bank card.

When the customer removes his cards, the ATM stand idle until a next customer arrives. The database server manages all the data about the three accounts, and it is not able to deal with more than two requests at the same time. Besides, it may be unavailable. When this happens, the fault detector is in charge of dealing with the requests of the four ATMs. It simply sends an error message.

We modeled the CP system with the Milestones language. It consists in a file of 358 lines of code. Our implementation contains a deliberate error. It allows for two or more ATMs to access the same account at the same time. This error can be found at a depth equal to 19 in the computation tree of the system. Table 7.7 presents statistics of the reachable state space computation of CP. We observe the same trends as the ones observed on the TT system (c.f. Table 7.2).

Table 7.7: Statistics about the reachable state space of the CP System, $\#R = 5800$, max $\#R_i = 291$

	BFS	**PE(DEAD)**	**PE**
time (sec)	3 178,81	**349,19**	**329,29**
#states	$1{,}81 \cdot 10^{10}$	$\mathbf{4{,}93 \cdot 10^{9}}$	$\mathbf{4{,}93 \cdot 10^{9}}$
#post	99	**1 479**	**1 479**
#nodes	927	**1 225**	**1 225**
max #nodes	543 250	**49 216**	**49 216**
avg #nodes	187 885	**17 829**	**18 027**

We check three properties on the CP system. Those properties are expressed in both CTL and LTL logic.

- The database server never deals with more than two ATMs at the

same time:

$$\neg \mathsf{EF}\ uds > 2 \qquad \text{(CTL)} \qquad (7.15)$$
$$\neg \mathsf{F}\ uds > 2 \qquad \text{(CTL)} \qquad (7.16)$$

- An account is never locked by more than one ATM:

$$\neg \mathsf{EF}\ uc0 > 1 \vee uc1 > 1 \vee uc2 > 1 \qquad \text{(CTL)} \qquad (7.17)$$
$$\neg \mathsf{F}\ uc_0 > 1 \vee uc1 > 1 \vee uc2 > 1 \qquad \text{(LTL)} \qquad (7.18)$$

- When a client does an operation with the ATM 0, he eventually receives an answer:

$$\neg \mathsf{EF}\ (\mathsf{EG}\ op_0 \neq 0) \qquad \text{(CTL)} \qquad (7.19)$$
$$\neg \mathsf{F}\ (\mathsf{G}\ op_0 \neq 0) \qquad \text{(LTL)} \qquad (7.20)$$

Table 7.8 reports the time for the verification of the previous properties with various approaches. The backward methods complete verification much faster than the forward approaches. Besides, the BMC approaches perform their verification faster than the forward approaches. Finally, we observe that the fastest tool and technique is NuMV when it performs BMC without any POR.

7.5 The Producer Consumer System

In this section we analyze more deeply the BPE algorithm:

- We compare the classical BMC method and the BPE method.
- We analyze the influence of the number of times Phase-1 is executed for each process (i.e. the parameter n).

We model a simple producer-consumer system, or PC system. The fact that this system is simple does not mean that its verification is simple as well. Indeed, the resulting transition system is not easily treated by the BMC approach, but it is more easily checked with classical BDD model checking. The PC system is a producer-consumer system where all producers and consumers contribute to the production of every single

Table 7.8: Statistics about the Reachable State Space of the CP System.

	(7.15)	(7.17)	(7.19)
BWD	10.35	10.45	12.87
FWD	1 320,25	1 322,78	1 342,45
PE	**248,41**	**250,14**	**278,89**

	(7.16)	(7.18)	(7.20)
BWD	14,57	6,73	23,34
FWD	—	—	—
evalLTLX	**987,35**	**835,73**	**901,96**
BMC		2,9	
BMC+BPE		0,1	
BMC+NuSMV		0,01	

item. The model is composed of $2m$ processes: m producers and m consumers. The producers and consumers communicate together via a bounded buffer which can contain at most x items. In this section, we consider that x is equal to 6. This buffer is modeled by a global variable which represents the number of items which are produced but not consumed yet. We now describe the producers and the consumers.

- Each producer works locally on a piece p. To that end, the producers must share tools which are represented by global variables. When a producer is done, it waits until all producers terminate their task. When it is the case and when there is a place to the buffer, p is added to the buffer. Afterwards, the producers start processing the next piece.

- When the buffer contains an element, consumers remove p from the buffer, they work on it locally. When all the consumers have finished their local work, a new cycle will start as soon as another piece can be removed from the buffer.

The Milestones program consists in a file of 442 lines of code. It contains m consumer processes and m producer processes. Statistics about its

state space computation are described in Table 7.9. Again, we observe the same phenomena as the ones on the TT system (c.f. Table 7.2) and on the CP system (c.f. Table 7.7)

Five LTL properties, not satisfied by the CP system, have been analyzed on this system. We describe here two of those properties:

- P_1 states that the bounded buffer is always empty.

$$\mathsf{G}\, buffer = 0 \tag{7.21}$$

- P_2 states that in all cases the buffer will eventually contain more than one piece.

$$\mathsf{GF}\, buffer > 1 \tag{7.22}$$

Table 7.10 and Table 7.11 compare the classical BMC method and the BPE method when applied to P_1 and P_2. Notice that BMC proceeds by increasing depth k until an error is found (c.f. iterative deepening). Classical BMC quickly runs out of resources whereas our method can treat much larger models in a few minutes. In regard to the verification time, we notice that our method significantly outperforms the BMC method for this example. We also notice that BPE traces are 3,4 to 6,75 times longer. This difference can come both from the addition of the idle transitions, and the considered paths themselves: contrary to BMC, our method does not consider all possible interleavings, thus it does not guarantee finding the shortest error traces.

Table 7.12 analyses the influence of the number of times Phase-1 is executed for each process (i.e. parameter n). We notice that for a given number of producers and consumers, n influences in a non-monotonic way the length of the error execution path, the verification time as well as the memory used during the verification. n influences the two aspects of the transformation of the model. On the one hand, the transition system is more reduced as n is increased due to more partial-order reduction. On the other hand, the number of added idle transitions is also influenced by this parameter. When n is increased, the number of Phase-1/Phase-2 cycles on the discovered error path tends towards the minimum number of unsafe transitions which participate to the violation of the property. We notice that each time the number of Phase-1/Phase-2 cycles decreases

Table 7.9: Statistics for the reachable state space computation of the PC system . "–" indicates that the computation did not end within 5 hours.

# drills	time (sec)			#states			#post		max R_i
	BFS	**PE(dead)**	**PE**	BFS	**PE(dead)**	**PE**	BFS	**PE(dead)**	**PE**	
1	0,525	**0,374**	**0,257**	1 059	**459**	**459**	112	**448**	**448**	35
2	0,826	**0,487**	**0,572**	51 859	**1 211**	**1 211**	192	**909**	**909**	35
3	1,313	**0,902**	**1,150**	3 807 747	**3 682**	**3 682**	272	**1 368**	**1 368**	36
4	2,859	**1,022**	**0,753**	$3{,}03 \cdot 10^{8}$	**27 871**	**27 871**	352	**1 832**	**1 832**	36
10	106,695	**2,399**	**2,555**	$8{,}51 \cdot 10^{19}$	$\mathbf{1{,}05 \cdot 10^{06}}$	$\mathbf{1{,}05 \cdot 10^{06}}$	832	**4 586**	**4 586**	37
20	2 201,940	**5,624**	**4,248**	$1{,}03 \cdot 10^{39}$	$\mathbf{5{,}48 \cdot 10^{07}}$	$\mathbf{5{,}48 \cdot 10^{07}}$	1 632	**9 176**	**9 176**	38

# drills	reach #nodes			avg #nodes			max #nodes			R
	BFS	**PE(dead)**	**PE**	BFS	**PE(dead)**	**PE**	BFS	**PE(dead)**	**PE**	
1	114	**152**	**152**	76	**44**	**44**	115	**70**	**70**	532
2	171	**252**	**252**	272	**75**	**75**	411	**117**	**117**	950
3	230	**386**	**386**	620	**106**	**106**	975	**165**	**165**	1 467
4	289	**524**	**524**	1 120	**150**	**150**	1 779	**216**	**216**	2 077
10	643	**1 894**	**1 894**	7 323	**359**	**359**	11 830	**527**	**527**	7 759
20	1 233	**5 116**	**5 116**	29 872	**726**	**726**	48 123	**1 047**	**1 047**	24 910

Table 7.10: Statistics of property P_1 of the producer-consumer model using BMC approach and BPE approach with $n = 8$. m is the number of producers (resp. consumers), k is the smallest bound for which an error is found, time is the verification time (in seconds), mem is the memory used by Yices when the bound equals k (in Megabyte), and # cycles is the number of cycles: Phase-1/Phase-2. — indicates that the computation did not end within 8 hours.

	BMC property P_1			BMC+BPE property P_1			
m	k	time (sec)	mem	**k**	**# cycles**	**time (sec)**	**mem**
1	10	10	29	**34**	**2**	**7**	**30**
2	18	44	41	**66**	**2**	**8**	**49**
3	26	11,679	65	**98**	**2**	**16**	**85**
4	—	—	—	**130**	**2**	**31**	**122**
5	—	—	—	**162**	**2**	**43**	**169**
6	—	—	—	**194**	**2**	**57**	**224**
7	—	—	—	**226**	**2**	**77**	**288**

by one (c. f. $n = 4$), the cpu time and the memory needed reach a local minimum. Then the cpu time and the memory used increase until the number of cycles decreases again.

Figures 7.3 shows that the producer-consumer system is a difficult problem to tackle with BMC, but it is more easily verified with the evalLTLX algorithm. Actually, with m = 7, the BPE algorithm takes approximately 68 minutes to find a counter-example of length 1,017, while the evalLTLX takes only 314 milliseconds to show the violation.

Table 7.11: Statistics of property P_2 of the producer-consumer model using BMC approach and BPE approach with $n = 8$. m is the number of producers (resp. consumers), k is the smallest bound for which an error is found, time is the verification time (in seconds), mem is the memory used by Yices when the bound equals k (in Megabyte), and # cycles is the number of cycles: Phase-1/Phase-2. — indicates that the computation did not end within 8 hours.

	BMC property P_2			BMC+BPE property P_2			
m	k	time	mem	**k**	**# cycles**	**time (sec)**	**mem**
1	26	73	33	**153**	**9**	**122**	**96**
2	44	29,898	131	**297**	**9**	**211**	**224**
3	—	—	—	**441**	**9**	**401**	**363**
4	—	—	—	**585**	**9**	**1,238**	**680**
5	—	—	—	**729**	**9**	**1,338**	**983**
6	—	—	—	**873**	**9**	**1,926**	**1,438**
7	—	—	—	**1,017**	**9**	**4,135**	**1,618**

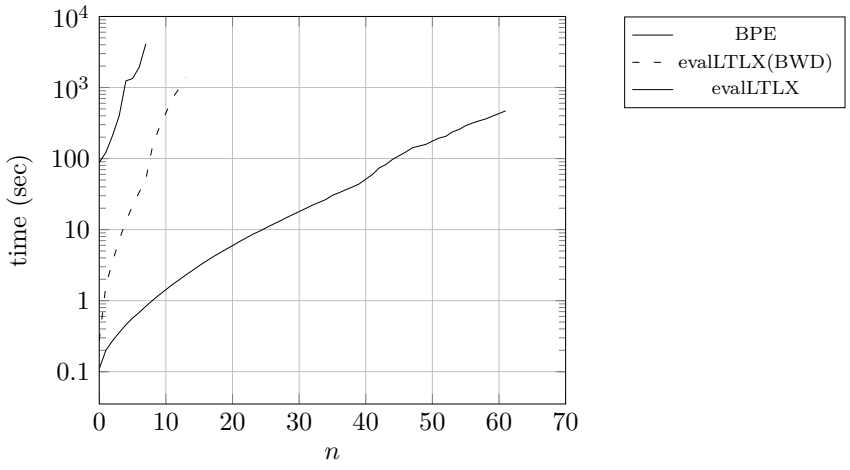


Figure 7.3: Verification times for the Producer-Consumer property P2

Table 7.12: Influence of parameter n when the number of producers (resp. consumers) equals 2. k is the smaller bound for which an error is found, # cycles is the number of cycles: Phase-1/ Phase-2, TIME is the verification time (in seconds), and MEM is the memory used by Yices when the bound equals k (in Megabytes).

	property P_1				property P_2			
n	k	# cycles	time	mem	k	# cycles	time	mem
0	18	18	44	41	44	44	29,898	131
1	35	7	12	41	95	19	855	159
2	45	5	11	40	135	15	235	167
3	39	4	10	47	169	13	305	194
4	51	3	8	47	187	11	217	192
5	63	3	10	50	231	11	375	308
6	75	3	12	57	275	11	381	240
7	87	3	13	58	319	11	583	318
8	66	2	8	49	297	9	211	224
9	74	2	9	57	333	9	240	295

7.6 Conclusion

In this chapter, we evaluated the new approaches presented in this thesis on four models. We modeled and verified a scalable turntable system, an elevator system, a cash point system, and finally a scalable producer-consumer system. For each of these systems, we computed the whole state space and two different reduced state spaces. Then, we checked whether those systems verified various CTL_X or LTL_X properties.

The reachable state space analysis allows us to make the following observations:

- Generally, the BDDs which represent the successive frontier are much smaller when partial order is applied.
- On the contrary, the BDDs which encode the reduced state space are much bigger when partial order is applied.
- The BDDs which encode the transition relations R_i of a process model are much smaller that the ones which represents the whole transition relation R. Generally, they are only composed of a few hundred nodes.
- Surprisingly, in three of the four examples, the PE(dead) approach and the PE approach produce identical BDDs.

The previous observations confirm that combining POR and BDD-based model checking achieves an improvement in regards to classical BDD-based model checking when one wants to check reachability properties on asynchronous transition system. Besides, those observations suggest that firstly computing a reduced state space and secondly performing the verification within this reduced state space can decrease the performance because the BDDs which encode the reduced state space are bigger that the ones which encode de whole state space.

The elevator system and the cash point system show that forward, backward model checking, and BMC are complementary, in the sense that given a system M, one may quickly verify M, while the others might fail to verify it. As far as we know, there is no general method that predicts which of the three methods is the most suitable to check M.

This section shows that the methods developed through this thesis are not applicable on all asynchronous systems. However, for most of

the systems which have been verified in this chapter, they achieve an improvement in comparison to classical model checking algorithms.

Chapter 8

Related Work

In this Chapter, we review some approaches that are related to the techniques which were introduced in the previous parts of this thesis. The first three approaches are related to ours because they combine partial-order reduction (POR) and symbolic method in one way or another. Then, we review the PhD thesis of T. Jussila which is devoted to bounded model checking (BMC) of asynchronous systems. It tackles the same problems as ours but in a different way. For instance, he does not try to reduce the original transition system. Then, the saturation method of G. Ciardo is presented [CLS01]. Given a transition system M, it partitions the transition relation R of M according to a different dependency relation than the one exploited in our POR approach. As opposed to POR approaches, it visits the whole reachable state space, but does this in an unusual way based on the partitioned transition relation. Then, we cover a technique which translates the fair-cycle detection problem into a reachability problem. This approach is interesting in the context of symbolic POR because this problem is a serious challenge in the context of a breadth-first search, and thus in set-based model checking.

We re-express those approaches within the same framework and in the same style as the algorithms of previous chapters. In particular, we present the algorithms in terms of transition systems, and not in terms of BDDs or SAT-solvers. Moreover, we remove from the original algorithms some optimizations which are not relevant to understand them. For instance, we remove the utilization of caches. We insist that the changes

we made do not modify the complexity of the algorithms, although the resulting algorithms can look significantly different from their original version.

The rest of this chapter is structured as follows. Section 8.1 presents a symbolic algorithm of R. Alur et al. which applies partial-order reduction to check a class of reachability properties referred as local properties [ABH+97]. Section 8.2 introduces the static partial-order reduction method of R. P. Kurshan which performs POR at compile time [KLM+98, KLY02], so that their resulting transition systems can be verified with symbolic approaches. Section 8.3 introduces a symbolic partial-order algorithm of P. A. Abdulla et al. which checks safety properties either by backward or forward reachability analysis [AJKP98]. Section 8.4 presents the techniques of T. Jussila to perform BMC of asynchronous systems [Jus05]. Section 8.5 introduces the saturation approach of G. Ciardo [CLS01]. Section 8.6 presents the approach of A. Biere et al. which translates the fair-cycle detection problem into a reachability problem [BAS02]. Section 8.7 gives conclusions.

8.1 Symbolic Verification of Local Properties

In [ABH+97], R. Alur et al. adapt the partial-order reduction algorithm "Algorithm 2" of [HGP92]. The authors do not give a name to their algorithm: we call it the AReduce algorithm. It computes a reduced transition system which preserves *local properties*. Intuitively, given a system with several processes, a local property is a safety property which refers only to the global variables and the local variables of only one process. The definition of a local property is quite technical; we refer the reader to [ABH+97, HGP92] for a precise definition. G. Gueta et al introduce explicit verification of similar properties in [GFYS07].

They start from a DFS algorithm of [HGP92] to obtain a modified BFS algorithm. Both only expand a subset ample(s) of enabled transitions at each step. The ample(s) are persistent sets because they respect the two conditions C_0 and C_1 of Section 2.4.1. Moreover, they satisfy the following variant of the cycle condition C_3:

C_{3b} If a state s is not fully expanded, then at least one transition in ample(s) does not lead to an already visited state.

Because checking a local property amounts to a reachability problem, the singleton condition C_4 which are set for preserving branching properties is not required. Moreover, we point out that the invisibility condition C_2 is not mandatory either.

Practically, the notion of *persistent function*, and *history function* are first defined. A persistent function is similar but not identical to the transition relations generated from a process model. It can be seen as a process model composed of a unique element $\{A_0\}$. A process model considers an action a is globally safe or not for all states. Contrarily, a persistent function considers that an action as either safe or not with respect to each state where it is enabled. Another difference comes from the fact that a process model defines more than one ample set per state, while a persistent function defines only one such ample set. The AReduce algorithm aims at automatically computing such a persistent R_p. Besides, R_p has to respect an additional condition which is explained in the next paragraphs.

In general, the algorithms which compute the reachable state space of a graph keep track of the already visited states in a set V. A history function is a mathematical function which models the set of already visited states V. It associates to each state s all the states of the graph which have been already visited before it. Alurs's approach computes a history function which is never used or returned. Its main purpose is to simplify the demonstration of the AReduce correctness.

Definition 8.1 (Persistent Function)**.** *Given a deterministic transition system $M = (S, R, I, L)$, a persistent function is a safe transition relation $R_p \subseteq R$, i.e.* $\text{enabled}(R_p, \{s\})$ *satisfies condition C_1.*

A trivial persistent function is the whole transition relation R itself, but, it does obviously not perform any reduction.

Definition 8.2 (History Function)**.** *Given a deterministic transition system $M = (S, R, I, L)$, a function $H : S \to 2^S$ is a history function if and only if for all states $s, t, u \in S$:*

$$t \in H(s) \wedge u \in H(t) \implies u \in H(s)$$

A persistent function R_p is ample with respect to a history function H if and only if for all states $s \in S$, if $\text{enabled}(R, \{s\}) \neq \text{enabled}(R_p, \{s\})$,

then there is an action $a \in \text{enabled}(R_p, \{s\})$ *and a state* $t \in S$ *such that* $s \xrightarrow{a}_{R_P} t$, $s \in H(t)$, *and* $t \notin H(s)$.

Intuitively, a persistent function R_p is ample with respect to a history function H if at each step it allows one to reach at least one state which has not been visited before. Based on that, the link between persistent functions, history functions, and partial-order reduction is made as follows.

Theorem 8.3 (c.f. [ABH+97])**.** *Given a transition system* $M = (S, R, I, L)$, *a sub-transition system* $M_R = (S, R_p, I, L)$ *of* M *with a persistent function* R_p, *if there exists a history function* H *which is ample with respect to* R_p *then* M *and* M_R *satisfy the same local properties.*

Intuitively, the AReduce algorithm (c.f. Algorithm 8.2) constructs incrementally both a history function H and a persistent function R_p which is ample with respect to H. It uses the ChoosePersistent subproblem (c.f. Algorithm 8.1) which computes at each step a part of the persistent function under construction. The authors of [ABH+97] do not provide any concrete implementation of ChoosePersistent, but instead give some general principles which could be used. Those principles are similar to the one which are used to generate a process model in Milestones (c.f. Section 6.3.4).

We notice that the AReduce algorithm is similar to ImProviso (c.f. Section 3.1.2). We recall that ImProviso is the starting point of the methods which are developed in this thesis. Actually, if a transition system M with a safe linear process model is given as input to the AReduce algorithm and to ImProviso, both will compute a reduced transition system M_R which contains only fully-forming states. Hence, in both cases M and M_R respect the same CTL_X properties, because they are visible-bisimilar. Here are the main differences between the two algorithms when they are used in such context:

Algorithm 8.1: ChoosePersistent

Header: Choose-Persistent(M, *visited*, $frontier$)

Precondition: M is a transition system, *visited* $\subseteq S$, and $frontier \subseteq S$.

Result: $R_p \subseteq frontier \times A \times S$, such that for all $s \in frontier$:

(1) enabled$(R_p, \{s\}) = \emptyset$ if and only if enabled$(R, \{s\}) = \emptyset$, and

(2) if enabled$(R_p, \{s\}) \neq$ enabled$(R, \{s\})$, then enabled$(R_p\{s\})$ satisfies condition C_1, and

(3) if enabled$(R_p, \{s\}) \neq$ enabled$(R, \{s\})$, then post$(R_p, \{s\}) \not\subseteq$ *visited*.

Algorithm 8.2: AReduce

Header: AReduce(M)

Precondition: M is a transition system.

Result: A reduced transition system M_R which preserves local properties, and a history function H.

Loop Invariant:

(1) $M_r = (A, AP, visited \cup frontier, R_R, I, L)$ is a well-defined transition system, and

(2) for all $s \in frontier$, $enabled(R_R, \{s\}) = \emptyset$, and

(3) for all $s \in visited$, $enabled(R_R, \{s\}) = \emptyset$ if and only if $enabled(R, \{s\}) = \emptyset$, and

(4) visited, and frontier are disjoint, and

(5) R_R is a persistent function, and

(6) H is a history function with respect to R_R.

Halting Condition: $frontier$ is empty.

Variant: $(\#S - \#visited)$.

Source Code: c.f. Listing 8.1

Listing 8.1: Implementation of the AReduce algorithm

```
1  AReduce(M = (S, R, I, L)) {
2    local frontier := I
3    local visited := ∅
4    local R_R := ∅
5    local H := ∅
6
7    while(frontier ≠ ∅) {
8      local R_p = ChoosePersistent(M, visited, frontier)
9      local image := post(R_p, frontier)
10
11     R_R := R_R ∪ R_p
12     H := H ∪ {(s, visited ∪ frontier) | s ∈ frontier}
13
14     visited := visited ∪ frontier
15     frontier := image \ visited
16   }
17   return M_R = (A, AP, visited, R_R, I, L), and H
18 }
```

- The authors of [ABH+97] suggest that the AReduce algorithm implementation represents the persistent relation transition as a *monolithic transition relation*. It means that it uses a single BDD to represent the whole persistent transition relation. In contrast, ImProviso partitions the transition relation into local transition relations for each process. Those local transition relations are represented with much smaller BDDs than the monolithic transition relation used by the AReduce algorithm.

- The AReduce algorithm assumes pessimistically that each previous expanded state might close a cycle. By contrast, ImProviso makes a smaller over-approximation of such states because it only needs to consider cycles formed exclusively by safe transitions during Phase-1. Consequently, it looks for possible cycles only with respect to states visited during Phase-1. Our experimental evaluation tends to show that the second approach is more efficient.

8.2 Static Partial-Order Reduction

Explicit-state POR techniques usually perform a modified depth first search. At each state s, a valid subset $ample(s)$ of the transitions enabled in s is explored. To ensure that the verification results on the reduced model hold for the full model, $ample(s)$ has to respect a set of conditions (c.f. Section 2.4.1). In particular, along each cycle on the reduced model at least one state must be fully expanded, i.e. ample(s) = enabled(s).

The symbolic methods amount to a breadth-first search, so that it is harder to detect cycles. In [KLM+98], R. P. Kurshan et al. introduce a *static partial-order reduction* algorithm. They notice that each cycle in the state space is composed of some local cycles. The method performs a static analysis of the checked system so as to discover local cycles. It starts from a high-level description of a transition system M_1. The high-level language which is employed to describe transition systems is similar to that of Milestones model checker, as presented in chapter 6. It is a processes-oriented modeling language based on guarded commands. The algorithm translates the high-level description of M_1 into a high-level description of another transition system M_2. It is isomorphic to a sub-transition system of M_1. Moreover, M_2 is stutter-equivalent to M_1 (c.f. Section 2.3.1), and so respects the same LTL_X properties. The reduced transition system M_2 can be handled with any model checking technique, in particular by symbolic techniques.

The key idea is to define, by performing a static analysis of $M_1 = (A, AP, S, R, I, L)$, a set of *sticky* actions A_{sticky} which contains at least all the visible actions and one action per cycle of M_1. Next, a safe process model $\{A_0, A_1, \ldots A_{m-1}\}$ of M_1 is generated such that none of the A_i's contains an action of A_{sticky}. From that, the guarded commands which define the transition relation of M_2 are generated. Those guarded commands ensure that the set of transitions ample(s) which are enabled from each state s of M_2 respects one of the following condition:

- ample(s) constitute a partial expansion of s because ample(s) contains only transition of a A_i, or
- ample(s) constitute a full expansion of s because ample(s) is equal to enabled(s).

Precisely, $A_{sticky} \subseteq A$ respects the two following conditions:

(1) A_{sticky} contains at least all the visible actions (c.f. Definition 2.4).

(2) For all cycles $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_n} s_0$ in M, $A_{sticky} \cap \{a_0, \ldots, a_n\} \neq \emptyset$.

The key idea comes from the fact that each cycle in the state space is composed of one or more local cycles. Based on that, the A_{sticky} set is computed by analyzing the guarded commands of each individual process. Different strategies are proposed to compute the smallest possible A_{sticky} set. The simplest strategy set up A_{sticky} to A, but in this case no reduction is possible. Another strategy constructs A_{sticky} so that each local cycle contains one action in the A_{sticky} set. More complicated strategies are also presented.

We insist that the new model is expressed in terms of the high-level language, and not in terms of transitions systems. Nevertheless, in fine, we are interested by the resulting transition system, so we characterize that resulting transition system. We refer the interested reader to [BK08, KLM+98] for more details on the translation mechanism.

First of all, a safe process model $\{A_0, A_1, \ldots, A_{m-1}\}$ of M is constructed such that the A_i sets cannot contain any sticky actions. To rephrase, it has to respect that for all sets A_i, and for all actions $a_i \in A_i$, $a_i \notin A_{sticky}$. Then, the state space S is partitioned into $m+1$ sets $\{S_0, S_1, \ldots, S_{m-1}, S_{fully}\}$. The states belonging to the S_{fully} set are fully expanded. Concerning the other sets S_i, all of their states enable exclusively actions of A_i. Precisely, for all states $s \in S$, $s \in S_i$ if and only if the two following conditions hold:

(1) there is an action $a_i \in A_i$, and a state $t \in S$ such that $s \xrightarrow{a_i} t \in R$,

(2) there is no $0 \leq j < i$ such $s_i \in S_j$.

In consequence, for all states $s_{fully} \in S_{fully}$ there is no $0 \leq i < m$ such that $s_{fully} \in S_i$. After partitioning the actions and the states, the transition relation is partitioned into $m+1$ sub-transition realtion $\{R_0, R_1, \ldots, R_{m-1}, R_{fully}\}$ such that the following properties hold:

(1) For all $0 \leq i < m$, $R_i \subseteq R$, and for all transitions $s \xrightarrow{a_i} t \in R$, $s \xrightarrow{a_i} t \in R_i$ if and only if $s \in S_i$ and and $a_i \in A_i$.

(2) For all transitions $s \xrightarrow{a_i} t \in R$, $R_{fully} \subseteq R$, and $s \xrightarrow{a_i} t \in R_{fully}$ if and only if $s \in S_{fully}$.

The static POR approach induces the following reduced transition system: $M_2 = (A, AP, S, (\bigcup_{i=0}^{m-1} R_i) \cup R_{fully}, I, L)$. It is shown in [BK08, KLM$^+$98] that M_2 is stutter-equivalent to M_1 and so respects the same LTL$_X$ properties.

R. P. Kurshan et al. extend the static partial-order reduction in [KLY02]. As in [KLM$^+$98], they start from a high-level description of a transition system M_1 with a safe process model. They translate M_1 into another stutter equivalent transition system. Intuitively, they manipulate the control flow graph of each process. When some conditions are fulfilled, they transform *a chain* of transitions – i.e. a finite acyclic path $s_0 \xrightarrow{a_0} \dots \xrightarrow{a_{n-1}} s_n$ of a control flow graph – into a unique transition $s_0 \xrightarrow{a_{new}} s_n$. In [Hol99], G. Holzmann performs the same kind of reduction which is called *merging local transitions*. The checked conditions of [Hol99] are stronger than those of [KLY02], but easier to check. Hence, more transitions are merged by the approach of [KLY02]. Within the framework of bounded model checking, T. Jussila also performs merging of transitions [Jus05]. His approach is introduced in Section 8.4.

R. P. Kurshan et al. also transform *a hammock* of transitions in a control flow graph – i.e. a set of transitions $\{s \xrightarrow{a_0} t, s \xrightarrow{a_1} t, \dots\}$ sharing the same source node s and the same target node t – into a unique transition $s \xrightarrow{a_{new}} t$. Finally, when some specific conditions hold, they transform the asynchronous sending and reception of a message into a rendez-vous. All those translation mechanisms might produce a much smaller machine than the original one because a number of interleavings are deleted.

The approaches of [KLM$^+$98, KLY02] take as input a high-level description of a transition system M_1. They generate a safe process model, then a stutter-equivalent transition system system M_2. Hence, M_1 and M_2 satisfy exactly the same LTL$_X$ properties. If instead of generating a safe process model, they generate a safe and linear process model, a visible-bisimilar transition system will be generated, so that M_1 and M_2 will satisfy exactly the same CTL$_X$ properties.

This approach differs from ours in that they perform the reduction at compile time. The algorithm defines at compile time for each state s, if s will be fully expanded or not. Moreover, if s is not fully expanded it is decided which ample set will be used on to extend s. Our algorithm makes those choices at run time.

F. Lerda et al. suggest that the ImProviso method is more efficient than the one introduced by R. P. Kurshan et al. [LST03]. However, we think that it would still be interesting to see how both approaches can benefit from each other. Given a high level description D_{M_1} of a transition system M_1, the following approach produces a visible-bisimilar transition system M_2:

1. from D_{M_1}, use the static partial-order reduction approach to generate the A_{sticky} set of actions,

2. as much as possible merge, chains of transitions and hammocks of transition with the approach of [KLY02],

3. automatically generate a safe and linear process model which does not contains any actions of A_{sticky},

4. use either our PartialExploration approach or the the static partial-order reduction approach to perform reduction A mix of both approaches could also be used. For instance, the cycle condition could be checked statically, and the rest of the reduction could be performed dynamically.

8.3 Abdulla's Approach

In [AJKP98], P. A. Abdulla et al. introduce a symbolic partial-order algorithm which checks safety properties either by backward or forward reachability analysis. So as to perform the reduction they use a notion similar to independency of actions which is referred as *commutativity in one direction*. Instead of manipulating sets of states, the proposed algorithm works with sets of sets of states.

The authors start by introducing the *IsReachable* algorithm which takes as input a transition system M and a set of states G (c.f. Specification 8.3). It looks for a reachable state which belongs to G. The algorithm maintains two sets of sets of states. Those sets of states represent the successive frontiers which are reached during the execution of the algorithm. The `Visited` set contains the already expanded frontier. The `NotVisited` set contains the frontiers which remain to explore. At each step, a frontier S_i is chosen from the `NotVisited` set. Given the

set of action A, a post-image T_i per action in A is computed from S_i. Then, each of those post-images T_i is put in the `NoVisited` set if T_i is not visited yet and if no set in `NotVisited` subsumes T_i. The best way to understand the IsReachable algorithm consists in inspecting the loop invariant which respects the following four conditions:

(1) The frontiers in `NotVisited` or in `Visited` contain only reachable states. Besides, to be sure that all states are explored, for all initial states s, `Visited` or `NotVisited` has at least a frontier which contains s.

(2) The frontiers in the `Visited` set have already been explored.

(3) The Boolean variable `found` is true if and only if there exists a frontier in `Visited` which contains a state of G.

(4) The NotVisited set and the Visited set are disjoint. This ensures that the algorithm terminates.

To improve the IsReachable algorithm, the authors show that it is valid to only fire transitions with some defined label at each step. In practice, it amounts to modifying line 8 as follows:[1]

```
 8  local A_i := select a valid subset of A from S_i
 9  local New := {F_1 = post(R|_{a_i}, S_i) |
10             a_i ∈ A_i ∧ ¬∃F_2 ∈ (Visited ∪ NotVisited) · F_1 ⊆ F_2}
```

To be a valid subset, A_i has to respect some conditions which are similar to the conditions C_0–C_4 of Section 2.4.1, except that they consider a different dependency relation which is not necessarily symmetric. We refer the reader to [AJKP98] for a precise definition of this dependency relation and those conditions. We note the following points:

[1] As consequence, the IsReachable loop invariant might become incorrect.

Algorithm 8.3: IsReachable

Header: IsReachable(M, G)

Precondition: $M = (A, AP, S, R, I, L)$ is a transition system, and $G \subseteq S$.

Result: *true* if and only if the reachable state space of M contains an element of G, i.e. $\text{post}^*(R, I) \cap G \neq \emptyset$.

Loop Invariant: The four following conditions hold:

(1) $I \subseteq \bigcup_{S_i \in (Visited \cup NotVisited)} S_i \subseteq \text{post}^*(R, I)$,
(2) $\text{post}(R, \bigcup_{S_i \in Visited} S_i) \subseteq \bigcup_{S_i \in (Visited \cup NotVisited)} S_i$,
(3) $found \Leftrightarrow \left(\bigcup_{S_i \in (Visited \cup NotVisited)} S_i\right) \cap G \neq \emptyset$,
(4) $Visited$ and $NotVisited$ are disjoint.

Halting condition: $frontier = \emptyset \vee found$

Variant: $2^{\#S} - \#Visited$.

Source Code: c.f. Listing 8.2

Listing 8.2: Implementation of the IsReachable algorithm

```
1  IsReachable(M = (A, AP, S, R, I, L), G) {
2    local NotVisited := {I}
3    local Visited := ∅
4    local found := false
5
6    while(frontier ≠ ∅ ∧ ¬found) {
7      local S_i := an element of NotVisited
8      local New := {F_1 = post(R|_a, S_i) |
9          a ∈ A ∧ ¬∃F_2 ∈ (Visited ∪ NotVisited) · F_1 ⊆ F_2}
10
11     found := (⋃_{S_i ∈ New} S_i) ∩ G ≠ ∅
12     Visited := Visited ∪ {S_i}
13     NotVisited := (NotVisited ∪ New)
14   }
15   return found
16 }
```

- The number of set operations which are performed during a run – e.g. computation of a post-image, intersection, ... – obviously depends on the real algorithm implementation. Nevertheless, an intuitive implementation will perform in the worst case on the order of $\#A * 2^{\#S}$ such operations. Indeed, the main loop is executed at most $2^{\#S}$ times. At each run, for each action of A_i a post-image computation is made. The maximum number of set operations $\#A * 2^{\#S}$ is extremely huge. However, the experimental results which are presented in [AJKP98] do not report any case which approaches this worst-case performance.

- The IsReachable algorithm works either in a forward way, or in a backward way. The latter amounts to running the algorithm on the reverse transition system M^{-1} but starting from the set G and trying to reach a initial state of I. Nevertheless, it is not mentioned how to select a valid subset A_i of A on the reversed transition system at each step. In particular, it is not mentioned if a process model can be computed on the original system, and then reused on the reversed one. Theoretically, our approaches can also be applied to a reverse transition system M^{-1}. However, we do not explore such an application because the process models we

compute are not applicable on M^{-1}.

- The algorithm is applicable, in a semi-decidable way, on transition systems which contain an infinite number of states. It means that the algorithm always terminates when at least one state of G is reachable. Contrarily, it could not terminate when no states of G are reachable. Other reachability algorithms fulfill this property as well, for instance breadth-first search algorithms, or iterative deepening algorithms. The BoundedPartialOrder algorithm of chapter 5 can also be applied on infinite model in a semi-decidable way.

Besides the worst-case complexity, the practical difference between this approach and ours is the checked properties. This approach deals both with backward and forward reachability analysis, while we are able to check a subset of CTL_X (c.f. Chapter 3) and LTL_X (c.f. Chapter 4 and Chapter 5) properties using only forward analysis.

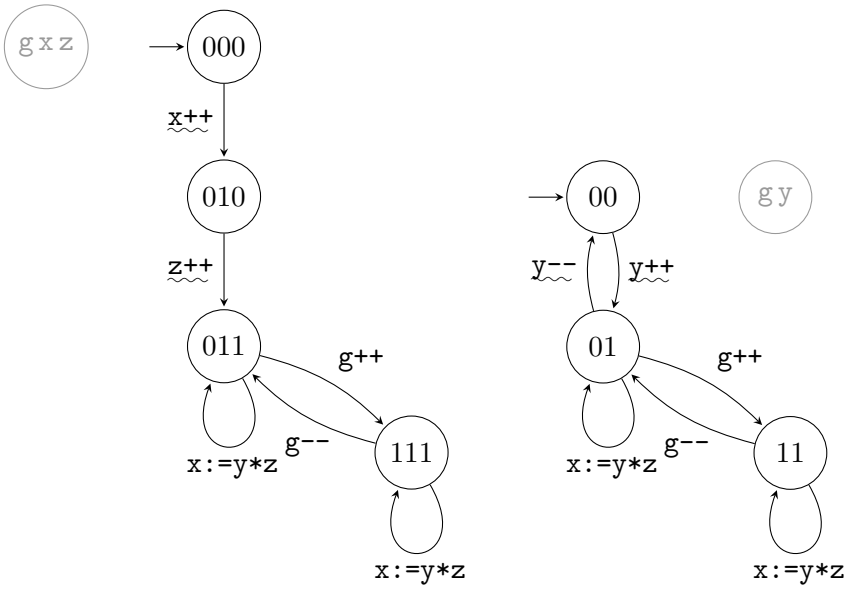
8.4 Bounded Model Checking of LTS

In his PhD dissertation, T. Jussila explores how to improve model checking of asynchronous systems [Jus05] using bounded model-checking approach as in Section 5.1.1. Those asynchronous systems are represented by means of *Labelled Transition Systems* (LTS). Compared to the transition systems which are defined in this thesis, an LTS can be seen as a transition system $M = (A, AP, S, R, I, L)$ with $AP = \emptyset$ and $L(s) = \emptyset$.

Definition 8.4 (Labelled Transition System)**.** *A* Labelled Transition System *is a structure* $M = (A_V \cup A_I, S, R, I)$ *where:*

- A_V *is a set of* visible actions*, and*
- A_I *is a set of* invisible actions *which is disjoint of* A_V*, and*
- S *is a set of states, and*
- $R \subseteq S \times (A_V \cup A_I) \times S$ *is a transition relation, and*
- $I \subseteq S$ *is a set of initial states.*

Figure 8.4 graphically presents two such LTSs which manipulate four Boolean variables `g`, `x`, `y` and `z`. The states of M_1 represent a valuation of the three variables `g`, `x` and `z`. The states of M_2 represent a valuation of the two variables `g` and `z`. The underlined actions are considered invisible, the others are considered visible.



(a) M_1 (b) M_2

Figure 8.4: Two synchronising LTSs

Jussila's work consists in defining four composition operators over LTS. It is a mathematical function which maps $n \geq 1$ LTS to a new one. For instance, the classical parallel composition operator of the two LTS of Figure 8.4 gives us our running example (c.f. Figure 8.5). To be precise, it does not represent the whole composition, but only the reachable states, i.e. the part of the system which we are interested in. Those composition operators are consistent with respect to deadlocks. It means that given n LTS, either all or none of the composition operators produce a new LTS which contains a reachable deadlock. Three out of four composition operators are also consistent with respect to reachability properties.

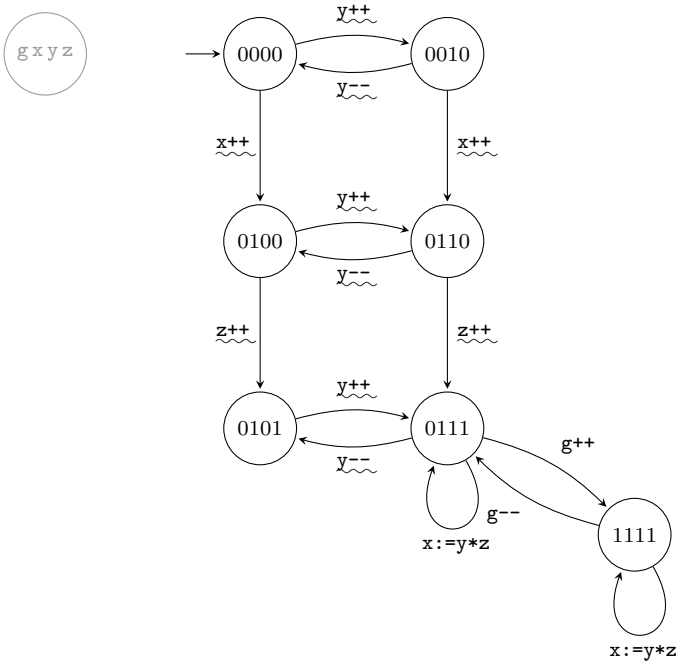


Figure 8.5: Classical Composition of M_1 and M_2 ($M_1 \,||\, M_2$)

Given the set of n LTS $\{M_0, M_1, \ldots M_{n-1}\}$ such that M_i, $M_i = (A_i, S_i, R_i, I_i)$, and the composed LTS $M = (A, S_0 \times S_1 \times \cdots \times S_{n-1}, R, I)$, following the principles of BMC of Section 5.1.1, any subset $\mathrm{tr}_k(M)$ of the executions of size k of M can be encoded into a propositional formula $F_k(M)$. In particular, $F_k(M)$ is unsatisfiable if and only if $\mathrm{tr}_k(M)$ is empty. Jussila's goal is to reduce the bound k needed to detect either a deadlock or a reachability property violation. The construction of the formula $F_k(M)$ respects the principles which are presented in Section 5.1.1. The main difference comes from the number of atomic propositions required to encode a state at a step t ($0 \leq t < k$). Given the state space $S_0 \times S_1 \times \cdots \times S_{n-1}$:

- The approach taken in this thesis requires around $\sum_{i=0}^{n-1} \log_2(\#S_i)$ $\approx \log_2(\#S)$ atomic propositions for representing a state at a step t.

- Jussila's approach needs $\sum_{i=0}^{n-1} \#S_i$ atomic propositions for representing a state at a step t. Indeed, an atomic proposition $\mathrm{in}(\mathtt{s_i}, \mathtt{t})$ is associated to each state $s_i \in S_i$ of each S_i. Intuitively, the atomic proposition $\mathrm{in}(\mathtt{s_i}, \mathtt{t})$ is true if and only if the composed state $(t_0, t_1, \ldots, t_{n-1})$ at step t contains the local state s_i, i.e. $t_i = s_i$. On the one hand, the classical parallel composition operator requires that exactly one atomic proposition $\mathrm{in}(\mathtt{s_i}, \mathtt{t})$ per S_i is true at each step t. On the other hand, non-classical composition operators remove that restriction. In particular, given a set of LTS, one approach of T. Jussila consists in determinizing the LTSs before composing them.

His work is decomposed in three parts: *partial-order semantics*, *on-the fly determinization*, and *merging of local transitions*. Those techniques can be combined together.

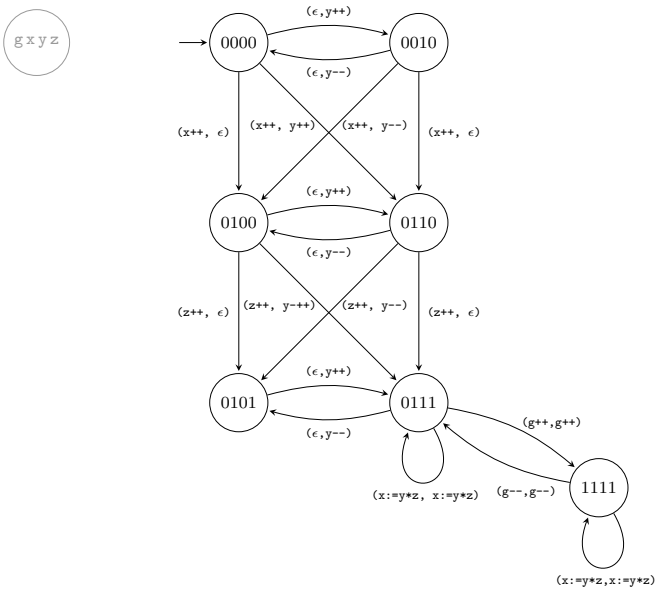
Partial-Order Semantics The first approach is the partial-order semantics. It defines two composition operators: the *step product* and the *process product*. The step product, noted $M_1 \,||_{st}\, M_2$ replaces the classical parallel composition operator with a non-standard operator which generates the same state space but a different transition relation. It allows the execution of several *independent* actions simultaneously.

The exact definition of independent actions are not important here, so we refer the reader to [Jus05, JN02] for a precise definition. Figure 8.6(a) shows the result of the step product composition of M_1 and M_2 from Figure 8.4. For instance, it shows that a transition can execute the two actions `x++` and `y++` in a unique transition. This behavior is not allowed by the classical parallel composition. As showed by Figure 8.6(a), this can reduce the bound needed to reach a particular state. For instance the state (1111) is reachable in 4 steps with the classical parallel composition operator, and it is reachable in 3 steps with the step product operator. The process product, noted $M_1 ||_{pr} M_2$, is the second variant of partial-order semantics. It only considers a subset of the different possible interleavings of $M_1 ||_{st} M_2$. It considers the interleavings which have a certain normal form not explained here. This can be seen by comparing Figure 8.6(a) and Figure 8.6(b).

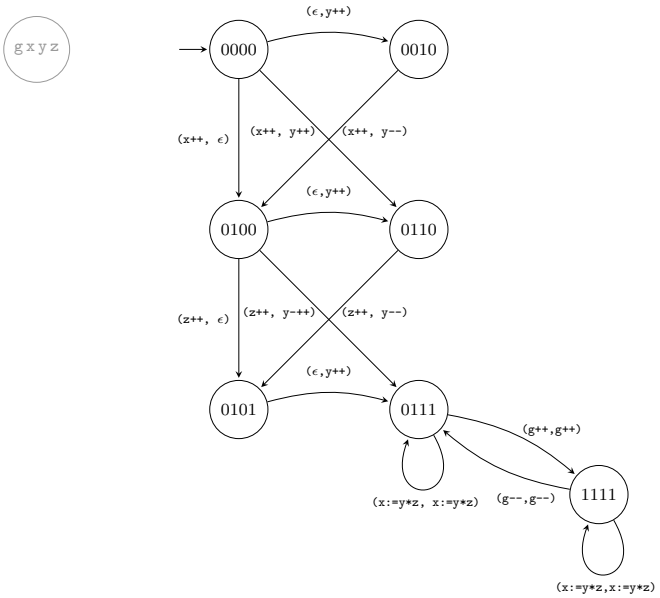
The two considered partial-order semantics are consistent with the parallel composition operator with respect to both deadlocks and reachability properties. They reduce the needed bound to discover either an error or a deadlock. The process product generates a LTS which has the same state space but fewer interleavings than the one generated by the step product. In general, this accelerates the final SAT solving problem.

A limited version of the process product which is called *Peephole Partial Order Reduction* was firstly presented in [WYKG08]. It considers a variant of transition systems with a usual interleaving model. Given an subset S_i of the states, and two independent actions a_1 and a_2, it does not consider the interleavings where a state of S_i appears from which a_1 and then $a2$ are fired. In other words, it does not consider the interleavings which contain a subpath $s_i \xrightarrow{a_1} s_{i+1} \xrightarrow{a_2} s_{i+2}$ such that $s_i \in S_i$.

On-the Fly Determinization It is well-known that a nondeterministic automaton which accepts a language L can be translated into another deterministic one which does not contain the empty string ϵ and which accepts the same language L [AU73]. In the same way, a set of LTS can be determinized, and then composed. The composed result which does not contain invisible actions is consistent with the classical parallel composition operator with respect to deadlocks and reachability properties. *On-the fly determinization* consists in deriving a propositional formula



(a) Step Product of M_1 and M_2 ($M_1 \,||_{st}\, M_2$)



(b) Process Product of M_1 and M_2 ($M_1 \,||_{pr}\, M_2$)

which represents the executions of the nondeterministic version of a LTS M_i [Jus05, JHN03, JHN05]. As seen in the beginning of this section, Jussila's encoding can easily represent the nondeterministic version of a LTS $M_i = (S_i, R_i, I_i, L_i)$ by allowing more than one atomic proposition $\text{in}(s_i, t)$ ($s_i \in S_i$) to be true at a step t. Removing invisible transitions is done as follows: given a state $s_i \in S_i$ the atomic proposition $\text{in}(s_i, t)$ is true if and only if s_i is an initial state, s_i can be reached at step t, or there exist both a state s_j which can be reached at step t and a invisible action a_i such that $s_j \xrightarrow{a_i} s_i$. M. In other words, $\text{in}(s_i, t)$ is true if and only if one of the three following conditions hold:

(1) $t = 0$ and s_i is an initial state, or

(2) $t \neq 0$ and there exists a $s_j \in S_i \cdot \text{in}(s_j, t-1) \wedge s_j \longrightarrow s_i$, or

(3) there exist a $s_j \in S_i$, and an invisible action a_i such that $\text{in}(s_j, t) \wedge s_j \xrightarrow{a_i} s_i$.

The determinized version of M_i is never created itself. The counterexamples are shortened because the invisible transitions are removed. Moreover, the number of executions is reduced since non-determinism is removed. Figure 8.6 illustrates that fact on the result of the on-the fly determinization composition of M_1 and M_2.

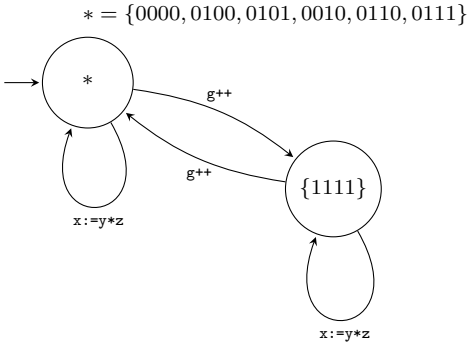


Figure 8.6: Determinized Synchronized Product of M_1 and M_2 ($M_1 \,||^d\, M_2$)

Merging of Local Transitions The final technique which is considered is called *local transition merging* [Jus04, Jus05]. Given two states s_0, s_n where a path $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ exists and some conditions are met, path π is replaced by a new transition $s_0 \xrightarrow{a_{new}} s_n$

Practically, a process starts by the *preprocessing phase* which removes invisible transitions while taking care of infinite sequences of invisible transitions. After that, transitions are added to the resulting model which correspond to some finite sequence of actions. Local transition merging composition and parallel composition are consistent with respect to deadlocks, but not to reachability properties. As with the three other composition operators, the bound needed to find a counterexample is reduced. Figure 8.7 shows the resulting graph when M_1 and M_2 are composed with local transition merging. Figure 8.7's arrows do not contain any label because each of its arrows actually represent one or more edges between the same pair of vertices. Indeed, the resulting LTS contains numerous labeled edges between vertices. Besides, in the present context, those labels are not required because we are only interested in reachable states.

We now compare our BoundedPartialExploration algorithm of Chapter 5, and the three approaches of Jussila's Thesis. The four methods present improvements to bounded model checking of asynchronous systems. Nevertheless, here are the main differences:

- As seen in the beginning of this section, in the worst case Jussila's encoding requires exponentially more variables than our encoding.
- Jussila's approaches only addresses reachability problems whereas our BoundedPartialExploration method allows us to verify of LTL_X properties.
- Jussila's goal is to reduce the bound needed to discover an error. To achieve that, the initial system is transformed into other systems which might be larger than the original one. Contrarily, our approach tries to reduce the graph by increasing the bound needed for finding a violation, while still reducing the verification cost.

We do not have quantitive data which compare Jussila's approaches to our BoundedPartialExploration algorithm. This remains to be investigated. However, we have the intuition that Jussila's partial-order

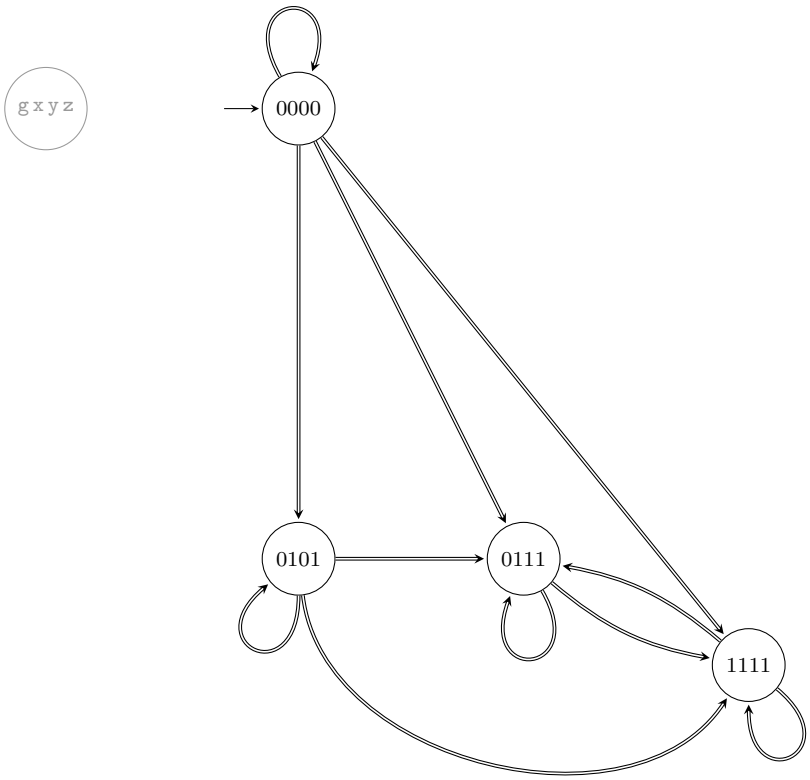


Figure 8.7: Limited Path Product of (the preprocessed version of) M_1 and M_2 ($M_1' \,||^l_{pt}\, M_2'$)

semantics and our partial-order reduction could be merged together to fire synchronously safe transitions, and thus obtain more reduction. We also think that combining the two approaches should also apply in other contexts than bounded model checking, e.g. set-based model checking. On-the-fly determinization can be seen as a complementary method to ours. In general, when asynchronous systems are considered, two causes of non-determinism are identified: the first one comes from the components themselves, and the second one comes from the interleaving execution model. The former is handled by on-the-fly determinization while our method tackles the latter. All three of Jussila's approaches are potentially applicable and open interesting directions for further work.

8.5 The Saturation Approach

G. Ciardo at al. introduce in a series of articles the *saturation* approach [CLM07, CLS00, CLS01, CMS06, CY05, Min04, Min06, WC09]. It computes the reachable state space of a transition system $M = (D(v_0) \times \cdots \times D(v_{n-1}), R, I, L)$ where each $D(v_i)$ is the finite domain of a variable v_i. Therefore, states are tuples which implicitly represent a valuation of a sequence of finite domaine variables $(v_0, v_1, \ldots, v_{n-1})$. The saturation approach exploits the two following facts:

- In order to compute the reachable state space of M, that state space can be visited in any order. Suppose a transition system with a set A composed of two actions a_1 and a_2. From the initial states of M, an algorithm might fire as many a_1 actions as possible. When no a_1 action remains to be fired, it continues by expending the a_2 actions from the already visited states. Meanwhile, the states which were discovered by actions a_2 might enable some a_1 actions. Thus, the algorithm needs to execute a_1 actions again, then a_2 actions, and so on until no states remain to be discovered.

- Given a variable v_ℓ, a sub-transition relation $R_{v_\ell} \subseteq R$ can be *independent* of the value of the variable v_ℓ. It means that for each transition $s \longrightarrow_{R_{v_\ell}} t$, its enabling does not depend on the v_ℓ value, and its firing to not modify the v_ℓ value. Such a transition relation is called an *independent transition relation* with respect to the v_ℓ variable.

Definition 8.5 (Local Transition Relation)**.** *Given a sequence of variables* $(v_0, v_1, \ldots, v_{n-1})$*, and a level* $0 \leq \ell \leq n$*, a sub-transition relation* $R_\ell \subseteq R$ *is a* local transition relation *with respect to the level* ℓ *if and only if* R_ℓ *is an independent transition relation with respect to all the variables* $v_0, v_1, \ldots, v_{\ell-1}$*.*

Before presenting the saturation algorithm itself, we explain how the transition system is encoded. Depending of the articles, sets of states $S' \subseteq S$ are symbolically encoded using various variants of *multi-way decision diagrams* (MDDs) [MD98]. MDDs extend BDDs by allowing multi-valued variables v_ℓ over a domain $D(v_\ell)$, so that the choices for nodes at level ℓ correspond to the values of D_ℓ. A node n_ℓ at level ℓ characterizes a subset S_ℓ of S'. All the paths from the top to n_ℓ represent the set $\text{prefix}(\ell, S_\ell)$ of prefixes of length ℓ of S_ℓ. All the paths from n_ℓ to the bottom represent the set $\text{suffix}(\ell, S_\ell)$ of suffixes of length $n - \ell$ of S_ℓ. Moreover, S_ℓ is the cross product of $\text{prefix}(\ell, S_\ell)$ and $\text{suffix}(\ell, S_\ell)$. The k nodes $n_0, n_1, \ldots, n_{k-1}$ at level ℓ characterize a partition $\{S_0, S_1, \ldots, S_{k-1}\}$ of S'. The transition relations are generally encoded by means of *Kronecker Products* [Dav81], or some MDD variants.

The saturation algorithm supposes a transition system $M = (D(v_0) \times \cdots \times D(v_{n-1}), R, I, L)$, a set of local transition relations $\{R_0, R_1, \ldots, R_n\}$ where each R_i is a local transition relation with respect to the level i and $R = \bigcup_{i=0}^{n} R_i$, and a set of states S_ℓ which is the cross product of $\text{prefix}(\ell, S_\ell)$ and $\text{suffix}(\ell, S_\ell)$ as defined above. The saturation approach performs the following operations:

- It saturates S_ℓ with respect to level ℓ. It means that it looks for states which are accessible from S_ℓ by only expanding the transition relations which are local with respect to a level i greater or equal to ℓ, i.e. $\{R_\ell, R_{\ell+1}, \ldots, R_{n-1}\}$. At the end, it returns a saturated version V of S_ℓ, i.e. $V = \text{post}^*(\bigcup_{i=\ell}^{n-1} R_i, S_\ell)$.

- To saturate S_ℓ, the algorithm performs a number of post-image computations. Those post-images are not directly computed on S_ℓ but on the suffixes $\text{suffix}(\ell, S_\ell)$ of S_ℓ. To achieve that, a variant $R_{|\ell}$ of R_ℓ is required. $R_{|\ell} \subseteq (D(v_\ell), D(v_{l+1}), \ldots, D(v_{n-1}))^2$ is the projection of R_ℓ to the variables $v_\ell, v_{\ell+1}, \ldots, v_{n-1}$. Finally, the

post-image operation is computed according to Equation (8.2):

$$S_\ell = \text{prefix}(\ell, S_\ell) \times \text{suffix}(\ell, S_\ell) \tag{8.1}$$

$$\text{post}(R_\ell, S_\ell) = \text{prefix}(\ell, S_\ell) \times \text{post}(R_{|\ell}, \text{suffix}(\ell, S_\ell)) \tag{8.2}$$

The saturation approach works well with MDDs for the following reasons:

- It tends to produce smaller intermediate MDDs than traditional approaches which compute at each step the post-image of all the visited states with the whole transition relation R [CMS06].
- The suffix sets $\text{suffix}(\ell, S_\ell)$ are characterized by a unique MDD node n_ℓ. Moreover, if two suffix sets $\text{suffix}(\ell, S_\ell)$ and $\text{suffix}(\ell, T_\ell)$ are equal, they are represented by the same MDD node n_ℓ. As so, if $\text{suffix}(\ell, S_\ell)$ is saturated before $\text{suffix}(\ell, T_\ell)$, $\text{suffix}(\ell, T_\ell)$ is saturated, then n_ℓ is marked as saturated. When the latter $\text{suffix}(\ell, T_\ell)$ is saturated, the algorithm directly notices than n_ℓ is marked as saturated, so it does not have to compute the saturation of $\text{suffix}(\ell, T_\ell)$.

The saturation algorithm is composed of two sub-problems. The main one (c.f. Specification 8.9) takes as input a level ℓ and a set of states S_ℓ. It looks for states which are accessible from S_ℓ by satureting $\bigcup_{i=\ell}^{n-1} R_i$. It returns a saturated version V of S_ℓ. To achieve that goal, it calls the oneStep sub-problem (c.f. Specification 8.8) which also takes as input a level ℓ and a set of states S_ℓ. It looks for states which are accessible from S_ℓ by performing $n > 0$ actions such that all the actions except the last one belong to $\bigcup_{i=l+1}^{n-1} R_i$ and the last action belongs to R_ℓ.

Extensive research has been performed around the concept of saturation. It was among other things extended to compute a kind of bisimulation [MC11], to check CTL properties [ZC09], to compute *strongly connected components* of transition systems [ZC10, ZC11], and to find shortest counter-examples [ZJC11].

In the sequel, we compare the way of working of the saturation approach with our PartialExploration method which combines POR and symbolic model checking. Unfortunately, we are not able to make a quantitative comparison about running time or memory consumption because we did not have access to a running version of the saturation

algorithm. Our comparison is focused on the tactics which are applied by both methods to visit the state space. Given a transition system $M = (S, R, I, L)$, the two approaches create sub-transition relations $\{R_0, R_1, \ldots, R_{n-1}\}$ from the whole transition relation R. At each step, the two approaches extend a set S_i which corresponds to a part of the visited state space. To visit the state space, they apply different strategies:

- In fine, the saturation algorithm fully expands all the states of a transition system and so visits the whole state space of M. Contrarily, our partial exploration approach partially expands some states and so produces a reduced states space of M. In general, this produces smaller intermediate BDDs (c.f. Chapter 7) than traditional approaches.
- The saturation algorithm divides the visited state space into various parts, and performs several post-image computations on those parts. When a part does not allow to discover new states, another one which maybe subsumes the previous one is considered. According to experimental results, this tends to produce much smaller intermediate MDDs [CMS06]. At each step, PartialExploration approach considers all the states which remain to deal with.
- In practice, the process model which is automatically created by the Milestones model checker (c.f. Chapter 6) and the partitioning of the transition relation R required by the saturation approach are constructed by analyzing which variables are accessed when an action a is performed. In the Milestones context, for each transition relation R_i of the generated process model, and each variable which compose the BDDs, we can automatically deduce if R_i is a local transition relation with respect to that variable.

Both the saturation algorithm and the PartialExploration approach have been adapted to verify CTL properties. The partial exploration algorithm has also been used to verify LTL properties. The saturation does not directly allow us to verify LTL properties, but it has been adapted to compute the maximal strongly connected components of a transition system. This could thus be used to check LTL properties.

Algorithm 8.8: OneStep

Global: A transition system $M = (S, R, I, L)$ where $S = D(v_0) \times \cdots \times D(v_{n-1})$ and a set of local transition relation $\{R_0, R_1, \ldots, R_{n-1}\}$ where each R_i is a local transition relation with respect to the level i, and $R = \bigcup_{i=0}^{n} R_i$

Header: OneStep(ℓ, S_ℓ)

Precondition: $0 \leq l < n$, and S_ℓ is a set of states. Moreover, S_ℓ can be decomposed as follows: $S_\ell = \text{prefix}(\ell, S_\ell) \times \text{suffix}(\ell, S_\ell)$.

Result: $\text{post}(R_\ell, \text{post}^*(\bigcup_{i=l+1}^{n-1} R_i, S_\ell))$.

Loop Invariant: $D' \subseteq D(v_\ell)$, and $V = \text{post}^*(\bigcup_{i=l+1}^{n-1} R_i, \{(s_0, \ldots, s_{n-1}) \in S_\ell \mid s_\ell \in D(v_\ell) \setminus D'\})$ and V can be decomposed as follows $V = \text{pre}(\ell, S_\ell) \times \text{pre}(\ell, V)$

Halting condition: $D' = \emptyset$

Variant: $\#D'$

Source Code:

```
1  OneStep(ℓ, S_ℓ) {
2    local V ∈ 2^S := ∅
3    local D' ⊆ D(v_ℓ) := D(v_ℓ)
4
5    while (D' ≠ ∅) {
6      local e := any element of D'
7      D' := D' \ {e}
8      T_0 := {(s_{ℓ+1}, ..., s_{n-1}) | (e, s_{ℓ+1}, ..., s_{n-1}) ∈ suffix(ℓ, S_ℓ)}
9      T_1 := Saturate(l + 1, (prefix(ℓ, S_ℓ) × {e}) × T_0)
10     V := V ∪ T_1
11   }
12   return prefix(ℓ, S_ℓ) × post(R_|ℓ, suffix(ℓ, V))
13 }
```

Algorithm 8.9: Saturate

Global: A transition system $M = (S, R, I, L)$ where $S = D(v_0) \times \cdots \times D(v_{n-1})$ and a set of local transition relation $\{R_0, R_1, \ldots, R_{n-1}\}$ where each R_i is a local transition relation with respect to level i, and $R = \bigcup_{i=0}^{n} R_i$

Header: Saturate(ℓ, S_ℓ)

Precondition: $0 \leq \ell < n$, and S_ℓ is a set of states. Moreover, S_ℓ can be decomposed as follows: $S_\ell = \text{prefix}(\ell, S_\ell) \times \text{suffix}(\ell, S_\ell)$.

Result: $\text{post}^*(\bigcup_{i=l}^{n-1} R_i, S_\ell)$.

Loop Invariant: $S_\ell \subseteq V \subseteq \text{post}^*(\bigcup_{i=l}^{n-1} R_i, S_\ell)$ and $V = \text{post}(\text{post}^*(\bigcup_{i=l+1}^{n-1} R_i, O))$, and V can be decomposed as follows $V = \text{prefix}(\ell, S_\ell) \times \text{suffix}(\ell, S_\ell)$

Halting condition: $V = O$

Variant: $\#S - \#O$

Source Code:

```
1  Saturate(ℓ, S_ℓ) {
2    local V ⊆ S := S_ℓ
3    if (l ≠ n ∧ suffix(ℓ, S_ℓ) is not marked as saturated){
4      local O ⊂ S := V
5      V := OneStep(ℓ, V);
6      while(O ≠ V) {
7        O := V
8        V := OneStep(ℓ, V);
9      }
10     mark suffix(ℓ, V) as saturated
11   }
12   return V;
13 }
```

In order to benefit from the two methods, we have the intuition that POR methods can be incorporated into the saturation approach. Actually, a process model could be used as a partitioning of the transition relation. Besides, the saturation could partially visit a safe part of a state space. Finally, we think that the greatest challenge would be to find a way to discover cycles. This problem could be resolved by applying a static analysis to the high-level description of the transition system (c.f. section 8.2).

8.6 From Cycle Detection to Reachability

In [BAS02], A. Biere et al. translate an LTL model checking problem into a reachability problem. The basic idea is that finding an LTL counter-example amounts to finding a *lasso-shaped* fair trace. It consists of a prefix that leads to a fair loop. After the translation, a fair loop is found by nondeterministically saving a state s_i during the search. Then, the search continues by checking if the current state is s_i and if all the fairness constraints are met since s_i was firstly visited. The algorithm could be seen as a symbolic incarnation of the classical nested depth first search algorithm [CVWY92]. We note that the transformation mechanism of [BAS02] was extended to be applied to some infinite transition systems [SB06], e.g. to push-down automata.

Given a transition system $M = (S, R, I, L, F)$, a proposition $p \in AP$, and a temporal property $\mathsf{EG}\, p$ to be satisfied (i.e. check that $\neg\mathsf{EG}\, p$), the approach starts by generating a transition system M_p^F which is encoded by means of BDDs. All of its states are composed of $\#F + 3$ components. They have the form $(s_1, s_2, ok, f_0, \ldots, f_{\#F-1})$ s_1 represents a state of S. s_2 represents the hypothetical first state of a loop. If s_2 equals to $\perp$, it means that the loop has not started yet. ok is true if and only if all the states that have already been reached satisfy p. All the f_i components states are true if and only if the fairness constraints $F_i \in F$ have been met since the loop started. Because every state of M_p^F contains the two states s_1 and s_2, at least twice as many BDD variables are required to encode M_p^F than M. The authors show that M and M_p^F are equivalent.

Definition 8.6 (M_p^F)**.** *Given a fair transition system* $M = (A, AP, S, R, I, L, \{F_0, F_1, \ldots, F_{k-1}\})$*, and* $p \in AP$*, a derived transition system*

$M_p^F = (\{q\}, A, S_p^F, R_p^F, I_p^F, L_p^F)$ *is derived where:*

- $S_p^F = S \times (S \cup \{\bot\}) \times \{\text{true}, \text{false}\}^{k+1}$ *where* $\bot \notin S$
- $(s_1, s_2, ok, f_0, \ldots, f_{k-1}) \xrightarrow{a} (s_1', s_2', ok', f_0', \ldots, f_{k-1}') \in R_p^F$ *if and only if all following conditions hold:*

  (1) $s_1 \xrightarrow{a} s_1' \in R$, *and*

  (2) $(s_2 = \bot \wedge s_2' = s_1') \vee s_2' = s_2$, *and*

  (3) $ok' = (ok \wedge p \in L(s_1'))$, *and*

  (4) $\forall i \in \{0, 1, \ldots, k\} \cdot f_i' = (s_2 \neq \bot \wedge (f_i \vee s_1' \in F_i))$
- $I_p^F = \{(i_1, i_2) \mid i_1 \in I \wedge i_2 \in (I \cup \{\bot\}) \wedge (i_2 = \bot \vee i_1 = i_2)\} \times \{\text{true}\} \times \{\text{false}\}^k$
- $L_p^F : S_P^F \to \{\emptyset, \{q\}\} : (s_1, s_2, ok, f_0, \ldots, f_{k-1}) \to R$ *such that* $q \in R \Leftrightarrow s_2 \neq \bot \wedge ok \wedge \forall i \in \{0, 1, \ldots, k-1\} \cdot f_i$.

Theorem 8.7 (c.f. [BAS02])**.** *Given a fair transition system* M *and its derived transition system* M_p^F

$$M \models_F \mathsf{EG}\, p \text{ if and only if } M_p^F \models \mathsf{EF}\, q$$

Every LTL model checking problem can be translated into a fair cycle detection problem (c.f. Section 4.1.3). Therefore, the approach is applicable to all LTL properties. Furthermore, M_p^F can be visited by means of any algorithm which deals with reachability problem such as DFS, BFS, or the forward model checking algorithms of Section 2.5.2.

Incorporating partial-order reduction into symbolic model checking to verify temporal properties is not trivial, among others things because detecting cycles is a difficult task. We have the intuition that the approach presented in this section which does not deal with POR might alleviate the problem. It would be interesting to study the effect of partial-order reduction techniques on the graph M_p^F. To achieve that goal, the following questions need to be answered:

1. Is a safe (and linear) process model on M still safe (and linear) on M_p^F?

2. If the ample set approach is used which of the conditions C_0–C_4, or which variants, have to be met? In particular, what do the cycle condition C_3 become? Can it be relaxed or even discarded?

3. M_p^F is much bigger than M, i.e. $\#S_p^F = 2^{k+1}\#S^2$. M_p^F is encoded by means of at least twice as many BDD variables as M. Because the size of a BDD can be exponential in terms of the numbers of BDD variables, the size of the BDDs can become unmanageable. Partial order reduction could alleviate that problem. However, if M_p^F were encoded into BDDs, what would the impact of partial-order reduction on memory and running-time be?

8.7 Conclusion

In this chapter, we reviewed model checking approaches which are related to ours. We started by presenting three methods which combine symbolic model checking and partial-order reduction:

- The method of R. Alur et al. which checks local properties, a sort of reachability properties [ABH$^+$97].

- The static partial-order reduction method of R. P. Kurshan which translates the source code of a system into a new one to perform POR at a syntactic level [KLM$^+$98, KLY02]. The resulting system is equivalent to the original one, and can be handled by any model checking techniques.

- The approach of P. A. Abdulla et al. which verifies reachability properties [AJKP98]. It considers an uncommon dependency relation called commutativity in one direction, and works with sets of sets of states instead of set of states.

We continue by introducing three approaches that are complementary to ours:

- Jussila's methods which are related to BMC of asynchronous systems [Jus05]. They consider various interleaving semantics to verify reachability problems.

- Ciardo's saturation approach which exploits the two following observations. The state space of a system can be visited in any order [CLS01]. Moreover, in a lot of cases, the firing of an action does not depend on all the variables of a system. Contrarily, it generally depends on a small number of such variables.

- The approach of A. Biere et al. which translates a liveness problem into a safety problem [BAS02]. It translates the fair-cycle detection problem into a reachability problem, and so deals with LTL properties.

In addition to the applied methods, the considered approaches differ from the class of properties they can handle. Three of the six presented strategies deal with a kind of reachability property: the methods of R. Alur et al., P. A. Abdulla al. and T. Jussila. Ciardo's saturation also deals with reachability properties but it was extended to among other things verifying CTL properties or handling bisimulation problems. The two remaining approaches of R. P. Kurshan et al. and A. Biere et al. deal with LTL properties. Finally, the methods presented in this thesis verify either LTL_X properties or CTL_X properties.

In general, the approaches presented in this section are complementary to the one presented in this thesis. In the context of set-based model checking, the static partial-order reduction or the method of A. Biere et al. could be used to detect cycles. Then, the PartialExploration algorithm, the saturation approach, or a mix of both could visit to the resulting graph. In the same way, the BoundedPartialExploration algorithm and the partial-order semantics of T. Jussila could be merged. Those suggestions lead to interesting open questions for future researches.

Chapter 9

Conclusion and Perspectives

The research goal of this thesis is to develop efficient symbolic model checking techniques for asynchronous systems. The key idea is to combine partial-order reduction (POR) and symbolic model checking to tackle the state space explosion problem which is inherent to model checking of concurrent systems. In Section 9.2, we outline the main limitations of our work, and we present several possible directions for future research.

9.1 Summary

The contributions presented in this thesis focus on symbolic verification of CTL_X and LTL_X properties on concurrent systems, which are described by finite transition systems. To perform this verification, we propose three algorithms which incorporate partial-order reduction either into BDD-based model checking or into SAT-based model checking. Given a transition system M, those algorithms only visit a part of a reduced transition system M_R.

The first algorithm, called *evalCTLX*, combines POR and set-based model checking to check whether a transition system verifies a CTL_X property or not. At the heart of the evalCTLX algorithm is the Partial-Exploration algorithm which performs partial-order reduction to visit a part of M_R. It is employed as an alternative to the forward model checking of [INH96] which allows to verify a subset of CTL in a forward

way. To check the whole class of CTL properties, it is combined with the classical backward model checking approach [CGP99]. Our evalCTLX algorithm applies the same principles but it uses the PartialExploration algorithm to perform partial-order reduction during the forward phase.

The second approach, referred to the evalLTLX algorithm, uses the PartialExploration algorithm to verify LTL_X properties. It adapts and merges the tableau-based LTL model checking of [CGH97] which amounts to searching for fair paths, the forward model checking of [INH96] and the PartialExploration algorithm to search for fair cycle using forward state traversal in a reduced state space.

The third algorithm, named BPE, adapts the PartialExploration algorithm to be suitable for bounded model checking. It is used to find violation of LTL_X properties. It reduces the search space which the SAT-solver has to deal with because it does not encode all the possible interleavings. To that end, it applies partial-order reduction principles and adds stuttering into the paths it encodes to obtain a more efficient SAT problem resolution.

To evaluate and compare our three algorithms, we developed the Milestones model checker from scratch. Milestones defines its own language to model a system. Besides our three approaches, it implements classical model checking techniques for comparison purposes. It also allows one to translate a Milestones program into a NuSMV or a Spin program.

Finally, we assess and compare the three approaches on four examples. As shown by the experimental results, these approaches have the following advantages:

- In the case of BDD-based model checking, it manipulates smaller BDDs than classical BDD-based approaches and so accelerates the verification process.

- In the case of SAT-based model checking, it visits a reduced transition system M_R which contains less non-determinism than M. This tends to accelerate the resolution of the underlying SAT problem, and so accelerate the verification process as well.

In conclusion, the following contributions have been presented, as announced in Section 1.1:

1. Our three new algorithms for efficient symbolic model-checking of asynchronous systems based on POR techniques were the subject of Chapters 3, 4, and 5:

   (a) Chapter 3 introduced the PartialExploration algorithm which is inspired from the ImProviso algorithm. Then, it presented the evalCTLX algorithm which incorporates the PartialExploration algorithm into forward model checking.

   (b) Chapter 4 presented the evalLTLX algorithm which verifies LTL_X properties. It merged our PartialExploration algorithm, forward model checking, and Clarke et al.'s tableau-based symbolic LTL model checking.

   (c) Chapter 5 introduced the BPE algorithm which applies PartialExploration principles into the bounded model checking approach to allow the verification of LTL properties.

2. Chapter 6 presented the Milestones model checker which implements the previous algorithms. For comparison purposes, it implements some other classical approaches. Besides, it translates its programs into NuSMV or Spin.

3. Each of the three algorithms was supported by a problem theory section (Section 3.2.1, Section 5.1, Section 4.1) providing detailed development and proof. In the Related Work Chapter 8, comparable approaches have also been re-expressed and rigorously specified within the same unifying framework.

4. Chapter 7 assessed the effectiveness and the scalability and compared the previous algorithms on four example models: a scalable turntable system, an elevator system, a cash point system, and a producer-consumer system.

9.2 Future Work

The evalCTLX algorithm applies partial-order reduction when forward model checking is applicable, otherwise it uses backward model checking without any partial order reduction. We see two main limitations to this approach:

1. Partial-order reduction is applied only during a part of the verification process. To improve the performances, could we increase the relative part during which partial-order reduction is performed?

2. On the one hand, partial-order reduction is applied during the forward phase and not during the backward one. On the other hand, experimental results show that in some cases the performances of the backward model checking are very high,while the performances of the forward model checking are very poor, and vice versa. Therefore, it would be interesting to study how partial-order reduction could be applied during backward model checking.

Undeniably, those two questions have a practical aspect, but they are also appealing from a theoretical point of view. To answer them, we propose three approaches. The two first approaches are related to our first question and the third approach is related to the second question:

- A possible direction consists in studying more deeply the links between forward model checking and backward model checking:

    1. The first part of this study would be devoted to answer to the following question: what kind of property is it possible to check in a forward way and what kind of property is it possible to check in a forward way? A first starting point is the work of T. A. Henzinger et al. which is relative to those two questions [HKQ03]. The authors formulate the problems of symbolic backward and forward model checking by means of two μ-calculi. The pre-μ calculus is based on the pre-image operation, and the post-μ calculus is based on the post-image operation. Then, the authors prove that all LTL properties can be expressed as post-μ queries, and therefore checked using symbolic forward state traversal. On the other hand, they show that there are simple CTL properties that cannot be checked in this way. A second starting point is the work of B. Vergauwen et al. which introduces an explicit forward model checking for CTL [VL93]. We have a strong intuition that this algorithm could be adapted to set-based model checking.
    2. Because forward model checking can be used to check the past version of the CTL operators, or *CTL+PAST*, and the

backward model checking temporal can be used to check classical version of the CTL operator. Another starting point is the work F. Laroussinie et al. about past CTL [LS97] which among other things translates a fragment of CTL+PAST into CTL, even though we need the opposite.

- Another interesting challenge is to extend symbolic LTL model checking by tableau of E. M. Clarke et al. [CGH97] in such a way that it deals with CTL* properties, then, to incorporate POR into the resulting algorithm. Actually, in explicit model checking, CTL* model checking can be performed by combining both the explicit LTL model checking algorithm by tableau and the CTL model checking algorithm by tableau [CGP99, BK08]. The resulting algorithm has essentially the same complexity as the explicit LTL model checking by tableau [CGP99, BK08]. Adapting this explicit algorithm could be done in a way similar to the adaptation of the explicit LTL model checking by tableau for symbolic model checking.

- Partial-order reduction could be applied during backward model checking. Given a transition system M, it amounts to applying partial-order reduction technique on the reverse transition system M^{-1}. Actually, with a low level language which supports non-determinism, it is possible to translate a description of a transition system M in that language into a description of a transition system M^{-1} in that language as well. As with Milestones, this representation of M^{-1} could be used to concretely compute the process model of M^{-1}. Based on that, we could reduce M^{-1}. We identify two obstacles to this approach: non-determinism and the detection of the initial states. We recall that POR approaches generally deal with deterministic transition system. It is easy to see that a deterministic transition system M can induce a non-deterministic transition system M^{-1}. Hence, the first task consists in adapting the POR methods to non-deterministic transition systems. The second obstacle comes from the fact that the reduced version of M^{-1} probably has to contain all the initial states of M, otherwise it seems difficult to state something about the validity of the reachable state space of M.

Classically, the partial-order reduction approaches rely on the notions of independence and invisibility of actions. Moreover, the POR condition C_1 of Section 2.4.1 describes in which conditions an action can enable another one when partial order reduction is applied. In general, checking independence, invisibility, and condition C_1 are a difficult problem, and so some over-approximations are classically used to ensure that the ample sets are valid. For instance, given a set of state S and a set of actions A, set-based approaches or SAT-Solvers could be used to answer the following types of questions:

- Are action a_1 and action a_2 independent?
- Can action a_1 activate action a_2? In which conditions a_1 does activate action a_2?
- Is a_1 invisible with respect to a set of propositions AP'?

The answer to those questions could be used to compute the previously mentioned over-approximations. Finally, these over-approximations could be used in any partial-order algorithm which relies on them.

Concerning our BPE algorithm, for now it is used to verify finite transition. However, from a theoretical point of view, it can be extended to handle models featuring variables on infinite domains. This can be achieved by using the capabilities of Satisfiability Modulo Theories solvers such as Yices and MiniSat. Furthermore, [Str01, WKS01, HJL05] show that a promising technique for improving the performance of BMC consists in using *incremental* SAT-solving. Our BPE algorithm could be adapted to the context of incremental SAT-solving. Moreover, we have the intuition that the partial order semantics of T. Jussila and our partial order reduction could be merged to fire synchronously safe transitions, and so obtain more reduction. For now, Jussila's approach is performed in the bounded model checking context. However, from a theoretical point of view this approach is also applicable in the context of set-based model checking. Therefore, in the context of set-based model checking, it could also be possible to combine the partial order semantics of T. Jussila and our partial order reduction to fire synchronously safe transitions, and so obtain more reduction.

To be usable in other contexts than this thesis, the Milestones model checker needs mainly two extensions. Firstly, counterexamples generation

needs to be added. [CGMZ95] shows how to generate such counterexamples. Secondly, some higher-level language extensions also needs to be added. To achieve that, the easiest, is to provide a set of macros which maps such extensions to the Milestones languages.

Finally, we stress that in the last two decades a lot of work has been done in the domain of symbolic model checking. We review some of that work in Chapter 8, e.g. the Saturation approach of G. Ciardo or the approach presented in of A.Biere et al which transforms a liveness problem into a safety one. An interesting topic for further work consists in studying how those approaches and the ones presented in this thesis can benefit from each other.

Bibliography

[ABH+97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351, 1997.

[AJKP98] Parosh Aziz Abdulla, Bengt Jonsson, Mats Kindahl, and Doron Peled. A general approach to partial order reductions in symbolic verification (extended abstract). In *Computer Aided Verification*, pages 379–390, 1998.

[AU73] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling: Parsing.* The Theory of Parsing, Translation, and Compiling. Prentice-Hall, 1973.

[BAS02] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electr. Notes Theor. Comput. Sci.*, 66(2):160–177, 2002.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[BCG88] M. C. Browne, E. M. Clarke, and O. Gumberg. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59(1-2):115–131, 1988.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDOS08] Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. Design and results of the 3rd annual satisfiability modulo theories competition (smt-comp 2007). *International Journal on Artificial Intelligence Tools*, 17(4):569–606, 2008.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[BTW+05] Elena M. Bortnik, Nikola Trčka, Anton Wijs, Bas Luttik, J. M. van de Mortel-Fronczak, Jos C. M. Baeten, Wan Fokkink, and J. E. Rooda. Analyzing a χ model of a turntable system using spin, cadp and uppaal. *J. Log. Algebr. Program.*, 65(2):51–104, 2005.

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In *Proc. of International Conference on Computer-Aided Verification*, 1999.

[CD89] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 428–437, London, UK, 1989. Springer-Verlag.

[CGH97] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. *Form. Methods Syst. Des.*, 10(1):47–71, 1997.

[CGMZ95] Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC*, pages 427–432, 1995.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. Mit Press, 1999.

[CLM07] Gianfranco Ciardo, Gerald Lüttgen, and Andrew S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31(1):63–100, 2007.

[CLS00] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *ICATPN*, pages 103–122, 2000.

[CLS01] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2001.

[CMS06] Gianfranco Ciardo, Robert M. Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *STTT*, 8(1):4–25, 2006.

[CMT11] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. Hydi: A language for symbolic hybrid systems with discrete interaction. In *EUROMICRO-SEAA*, pages 275–278. IEEE, 2011.

[CVWY92] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[CY05] Gianfranco Ciardo and Andy Jinqing Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2005.

[Dav81] Marc Davio. Kronecker products and shuffle algebra. *IEEE Trans. Comput.*, 30:116–125, February 1981.

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for dpll(t). In Thomas Ball and

Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DOP00] B. Tim Denvir, José Nuno Oliveira, and Nico Plat. The cash-point (atm) 'problem'. *Formal Asp. Comput.*, 12(4):211–215, 2000.

[EFT93] Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating bdds for symbolic model checking in ccs. *Distrib. Comput.*, 6(3):155–164, 1993.

[ELLL04] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. *STTT*, 6(4):277–301, 2004.

[EPF12] EPFL. Lausanne, switzerland, The Scala Programming Language Website. `http://www.scala-lang.org/`, January 2012.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. Cadp (cæsar/aldebaran development package): A protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of CAV'96 (New Brunswick, New Jersey, USA)*, volume 1102, pages 437–440, 1996.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In Dragan Bosnacki and Stefan Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2007.

[GJM+97] Hubert Garavel, Mark Jorgensen, Radu Mateescu, Charles Pecheur, Mihaela Sighireanu, and Bruno Vivien. Cadp'97 –

status, applications and perspectives. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, 1997.

[GKPP99] Rob Gerth, Ruurd Kuiper, Doron Peled, and Wojciech Penczek. A partial order approach to branching time logic model checking. *Information and Computation*, 150(2):132–152, 1999.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[HGP92] Gerard J. Holzmann, Patrice Godefroid, and Didier Pirottin. Coverage preserving reduction strategies for reachability analysis. In Richard J. Linn Jr. and M. Ümit Uyar, editors, *PSTV*, volume C-8 of *IFIP Transactions*, pages 349–363. North-Holland, 1992.

[HJL05] Keijo Heljanko, Tommi A. Junttila, and Timo Latvala. Incremental and complete bounded model checking for full pltl. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2005.

[HKQ03] Thomas A. Henzinger, Orna Kupferman, and Shaz Qadeer. From pre-historic to post-modern symbolic model checking. *Formal Methods in System Design*, 23(3):303–327, 2003.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[Hol99] Gerard J. Holzmann. The engineering of a model checker: the gnu i-protocol case study revisited. In *SPIN*, pages 232–244, 1999.

[IEE95] IEEE. Verilog hdl language reference manual. IEEE Draft Standard 1364, Institution of Electrical and Electronics Engineers, 1995.

[INH96] Hiroaki Iwashita, Tsuneo Nakata, and Fumiyasu Hirose. CTL model checking based on forward state traversal. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 82–87, Washington, DC, USA, 1996. IEEE Computer Society.

[ISO88] ISO/IEC. Lotos — a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, 1988.

[JHN03] Toni Jussila, Keijo Heljanko, and Ilkka Niemelä. Bmc via on-the-fly determinization. *Electr. Notes Theor. Comput. Sci.*, 89(4):561–577, 2003.

[JHN05] Toni Jussila, Keijo Heljanko, and Ilkka Niemelä. Bmc via on-the-fly determinization. *STTT*, 7(2):89–101, 2005.

[JN02] Toni Jussila and Ilkka Niemelä. Parallel program verification using bmc. In *In: ECAI 2002 Workshop on Model Checking and Artificial Intelligence*, pages 59–66, 2002.

[Jus04] Toni Jussila. Bmc via dynamic atomicity analysis. In *ACSD*, pages 197–206. IEEE Computer Society, 2004.

[Jus05] Toni Jussila. On bounded model checking of asynchronous systems. Research Report A97, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, October 2005. Doctoral dissertation.

[KLM+98] Robert P. Kurshan, Vladdimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static partial order reduction. In

TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems, pages 345–357, London, UK, 1998. Springer-Verlag.

[KLY02] Robert P. Kurshan, Vladimir Levin, and Hüsnü Yenigün. Compressing transitions for model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 569–581. Springer, 2002.

[KNU73] D.E. KNUTH. *The Art Of Computer Programming : Volume 3 : Sorting And Searching.* Addison-Wesley Ser. in Computer Science & Information Processing. Reading [etc.] : Addison-Wesley Publishing Company, 1973.

[LN00] J. Lind-Nielsen. *Verification of large state/event systems: Ph.D. thesis.* IT-TR. Department of Information Technology, Technical University of Denmark, 2000.

[LN08] Jørn Lind-Nielsen. Buddy - a binary decision diagram package. `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/index.html`, June 10, 2008.

[LNAH+01] Jørn Lind-Nielsen, Henrik Reif Andersen, Henrik Hulgaard, Gerd Behrmann, Kåre J. Kristoffersen, and Kim Guldstrand Larsen. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, 18(1):5–23, 2001.

[LS97] François Laroussinie and Ph. Schnoebelen. Specification in ctl+past, verification in ctl. *Electr. Notes Theor. Comput. Sci.*, 7:161–184, 1997.

[LST03] Flavio Lerda, Nishant Sinha, and Michael Theobald. Symbolic model checking of software. In Byron Cook, Scott Stoller, and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[Mat06] R. Mateescu. *Systèmes temps réel 1 - techniques de description et de vérification*, chapter 5, pages 151–180. IC2 treatise. Lavoisier, 2006.

[MC11] Malcolm Mumme and Gianfranco Ciardo. A fully symbolic bisimulation algorithm. In Giorgio Delzanno and Igor Potapov, editors, *RP*, volume 6945 of *Lecture Notes in Computer Science*, pages 218–230. Springer, 2011.

[MD98] D. Miller and Rolf Drechsler. Implementing a multiple-valued decision diagram package. In *ISMVL*, pages 52–57, 1998.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Min04] Andrew S. Miner. Saturation for a general class of models. In *QEST*, pages 282–291. IEEE Computer Society, 2004.

[Min06] Andrew S. Miner. Saturation for a general class of models. *IEEE Trans. Software Eng.*, 32(8):559–570, 2006.

[MS03] Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Sci. Comput. Program.*, 46(3):255–281, 2003.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[Nal98] Ratan Nalumasu. *Formal Design and Verification Methods for Shared Memory Systems*. PhD thesis, University of Utah, 1998.

[NG96] Ratan Nalumasu and Ganesh Gopalakrishnan. Partial order reduction without the proviso. Technical Report Thecnical Report UUCS-96-008, University of Utah, Department of Computer Science, 1996.

[NG97a] Ratan Nalumasu and Ganesh Gopalakrishnan. A new partial order reduction algorithm for concurrent system verification. In *CHDL'97: Proceedings of the IFIP TC10 WG10.5 international conference on Hardware description languages and their applications : specification, modelling, verification and synthesis of microelectronic systems*, pages 305–314, London, UK, UK, 1997. Chapman & Hall, Ltd.

[NG97b] Ratan Nalumasu and Ganesh Gopalakrishnan. Pv: A model-checker for verifying ltl-x properties. In *Fourth Nasa Langley Formal Methods Workshop*, pages 153–161. Nasa Conference Publication 3356, 1997.

[NG98a] Ratan Nalumasu and Ganesh Gopalakrishnan. A partial order reduction without the proviso. Technical Report Thecnical Report UUCS-98-017, University of Utah, UT, Department of Computer Science, 1998.

[NG98b] Ratan Nalumasu and Ganesh Gopalakrishnan. Pv: An explicit enumeration model-checker. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *FMCAD*, volume 1522 of *Lecture Notes in Computer Science*, pages 523–528. Springer, 1998.

[NG02] Ratan Nalumasu and Ganesh Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3):231–247, 2002.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[Pel96] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *ICSE*, pages 105–118, 1992.

[SB06] Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 149(1):79–96, 2006.

[Str01] Ofer Strichman. Pruning techniques for the sat-based bounded model checking problem. In Tiziana Margaria and Thomas F. Melham, editors, *CHARME*, volume 2144 of

Lecture Notes in Computer Science, pages 58–70. Springer, 2001.

[Val90] A. Valmari. A stubborn attack on state explosion. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 25–42. AMS-ACM, 1990.

[VL93] Bart Vergauwen and Johan Lewi. A linear local model checking algorithm for ctl. In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 447–461. Springer, 1993.

[VP08] José Vander Meulen and Charles Pecheur. Efficient symbolic model checking for process algebras. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596, pages 69–84. LNCS, 2008.

[VP09] José Vander Meulen and Charles Pecheur. Combining partial order reduction with bounded model checking. In *Communicating Process Architectures 2009 - WoTUG-32*, volume 67 of *Concurrent Systems Engineering Series*, pages 29 – 48. IOS Press, 2009.

[VP11a] José Vander Meulen and Charles Pecheur. Combining partial order reduction and symbolic model checking to verify LTL properties. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA FORMAL METHODS 2011*, volume 6617 of *LNCS*, pages 406–421. Springer, 2011.

[VP11b] José Vander Meulen and Charles Pecheur. Milestones: A model checker combining symbolic model checking and partial order reduction. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA FORMAL METHODS 2011*, volume 6617 of *LNCS*, pages 525–531. Springer, 2011.

[WC09] Min Wan and Gianfranco Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *SOFSEM*, volume 5404 of *Lecture Notes in Computer Science*, pages 582–594. Springer, 2009.

[WKS01] Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. Satire: A new incremental satisfiability engine. In *DAC*, pages 542–545. ACM, 2001.

[WW96] Bernard Willems and Pierre Wolper. Partial-order methods for model checking: From linear time to branching time. In *LICS*, pages 294–303. IEEE Computer Society, 1996.

[WYKG08] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2008.

[ZC09] Yang Zhao and Gianfranco Ciardo. Symbolic ctl model checking of asynchronous systems using constrained saturation. In Zhiming Liu and Anders P. Ravn, editors, *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 368–381. Springer, 2009.

[ZC10] Yang Zhao and Gianfranco Ciardo. Symbolic computation of strongly connected components using saturation. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 202–211, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[ZC11] Yang Zhao and Gianfranco Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *ISSE*, 7(2):141–150, 2011.

[ZJC11] Yang Zhao, Xiaoqing Jin, and Gianfranco Ciardo. A symbolic algorithm for shortest eg witness generation. In Zhenhua Duan and C.-H. Luke Ong, editors, *TASE*, pages 68–75. IEEE Computer Society, 2011.