Communicating Process Architectures 2009
Peter Welch, Herman Roebbers and Tobe Announced (Eds.)
IOS Press, 2009

© *2009 The authors and IOS Press. All rights reserved.*

Combining Partial Order Reduction with Bounded Model Checking

José Vander Meulen and Charles Pecheur
Université catholique de Louvain
{jose.vandermeulen, charles.pecheur} @uclouvain.be

Abstract. Model checking is an efficient technique for verifying properties on reactive systems. Partial-order reduction (POR) and symbolic model checking are two common approaches to deal with the state space explosion problem in model checking. Traditionally, symbolic model checking uses BDDs which can suffer from space blow-up. More recently bounded model checking (BMC) using SAT-based procedures has been used as a very successful alternative to BDDs. However, this approach gives poor results when it is applied to models with a lot of asynchronism. This paper presents an algorithm which combines partial order reduction methods and bounded model checking techniques in an original way that allows efficient verification of temporal logic properties (LTL_X) on models featuring asynchronous processes. The encoding to a SAT problem strongly reduces the complexity and non-determinism of each transition step, allowing efficient analysis even with longer execution traces. The starting-point of our work is the Two-Phase algorithm (Namalesu and Gopalakrishnan) which performs partial-order reduction on process-based models. At first, we adapt this algorithm to the bounded model checking method. Then, we describe our approach formally and demonstrate its validity. Finally, we present a prototypal implementation and report encouraging experimental results on a small example.

Introduction

Model checking is a technique used to verify concurrent systems such as distributed applications and communication protocols. It has a number of advantages. In particular, model checking is automatic and usually quite fast. Also, if the design contains an error, model checking will produce a counterexample that can be used to locate the source of the error [?].

In the 1980s, several researchers introduced very efficient temporal logic model checking algorithms. McMillan achieved a breakthrough with the use of symbolic representations based on the use of Ordered Binary Decision Diagrams (BDD) [?]. By using symbolic model checking algorithms, it is possible to verify systems with a very large number of states [?]. Nevertheless, the size of the BDD structures themselves can become unmanageable for large systems. Bounded Model Checking (BMC) uses SAT-solvers instead of BDDs to search errors on bounded execution path [?]. BMC offers the advantage of polynomial space complexity and has proven to provide competitive execution times in practice.

A common approach to verify a concurrent system is to compute the product finite-space description of the processes involved. Unfortunately, the size of this product is frequently prohibitive due, among other causes, to the modelling 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 analyse one of them [?].

This paper presents a technique which combines together the BMC method and the POR method for verifying linear temporal logic properties. We start from the Two-Phase algorithm

(TP) of Namalesu and Gopalakrishnan [**?**]. We merge a variant of TP with the BMC procedure. This allows the verification of models featuring asynchronous processes. 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 contrast, our previous work introduced an algorithm to verify branching temporal logic properties which merges POR and BDD-based model checking [**?**].

To assess the validity of our approach, we start by introducing two methods which can be combined together for transforming computation trees. The POR method captures partial-order reduction criteria [**?**,**?**,**?**]. The idle-extension shows how a finite number of transitions can be added while also preserving temporal logic properties. Then, the Stuttering Bounded Two-Phase (SBTP) reduction is introduced, as a particular instance of a combination of these two methods inspired from TP. Finally, we present how a finite unfolding of SBTP is encoded as a propositional formula suitable for BMC.

The remainder of the paper is structured as follows. Section **??** recalls some background concepts, definitions and notations that are used throughout the paper: bounded model checking, bisimulations and POR. In Section **??**, two transformations of computation trees which preserves CTL^*_X properties are presented, as well as the SBTP algorithm and its transformation to a BMC problem. Section **??** presents the extension of our prototype implementing the BMC of SBTP method. In Section **??**, we present the results obtained by applying our method on a case study. Section **??** reviews related works. Finally, Section **??** gives conclusions as well as directions for future work.

1. Background

1.1. Transitions Systems

A transition system which is a particular class of state machine represents the behavior of a system. A state of the transition systems is a snapshot of the system at a particular time, formally each state is labelled with atomic propositions. The actions performed by the system are modeled by means of transitions between states. Formally each transition carries a label which represents the performed action [**?**] [1]. In the rest of this paper, we assume a set AP of atomic propositions and a set A of transitions. Without loss of generality, the set AP can be restricted to the propositions that appear in the property to be verified on the system.

Definition 1 (Transition System). *Given a set of transitions A and a set of atomic propositions AP, a* transition system *(over A and AP) is a structure $M = (S, T, s_0, L)$ where S is a finite set of states, $s_0 \in S$ is an initial state* [2]*, $T \subseteq S \times A \times S$ is a transition relation and $L : S \to 2^{AP}$ is an interpretation function over states.*

We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in T$. A transition α is *enabled* in a state s iff there is a state s' such that $s \xrightarrow{\alpha} s'$. We write $enabled(s, T)$ for the set of enabled transitions of T in s. When the context is clear, we write $enabled(s)$ instead of $enabled(s, T)$. We assume that T is total (i.e. $enable(s) \neq \emptyset$ for all $s \in S$). A transition α is *deterministic* in a state s iff there is at most one s' such that $s \xrightarrow{\alpha} s'$.

[1]Our treatment differs slightly from [**?**] which views T as a set of (unlabelled) transition relations $\alpha \subseteq S \times S$. Using labelled transitions amounts to the same structures and is mathematically cleaner to define.

[2]For simplicity, s_0 is a single state. All the arguments of this paper can be easily generalized to many initial states (i.e. $S_0 \subseteq S$).

A transition $\alpha \in T$ is *invisible* if for each pair of states $s, s' \in S$ such that $s \xrightarrow{\alpha} s'$, $L(s) = L(s')$. A transition is *visible* if it is not invisible. An execution path of M is an infinite sequence of consecutive transitions steps $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$.

A *computation tree* can be built from a transition system M. $s_0 \in S$ is the root of a tree that unwinds all the possible executions from that initial state [**?**]. The computation tree of M ($CT(M)$) is itself a transition system and is essentially equivalent to M, in a sense that will be made precise below.

1.2. Model Checking

This section briefly introduces model checking. For more details, we refer the reader to [**?**]. Model checking is an automatic technique to verify that a concurrent systems such a distributed application and a communication protocol, satisfies a given property. Intuitively, the system is modeled as a finite transition system, and model checking performs an exhaustive exploration of the resulting state graph to fulfill the verification. If the system violates the property, model checking will generate a counterexample which will help to locate the source of the error.

A common approach to verify a concurrent system is to compute the combined finite-space description of the processes involved. Unfortunately, the size of this combination can grow exponentially, due to all the different interleavings among the executions of all the processes. Partial Order Reduction (POR) techniques reduce the number of interleaving sequences that must be 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 analyse one of them [**?**].

Temporal logic is used to express properties to be verified. In addition to the elements of propositional logic, this logic provides temporal operators for reasoning over different steps of the execution. There are several types of temporal logics such as linear temporal logic (LTL), computation tree logic (CTL), or CTL^* which subsumes both LTL and CTL. For instance, LTL formulæ are interpreted over each execution path of the model. In LTL, $\mathbf{G}\varphi$ (globally φ) says that φ will hold in all future states, $\mathbf{F}\varphi$ (finally φ) says that φ will hold in some future states, $\varphi\ \mathbf{U}\ \psi$ (φ until ψ) says that ψ will hold in some future states and at every preceding states φ holds, and $\mathbf{X}\varphi$ (next φ) says that φ is true in the next state. In this paper we will consider LTL_X, the fragment of LTL without the **X** operator. Similarly, CTL^*_X (resp. CTL_X) is the fragment of CTL^* (resp. CTL) without the **X** operator.

By using temporal logic model checking algorithms, we can check automatically whether a given system, modeled as a transition system, satisfies a given temporal logic property. In the 1980's, very efficient temporal logic model checking algorithms were introduced for these logics [**?**,**?**,**?**,**?**]. For instance, to check if a system M satisfies a LTL property φ, the algorithm presented in [**?**] constructs an automaton B over infinite words named Büchi automaton from the negation of φ [**?**]. Then it searches for violations of φ by checking the executions of the state graph which result from the combination of M and B.

1.3. Bounded Model Checking

In 1992, a step forward was reached by McMillan by using a symbolic approach, based on Binary Decision Diagrams (BDD), to reason on set of states rather than individual states. This technique made it possible to verify systems with a very large number of states [**?**]. However for large models, the size of the BDD structures themselves can become intractable.

In contrast, the Bounded Model Checking (BMC) uses SAT solver instead of BDDs as the underlying computational device [**?**]. The idea of BMC is to characterize an error execution path of length k as a propositional formula, and search for solutions to that formula with 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 $p\ \mathbf{U}\ q \equiv q \vee (p \wedge \mathbf{X}(p\ \mathbf{U}\ q))$ 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. We continue this process until a counterexample is found or any limit is reached.

BMC allow to check LTL properties on a 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 **??**(a)) and paths without such a back-loop (c.f. Figure **??**(b)). The former can be seen as 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 cases, such a prefix π is sufficient to show that a path violates a property f. For instance, let f be the property $\mathbf{G}p$. If π contains a state which does not satisfy p then all paths which start with the prefix π violate $\mathbf{G}p$.

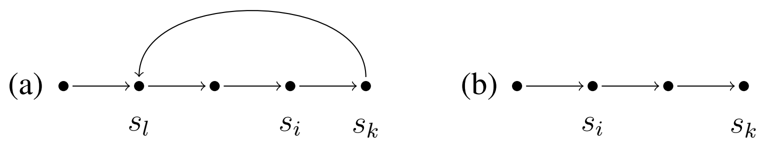

Figure 1. The two cases for a bounded path [?].

The propositional formula $[\![M, \neg f]\!]_k$ which is submitted to the SAT solver is constructed as follows, where f is the LTL property to be verified.

Definition 2 (BMC encoding)**.** *Given a transition system $M = (S, T, s_0, L)$, a LTL formula f, and a bound $k \in \mathbb{N}$:*

$$[\![M, \neg f]\!]_k = [\![M]\!]_k \wedge \Big((\neg L_k \wedge [\![\neg f]\!]) \vee \bigvee_{l=0}^{k} ({}_lL_k \wedge {}_l[\![\neg f]\!]) \Big) \textit{ where}$$

- *$[\![M]\!]_k$ is a propositional formula which represents the unfolding of k steps of the transition relation,*
- *${}_lL_k$ is propositional formula which is true iff there is a transition from s_k to s_l,*
- *L_k is propositional formula which is true iff there exists a l such that ${}_lL_k$,*
- *$[\![\neg f]\!]$ is a propositional formula which is the translation of $\neg f$ when $[\![M]\!]_k$ does not contain any back loop, and*
- *${}_l[\![\neg f]\!]$ is a propositional formula which is the translation of $\neg f$ when $[\![M]\!]_k$ does contain a back loop to state s_l.*

It is shown in [?] that if $M \not\models f$ then there is a $k \geq 0$ such that $[\![M, \neg f]\!]_k$ is satisfiable. Conversely, if $[\![M, \neg f]\!]_k$ has no solutions for any k then $M \models f$ [3].

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 [?]. Intuitively, DPLL performs alternatively two phases. The first one 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

[3]Actually, [?] shows that it is sufficient to look for bounded solutions of $[\![M, \neg f]\!]_k$ up to bound $k \leq K$ which depends on f and M.

1.4. Bisimulation Relations

A bisimulation is a binary relation between two transition systems M and M'. Intuitively, a bisimulation can be constructed between two systems if one can simulate the other and vice-versa. For instance, bisimulation techniques are used in model checking to reduce the number of states of M while preserving some kind of properties (e.g. LTL_X, CTL_X , …). The literature proposes a large number of variants of bisimulation relations [**?**,**?**]. This section describes two kinds of bisimulation relations used in the sequel.

1.4.1. Bisimulation Equivalence

Bisimulation equivalence is the classical notion [**?**], here adapted to transition systems by requiring identical state labellings, which ensures that CTL^* properties are preserved [**?**]. Intuitively, bisimulation equivalence groups states that are impossible to distinguish, in the sense that both have the same labelling and offer the same transitions leading to equivalent states.

Definition 3 (Bisimulation Equivalence)**.** *Let $M = (S, T, s_0, L)$ and $M' = (S', T', s'_0, L')$ be two structures with the same set of atomic propositions AP. A relation $B \subseteq S \times S'$ is a bisimulation relation between M and M' if and only if for all $s \in S$ and $s' \in S$, if $B(s, s')$ then the following conditions hold:*

- $L(s) = L(s')$.
- *For every state $s_1 \in S$ such that $s \xrightarrow{\alpha} s_1$ there is a $s'_1 \in S'$ such that $s' \xrightarrow{\alpha} s'_1$ and $B(s_1, s'_1)$.*
- *For every state $s'_1 \in S'$ such that $s' \xrightarrow{\alpha} s'_1$ there is a $s_1 \in S$ such that $s \xrightarrow{\alpha} s_1$ and $B(s_1, s'_1)$.*

M and M' are bisimulation-equivalent *iff there exists a bisimulation relation B such that $B(s_0, s'_0)$.*

In [**?**] it is shown that unwinding a structure results in a bisimulation-equivalent structure. So, we conclude that a computation tree which is generated from a model M is bisimulation-equivalent to M. Furthermore, bisimulation equivalence preserves CTL^* properties, as shown in [**?**].

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

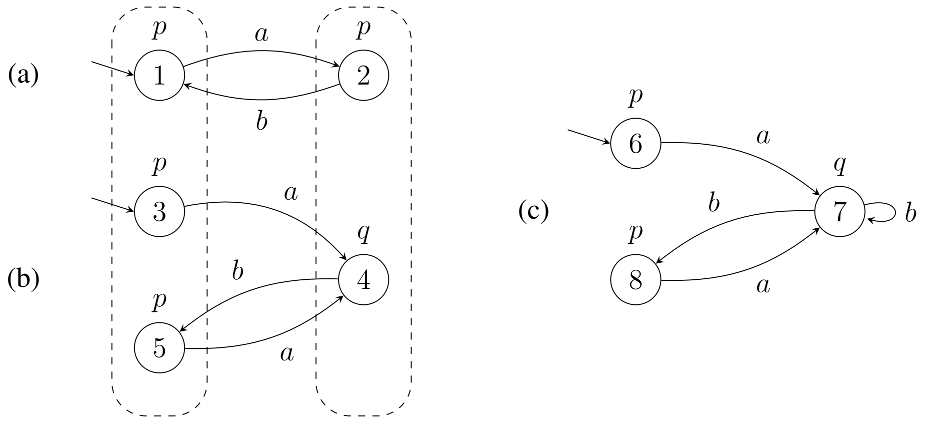

Figure 2. Bisimilar and nonbisimilar structures

1.4.2. The Visible Bisimulation

Visible bisimulation is a weaker equivalence that only preserves CTL^*_X properties, and thus also CTL and LTL properties. Our POR methods preserve visible bisimilarity and therefore those logics. Intuitively, the visible bisimulation associates two states s and t 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 t's future.

Definition 4 (Visible Bisimulation [**?**])**.** *A relation $B \subseteq S \times S'$ is a* visible simulation *between two structures $M = (S, T, s_0, L)$ and $M' = (S', T', s'_0, L')$ iff $B(s_0, s'_0)$ and for every $s \in S$, $s' \in S$ such that $B(s, s')$, the following conditions hold:*

1. $L(s) = L'(s')$
2. *Let $s \xrightarrow{a} t$. There are two cases:*
   - *a is invisible and $B(t, s')$, or*
   - *there exists a path $s' \xrightarrow{c_0} s'_1 \xrightarrow{c_1} \cdots \xrightarrow{c_{n-1}} s'_n \xrightarrow{a'} t'$ in M', such that $B(s, s'_i)$ for $0 < i \leq n$ and c_i is invisible for $0 \leq i < n$. Furthermore, if a is visible, then $a' = a$. Otherwise, a' is invisible.*
3. *If there is an infinite path $s = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ in M, where all a_i are invisible and $B(s_i, s')$ for $i \geq 0$, then there exists a transition $s' \xrightarrow{c} t'$ such that c is invisible, and for some $j > 0$, $B(s_j, t')$*

B is a visible bisimulation *iff both B and B^{-1} are visible simulations. M and M' are visibly-bisimilar iff there is a visible bisimulation B.*

Figure **??** (a) and **??** (b) are visibly-bisimilar. To see this, we construct the relation which put together states of Figure **??** (a) and states of Figure **??** (b) that are linked by a dashed line together. The action a and b can be executed in any order leading to the same result, from the standpoint of verification. Figure **??** (a) and Figure **??** (c) are not visibly-bisimilar, the node 12 in Figure **??** (c) does not correspond to any states in Figure **??** (a).

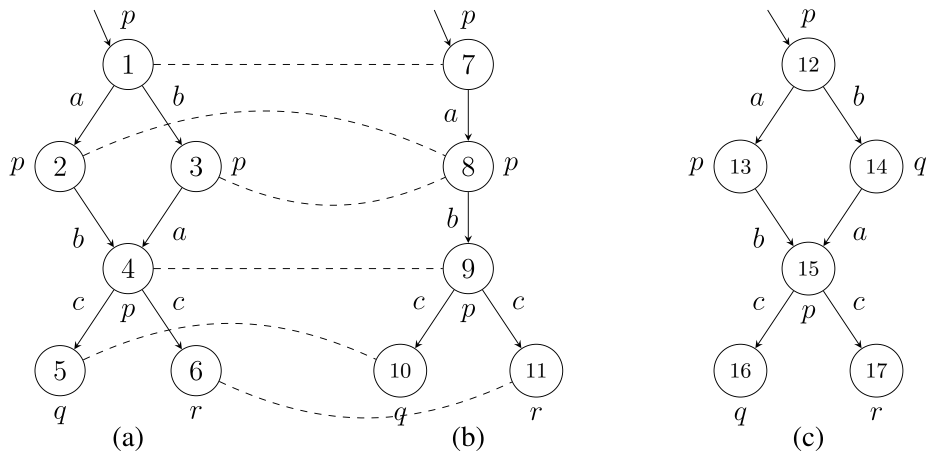

Figure 3. Visibly-bisimilar and not visibly-bisimilar structures

There also exists a weaker bisimulation, called stuttering bisimulation. In general, the POR literature is based on the notion of stuttering bisimulation to reason about POR. In [**?**], it is shown that a visible bisimulation is also a stuttering bisimulation, and also preserves CTL^*_X properties. In general, it is easier to reason about visible bisimulation than about stuttering bisimulation because the former implies an argument about states and the latter implies an argument about infinite paths. Actually, the POR method which is applied in the sequel produces a reduced graph from a model M which is visible-bisimilar to M.

1.5. Partial-Order Reduction

The goal of partial-order reduction methods (POR) is to reduce the number of states explored by model-checking, by avoiding to explore different equivalent interleavings of concurrent events [**?**,**?**,**?**]. Naturally, these methods are best suited for strongly asynchronous programs. Interleavings which are required to be preserved may depend on the property to be checked.

Partial-order reduction is based on the notions of *independence* between transitions and *invisibility* of a transition. Two transitions are *independent* if they do not disable one another and executing them in either order results in the same state.

Intuitively, if two independent transitions α and β are invisible w.r.t. the property f that one wants to verify, then it does not matter whether α is executed before or after β, because 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. In effect, POR amounts to exploring a reduced model $M' = (S', T', s_0, L)$ with $S' \subseteq S$ and $T' \subseteq T$. In practice, classical POR algorithms [**?**,**?**] execute a modified depth-first search (DFS). At each state s, an adequate subset $ample(s)$ of the transitions enabled in s are explored. To ensure that this reduction is adequate, that is, that verification results on the reduced model hold for the full model, $ample(s)$ has to respect a set of conditions, based on the independence and invisibility notions previously defined. In some cases, all enabled transitions have to be explored. The following conditions are set forth in [**?**,**?**]:

C0 $ample(s) = \emptyset$ if and only if $enable(s) = \emptyset$.

C1 Along every path *in the full state graph* that starts at s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first.

C2 If $ample(s) \neq enabled(s)$, then all transitions in $ample(s)$ are invisible.

C3 A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in $ample(s)$ on the cycle.

On finite models, conditions C0, C1, C2 and C3 are sufficient to guarantee that the reduced model preserves properties expressed in LTL_X. On infinite models (such as computation trees) condition C3 must be rephrased as the following condition, which intuitively states that all the transitions in $enabled(s)$ will eventually be expanded.

C3b An infinite path is not allowed if it contains a state in which some transition α is enabled, but is never included in $ample(s)$ on the path.

In order to demonstrate that C0, C1, C2 and C3b preserve LTL_X properties, a similar argument as the one presented in [**?**] can be used. The only difference is the method applied for demonstrating the Lemma 28 of [**?**]. This Lemma can be demonstrated by using condition C3b instead of condition C3.

Ensuring preservation of branching temporal logics requires an additional constraint which is significantly more restrictive [**?**]:

C4 If $ample(s) \neq enabled(s)$, then $ample(s)$ contains only one transition that is deterministic in s.

When conditions C0 to C4 are satisfied, [**?**] shows that there is a visible bisimulation between the complete and reduced models, which ensures preservation of CTL^*_X properties (and thus CTL_X and LTL_X). The same argument can be used to demonstrate that there is also a visible bisimulation between the full and the reduced state graph, when conditions C0, C1, C2, C3b, and C4 are satisfied.

Conditions C1 and C2 depend on the whole state graph and are not directly exploitable in a verification algorithm. Instead, one uses sufficient conditions, typically derived from the structure of the model description, to safely decide where reduction can be performed.

1.6. Process Model

In the sequel, we assume a process-oriented modeling language. We define a *Process Model* as a refinement of a transition system:

Definition 5 (Process Model)**.** *Given transition system* $M = (S, T, s_0, L)$*, a process model consists of a finite set* P *of* m processes p_0, p_1, ..., p_{m-1}. *For each* p_i*, we define safe deterministic actions* $A_i \subseteq A$ *and safe deterministic transitions* $T_i = T \cap (S \times A_i \times S)$ *such that for all* $a \in A_i$*,* a *is invisible, and for all* $s \in S$*:* $ample(s) = enable(s, T_i) = \{a\}$ *satisfies conditions C1 and C4.*

All T_i contain only safe deterministic transitions. Given a state s and a T_i, s is safe deterministic w.r.t. T_i if and only if $enable(s, T_i) \neq \emptyset$.

For instance, suppose a concurrent program S composed of a finite number of thread(s) m. Each thread has exclusive access to some local variables, as well as some global variables that all threads can read or write. This program can be translated into a process model M. The translation procedure may translate each thread of S into a process p_i. In particular, a (deterministic) instruction of p_i that affects only a local variable `x` (e.g. `x = 3`) will meet the conditions of Definition **??** and can be modelled as a safe determinisitc action $a_{x=3} \in A_i$. Indeed, all transition $s \xrightarrow{a_{x=3}} t$ resulting from the execution of that instruction will be safe deterministic. Thus, $ample(s) = enable(s, T_i) = \{a_{x=3}\}$ is a valid ample set for POR.

1.7. The Two-Phase Approach to Partial Order Reduction

This section presents the Two-Phase algorithm (TP) which was firstly introduced in [**?**]. Starting from a model M, it generates a reduced model M' which is visible-bisimilar to M. It is a variant of the classical DFS algorithm with POR [**?**,**?**]. It alternates between two distinct phases:

- Phase-1 only expands safe deterministic transitions considering each process at a time, in a fixed order. As long as a process is deterministic, the single transition that is enabled for that process is executed. Otherwise, the algorithm moves on to the next process. After expanding all processes, the last reached state is passed on to Phase-2.
- Phase-2 performs a full expansion of the state resulting from the Phase-1, then applies Phase-1 recursively to all reached states.

To avoid postponing a transition indefinitely, at least one state is fully expanded on each cycle in the reduced state space. Such an indefinite postponing can only arise within Phase-1. It is handled by detecting cycles within the current Phase-1 expansion. When such a cycle is detected, the algorithm moves to the next process or to Phase-2.

As shown in [**?**], the Two-Phase algorithm produces a reduced state space which is visible-bisimilar to the whole one and therefore preserves CTL^*_X properties. This follows from the fact that TP is a classical DFS algorithm with POR and that $ample(s)$ meets conditions C0 to C4 of Section **??**.

2. The Stuttering Bounded Two-Phase Algorithm

This section presents a variant of the two-phase approach to partial-order reduction, called the *Stuttering Bounded Two-Phase* method (SBTP).

In contrast to the original TP, which performs Phase-1 partial expansions as long as possible, our SBTP method imposes a fixed number n of Phase-1 expansions for each process. If less than n successive steps are enabled for some process, invisible *idle* transitions are performed instead. Figure **??** illustrates the resulting computation tree, for two processes with $n = 3$ transitions each.

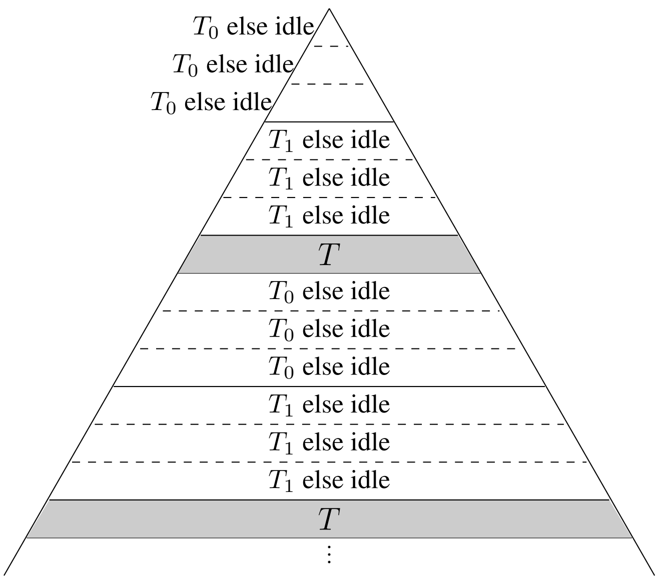

Figure 4. $SBTP(M, 3)$ with two processes and $n = 3$.

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 as a bounded model-checking problem using SAT solvers.

We consider computation trees (CTs) rather than general transition systems. 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 naturally with the SAT-based bounded model-checking approach, which does not attempt to prevent exploring the same state several times on the same path, as opposed to conventional enumerative model-checkers.

To precisely define the SBTP approach, we first characterize a broad class of derived CTs reduced according to partial-order criteria and extended with (finite chains of) idle transitions, and show that they are visible-bisimilar to the CT they derive from. Then we define the CT corresponding to the SBTP method we just outlined, as a particular instance of this class of derived CTs. Finally, we express a constraint system whose solutions are (finite or infinite periodic) bounded execution paths of the CT produced by SBTP.

2.1. Transforming the Computation Tree

This section presents two classes of derived computation trees that are visible-bisimilar to a given computation tree CT: *partial-order reductions* (POR) of CT, which removes states and transitions according to POR criteria, and *idle-extensions* of CT, which adds (finitely many) idle transitions to each state.

Both derivations can be combined: indeed, given an initial CT we can successively derive CT' as a POR of CT then CT'' as an idle extension of CT'. By transitivity of equivalence, we know that CT'' is visible-bisimilar to CT.

Partial-Order Reduction of CTs The definition of POR on computation trees is a straight application of the criteria quoted in Section **??**.

Definition 6 (POR). *Given $CT = (S, T, s_0, L)$ and $CT' = (S', T', s_0, L)$ two computation trees such that $S' \subseteq S$ and $T' \subseteq T$, CT' is a* partial-order reduction *(POR) of CT if and only if $ample(s)$ respects the conditions C0, C1, C2, C3b and C4 from Section* **??** *over CT, where for all s in S, $ample(s) = enabled(s, T')$ when $s \in S'$ and $ample(s) = enabled(s, T)$ otherwise*[4].

Theorem 1. *If CT' is a partial-order reduction of CT, then $CT' \approx CT$.*

Proof. This can be demonstrated by constructing a visible bisimulation between M and M'. The relation $\sim \subseteq S \times S$ is defined such that $s \sim s'$ iff there exists a path $s = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n_1}} s_n = s'$ such that a_i is invisible and $\{a_i\}$ satisfies C1 from state s_i for $1 \leq i < n$. It was shown in [**?**] that the relation $\approx = \sim \cap (S \times S')$ is a visible bisimulation between M and M'. □

Idle-Extension of CTs The idle-extension consists in adding a finite (possibly null) number of idle transitions on states of CT, giving CT'. Intuitively, an idle transition is a transition which does nothing and so does not modify the current state.

Definition 7 (Idle-Extension). *Given a computation tree $CT = (S, T, s_0, L)$, an* idle-extension *of CT is a computation tree $CT' = (S', T', s_0, L)$ over an extended set of transitions $A \cup \{idle\}$, with $S' \supseteq S$ and such that for all $s \in S$ there is a finite sequence $s = s_0 \xrightarrow{idle} s_1 \xrightarrow{idle} \cdots \xrightarrow{idle} s_n$ in CT' where:*

- *$s_1, \dots, s_n$ are new states not in S,*
- *$L(s_1) = \cdots = L(s_n) = L(s)$,*
- *$idle$ is the only enabled transition in $s_0, \dots, s_{n-1}$,*
- *for all $s \xrightarrow{a} t$ in CT we have $s_n \xrightarrow{a} t$ in CT'.*

We write $s \xrightarrow{idle*} s_i$ when such a sequence exists and call s_i an idle-successor *of s and s the* idle-origin *of s_i.*

Note that the $idle$ transition is invisible according to this definition. Since the idle-extension is a tree, idle-successors are never shared between multiple idle-origins.

Theorem 2. *If CT' is an idle-extension of CT, then $CT' \approx CT$.*

Proof. Let $CT = (S, T, s_0, L)$ and $CT' = (S', T', s_0, L)$. We define $B \subseteq CT \times CT'$ such that $B(s, s')$ iff s' is an idle-successor of s (including s itself). We will prove that B is a visible bisimulation between CT and CT'. First, obviously we have $B(s_0, s_0)$. Next, we consider s, s' such that $B(s, s')$ and check that the three conditions of Definition **??** are satisfied both ways. By definition of B, s' is an idle-successor of s.

1. $L(s) = L(s')$ by Definition **??**.
2. If $s \xrightarrow{a} t$ in CT, then there is $s' \xrightarrow{idle*} s'' \xrightarrow{a} t$ in CT', with $B(t, t)$.
   Conversely, if $s' \xrightarrow{a} t'$ in CT' then either $a = idle$, which is invisible, and t' is another idle-successor of s so $B(s, t')$, or $a \neq idle$, in which case s' is the last idle-successor of s and $s \xrightarrow{a} t'$ in CT, with $B(t', t')$.
3. Suppose that there exists an infinite path $s \xrightarrow{a_1} t_1 \xrightarrow{a_2} t_2 \cdots$ in CT, where all a_i are invisible and $B(t_i, s')$ for all t_i. Then s' is a shared idle-successor of all t_i, which is impossible according to Definition **??**.

[4]The case where $s \notin S'$ is for technical soundness only.

Conversely, suppose that there exists an infinite path $s' \xrightarrow{a_1} t'_1 \xrightarrow{a_2} t'_2 \cdots$ in CT', where all a_i are invisible and $B(s, t'_i)$ for all t'_i. Then all t'_i are idle-successors of s, which is again impossible according to Definition **??**.

□

2.2. The Stuttering Bounded Two Phase Computation Tree

In order to accelerate the SAT procedure, we want to consider a modified computation tree of a model such that the same (possibly partial) transition relations are applied to all states at a given depth across the tree.

This result can be obtained by applying Stuttering Bounded Two Phase (SBTP) which is a variant of the Two-Phase algorithm (TP). For the simplicity of the arguments, the method presented in this Section and in Section **??** considers only the case of finite traces without back-loops (c.f. Section **??**). Section **??** explains how to reason about back-loops.

We consider a process model $M = (S, T, s_0, L)$ with m processes p_0, p_1, …, p_{m-1} (c.f. Section **??**) SBTP's Phase-1 expands exactly n deterministic transitions of p_0, then n deterministic transitions of p_1, …, then n deterministic transitions of p_{m-1} ($n \in \mathbb{N}$). If less than n safe deterministic transitions are allowed, then idle transitions are performed instead. After Phase-1, a Phase-2 expansion occurs even if there are safe deterministic transitions remaining. The computation tree produced by $SBTP(M, n)$ is defined in Listing **??**, where $BCT(s, t, i)$ computes the transition relation from state t at depth i using transitions from state s.

```
$SBTP(M,n) = (S',T',s_0,L)$ where

$c = m \cdot n + 1$,
$T' = BCT(s_0, s_0, 0)$,
$BCT(s,t,i) =$
if $n \cdot p \leq i \bmod c \leq n \cdot (p+1) \wedge s \xrightarrow{a} s' \in T_p$ then
    $\{t \xrightarrow{a} s'\} \cup BCT(s', s', i+1)$
else if $n \cdot p \leq i \bmod c \leq n \cdot (p+1) \wedge enable(s, T_p) = \emptyset$ then
    $\{t \xrightarrow{idle} t'\} \cup BCT(s, t', i+1)$ where $t'$ is a idle-successor of $s$
else
    // $p = M$
    $\bigcup_{(s,a,s') \in T} \{\{t \xrightarrow{a} s'\} \cup BCT(s', s', i+1)\}$, and

$S' = \{s \mid s$ is reachable from $s_0$ using $T'\}$
```

Listing 1: SBTP

It is easily seen that the computation tree produced by SBTP is an idle-extension of a partial order reduction of $CT(M)$, and is therefore visible-bisimilar to $CT(M)$.

We 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 **??**(b) illustrates the result of applying one full cycle of SBTP to the CT of Figure **??**(a), with two processes and $n = 3$. The gray arrows of Figure **??**(a) represent transitions which are ignored by Phase-1.

2.3. Applying Bounded Model Checking to SBTP

This section describes the actual bounded model checking problem used in our approach. This problem encodes bounded executions of the SBTP computation tree defined in the previous section.

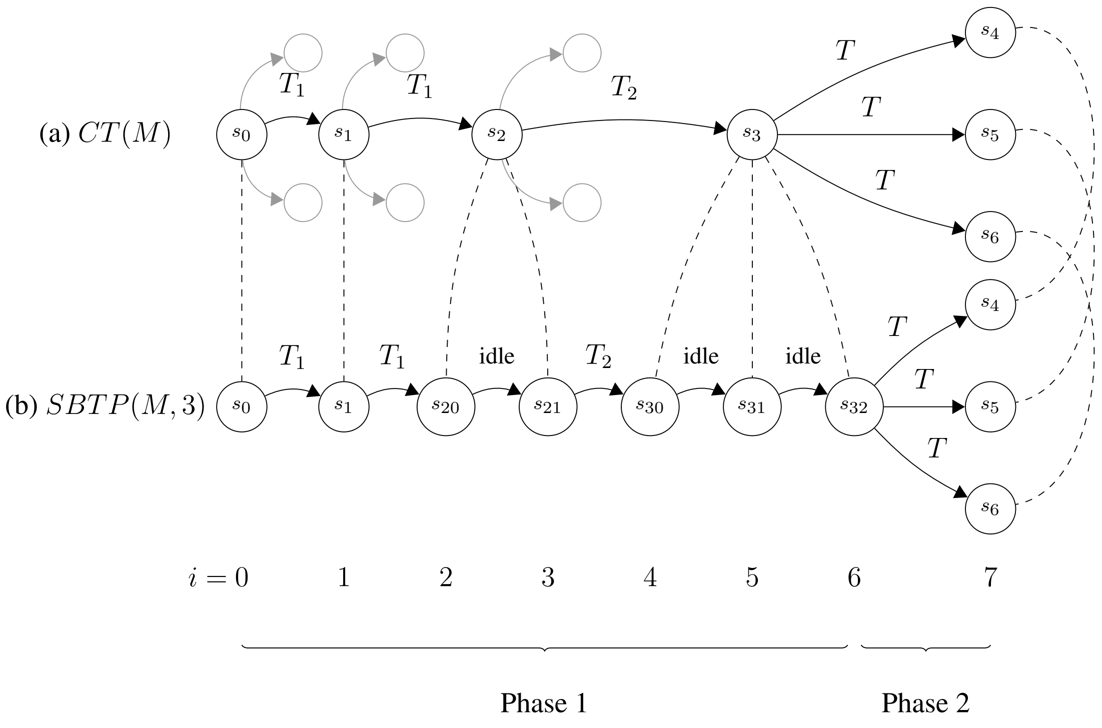

Figure 5. $CT(M)$ vs $SBTP(M,3)$, if s and s' are linked by a dashed line then $s \approx s'$ and $s' \approx s$.

Given a process model M with m processes, a LTL_X property f, and $n, k \in \mathbb{N}$, our approach uses a variant of the method presented in [**?**] to create a propositional formula $[\![M, \neg f]\!]^{SBTP}_{k,n}$. Contrary to the classical bounded model checking methods which uses 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 T used in Phase-2. The others, used in Phase-1, only contain for each process p_i, the safe deterministic transitions of p_i and idle transitions on states where no such safe deterministic transitions are enabled. We denote these relation transitions by T_i^{idle}. Given two states s, t and an action a , $T_i^{idle}(s,a,t)$ if and only if either $enable(s, T_i) = \{a\}$ and $s \xrightarrow{a} t$, or $enable(s, T_i) = \emptyset$ and $a = idle$. Given the number of processes m and parameter n, we know which phase is used in the unfolding process at each depth i of the unfolding process. Furthermore, if Phase-1 is expanded at i, we know which process is being unfolded (c.f. Figure **??**). The transition relation $T_n(i, s, a, s')$ expanded at level i is defined as follows:

Definition 8 ($T_n(i, s, a, s')$)**.** *Given $M = (S, T, s_0, L)$ with m processes $p_0, p_1, \ldots, p_{m-1}$. Let $c = m \cdot n + 1$ the number of steps of a cycle: n Phase-1 steps for each of the m processes plus one Phase-2 step, $i \in \mathbb{N}$, $s, s' \in S$, and $a \in A$:*

$$T_n(i, s, a, s') := \begin{cases} T(s, a, s') & \textit{if } i \bmod c = m \cdot n \textit{ (Phase-2)} \\ T_j^{idle}(s, a, s') & \textit{where } j = (i \bmod c) \operatorname{div} n \textit{ otherwise (Phase-1)} \end{cases}$$

We are able to apply POR on bounded model checking by making use of the previous definition into Definition **??** which translate a transition system and a LTL property into a propositional formula:

Definition 9 (SBTP encoding)**.** *Let M be a process model which contains m processes, f be a LTL_X property and $n, k \in \mathbb{N}$:*

$$[\![M, \neg f]\!]_{k,n}^{SBTP} := I(s_0) \wedge \bigwedge_{i=0}^{i=k-1} T_n(i, s_i, a, s_{i+1}) \wedge {}_lL^k \wedge [\![\neg f]\!]_k$$

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

Theorem 3. *Let M be a process model with m processes, f be an LTL_X formula, and $n \in \mathbb{N}$. There exists k $\geq$ 0 such that $[\![M, \neg f]\!]_{k,n}^{SBTP}$ if and only if $M \not\models f$*

Proof.

$$\begin{array}{lll}
 & \exists k : [\![M, \neg f]\!]_{k,n}^{SBTP} \text{ is satifiable} & \\
\iff & \exists k : [\![SBTP(M, n), \neg f]\!]_k \text{ is satifiable} & \big([\![M, \neg f]\!]_{k,n}^{SBTP} = [\![SBTP(M, n), \neg f]\!]_k\big) \\
\iff & \exists k : SBTP(M, n) \not\models_k f & \text{(by validity of BMC (c.f. Theorem 2 of [?]))} \\
\iff & SBTP(M, n) \not\models f & \text{(by validity of BMC (c.f. Theorem 1 of [?]))} \\
\iff & M \not\models f & (SBTP(M, n) \approx CT(M) \approx M)
\end{array}$$

□

$[\![M, \neg f]\!]_{k,n}^{SBTP}$ is well suited for the DPLL algorithm in the sense that the Phase-1 transition T_j^{idle} produces mostly efficient unit propagation with little backtracking. Suppose that we want to find a satisfying assignment for the path $s_0 \xrightarrow{a_0} s_1 \cdots$ and that s_0 is a deterministic state. Once the variable's values of s_0 are completely decided, the variable's values of s_1 can be completely inferred by the propagation unit phase. Because s_0 is a deterministic state, there is exactly one possibility for the variable's value of s_1.

2.4. BMC with the back-loops

This section shows why paths with back-loops invalidate the arguments of Section **??**, and how to extend those arguments of Section **??** to deal with back-loops.

Figure **??** represents a path π which contains a back-loop. It is easy to see that this finite loop induces an infinite path which does not belong to $SBTP(M, 2)$. This path belongs to the computation tree presented in Figure **??**. All the execution paths start with a prefix of length $k_1 = 3$ of the form $s_0 \xrightarrow{T_1^{idle}} s_1 \xrightarrow{T_1^{idle}} s_2 \xrightarrow{T_2^{idle}}$, followed by an infinite expansion of the loop of length $k_2 = 6$: $s_i \xrightarrow{T_2^{idle}} s_{i+1} \xrightarrow{T} s_{i+2} \xrightarrow{T_1^{idle}} s_{i+3} \xrightarrow{T_1^{idle}} s_{i+4} \xrightarrow{T_2^{idle}} s_{i+5} \xrightarrow{T}$.

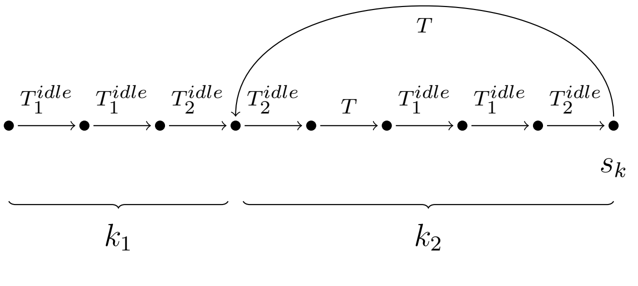

Figure 6. A finite path with a back-loop.

Given a process model $M = (S, T, s_0, L)$ with m processes, and lengths k_1 and k_2, we can build variants of $SBTP(M, n)$ that correspond to the computation tree of Figure **??**.

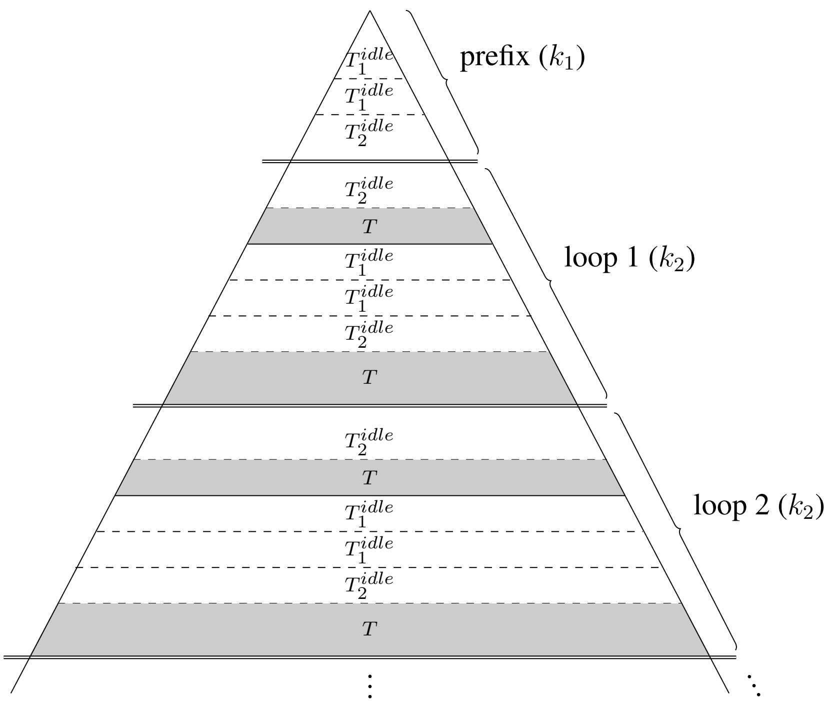

Figure 7. variant of $SBTP(M, n)$.

These variants are still idle extensions of partial order reductions of $CT(M)$, hence visible-bisimilar to M. We can then construct a complete version of $[\![M, \neg f]\!]_{k,n}^{SBTP}$ with back-loops, similar to Definition **??**, that essentially corresponds to the union of those modified SBTP computation trees.

Note that in order to satisfy the condition C3b of Section **??**, the full transition relation T must be used to check whether there exists a back loop or not. If a transition relation T_j^{idle} was used instead, we could have a loop that does not contain a Phase-2 expansion, thus postponing some transitions indefinitely and violating condition C3b.

3. Implementation

We extended the model checker presented in [**?**] to support the BMC over SBTP models. It allows us to describe concurrent systems and to verify LTL_X properties. Our prototype has been implemented with the Scala language [**?**]. We decided to use the Scala language because it is a multi-paradigm programming language, fully interoperable with Java, designed to integrate features from object-oriented programming and functional programming. Scala is a pure object-oriented language in the sense that every value is an object. Scala is also a functional language in the sense that every function is a value.

The model checker defines a language for describing transitions systems. The design of the language has been influenced on the one hand by process algebras and on the other hand by the NuSMV language [**?**]. A model of a concurrent system declares a set of global variables, a set of shared actions and a set of processes. A process p_i declares a set of local variables, a set of local actions and the set of shared actions which p_i is synchronized on. Each process has a distinguished local program counter variable pc. For each value of pc, the behavior of a process is defined by means of a list of action-labelled guarded commands of the form $[a]c \rightarrow u$, where a is an action, c is a condition on variables and u is an assignment updating some variables. Shared actions are used to define synchronization between the processes. A shared action occurs simultaneously in all the processes that share it, and only when all enable it. For each process p_i, we use an heuristic which is based on syntactic information about p_i to compute a safe approximation A_i of the safe deterministic transitions. These conditions are described in [**?**]. We only allow properties to refer the global variables. Intuitively, the safe deterministic transitions are those which perform a deterministic action (e.g. a deterministic assignment) and do not access any global variables or global labels. A more complex heuristic could take into account the variables occurring in the properties being verified to improve the quality of the safe approximation A_i.

The model checker takes a model in this language as input. The number of steps per process in Phase-1 (parameter n) is fixed by the user. To find an error, it applies an iterative deepening, producing a SAT problem corresponding to Definition **??** for increasing depths k. The Yices SMT solver is used to check the satisfiability of the generated formula [**?**]. We decided to use Yices because it offers built-in support for arithmetic operators defined in our language. If a counterexample is found, a trace which violates the property is displayed.

4. Case Study

In order to assess the effectiveness of our method, we applied it to a variant of a producer-consumer system where all producers and consumers contribute on the production of every single item. The model is composed of $2m$ processes: m producers and m consumers. The producers and consumers communicate together via a bounded buffer. Each producer works locally on a piece p, then it waits until all producers terminate their task. Then, p is added to the bounded buffer, and the producers start processing the next piece. When the consumers remove p from the bounded-buffer, they work locally on it. When all the consumers have terminated their local work, an other piece can be removed from the bounded-buffer.

Two properties have been analyzed on this model: P_1 states that the bounded buffer is always empty, and P_2 states that in all cases the buffer will eventually contain more than one piece.

Table **??** and Table **??** compare the classical BMC method and the SBTP 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 of the verification time, we notice that our method significantly outperforms the BMC method for this example. We also notice that SBTP traces are 3.4 to 6.75 times longer. This difference can come from either the addition of the idle transitions, or the considered paths themselves: contrary to BMC, our method does not consider all possible interleavings, thus it is possible that the smallest error traces are not considered.

Table **??** analyses the influence of the number of times Phase-1 is executed for each process (i.e. the 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 one hand, the graph 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 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 cycles is decremented 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 augment until the number of cycles is decremented again.

Table 1. Statistics of property P_1 of the producer-consumer model using BMC approach and SBTP approach with $n = 8$. m is the number of producers (resp. consumers), # states is the state space size, 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 with 8 hours.

		BMC property P_1			SBTP property P_1			
m	# states	k	TIME (sec)	MEM (MB)	k	# cycles	TIME (sec)	MEM (MB)
1	1,059	10	10	29	34	2	7	30
2	51,859	18	44	41	66	2	8	49
3	3,807,747	26	11,679	65	98	2	16	85
4	$\approx 10^8$	—	—	—	130	2	31	122
5	$\approx 10^{10}$	—	—	—	162	2	43	169
6	$\approx 10^{12}$	—	—	—	194	2	57	224
7	$\approx 10^{14}$	—	—	—	226	2	77	288

Table 2. Statistics of property P_2 of the producer-consumer model using BMC approach and SBTP approach with $n = 8$. m is the number of producers (resp. consumers), # states is the state space size, 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 with 8 hours.

	BMC property P_2				SBTP property P_2			
m	# states	k	TIME (sec)	MEM (MB)	k	# cycles	TIME (sec)	MEM (MB)
1	1,059	26	73	33	153	9	122	96
2	51,859	44	29,898	131	297	9	211	224
3	3,807,747	—	—	—	441	9	401	363
4	$\approx 10^8$	—	—	—	585	9	1,238	680
5	$\approx 10^{10}$	—	—	—	729	9	1,338	983
6	$\approx 10^{12}$	—	—	—	873	9	1,926	1,438
7	$\approx 10^{14}$	—	—	—	1,017	9	4,135	1,618

Table 3. Influence of the 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 Megabyte).

	property P_1				property P_2			
n	k	# cycles	TIME (sec)	MEM (MB)	k	# cycles	TIME (sec)	MEM (MB)
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

5. Related Work

Different approaches have been developed to apply symbolic model checking on asynchronous systems.

In [?], Enders et al. show how to encode a transition relation $T(s, a, s')$ into BDDs. This paper focusses on the ordering of the variables within BDDs. It is well-know that the size of BDDs, and therefore performance of BDD-based model checking, strongly depends on this ordering. In general finding the best variable ordering is a NP-complete problem. The paper presents an heuristic which produces BDDs that grow linearly in the number of asynchronous components according to experimental results.

In [?], Gerth et al. show how to perform partial order reduction in the context of process algebras. They show that condition C0 to C4 of Section **??** can be applied to produce a reduced structure that is branching-bisimilar, and hence preserve Hennessy-Milner logic [?].

Other approaches combine symbolic model checking and POR to verify different classes of properties. In [?], Alur et al. transform an explicit model checking algorithm performing partial order reduction. This algorithm is able to check invariance of local properties. They start from a DFS algorithm to obtain a modified BFS algorithm. Both expand an ample set of transitions at each step. In order to detect the cycles, they assume pessimistically that each previously expanded state might close a cycle. In [?], Abdulla et al. present a general method for combining POR and symbolic model checking. Their method can check safety properties either by backward or forward reachability analysis. So as to perform the reduction, they employ the notion of commutativity in one direction, a weakening of the dependency relation which is usually used to perform POR. In [?], Kurshan et al. introduce a partial order reduction algorithm based on static analysis. They notice that each cycle in the state space is composed of some local cycles. The method performs a static analysis of the checked model so as to discover local cycles and set up all the reductions at compile time. The reduced state space can be handled with symbolic techniques.

This paper complements our previous work which combined symbolic model checking and partial order reduction [?]. That work introduces the FwdUntilPOR algorithm that combines two existing techniques to provide symbolic model checking of a subset of CTL on asynchronous models. The first technique is the ImProviso algorithm which efficiently merges POR and symbolic methods [?]. It is a symbolic adaptation of the Two-Phase algorithm. The second technique is the forward symbolic model checking approach applicable to a subset of CTL [?]. Contrary to FwdUntilPOR which checks CTL_X properties using BDD-based model checking, our method deals with LTL_X properties using a SAT solver.

In [?], Jussila presents three improvement to apply bounded model checking to asynchronous systems. Jussila considers reachability properties, whereas our method allows the verification of LTL_X properties. The *partial order semantics* replaces the standard interleaving execution model with non-standard models allowing the execution of several independent actions simultaneously. Then, the *on-the-fly determinization* consists to determinize the different components during their composition. This is done creating a propositional formula whose models correspond to the executions from the determinized equivalents of the components. We point out that the state automaton resulting from the determinization of the components are never constructed. Finally, the *merging of local transitions* can be seen as introducing additional transitions to the components. These transitions correspond to the execution of a sequence of local actions. When a transition is added, the component has to contain a path between the transition's source and target states.

The partial order semantics addresses the same problem as we do. Both methods consider a model which contains less execution paths than the original model. 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 approaches are potentially applicable and open interesting directions for further work. However, none of those methods provides a BMC encoding using only a subset of the relation transition at some steps, which has proven to provide important performance gains in our approach.

6. Conclusion

In this paper, we introduced a technique which applies the partial order reduction methods to bounded model checking. It provides an algorithm which is appropriate to the verification of LTL_X properties to asynchronous models. The formulæ produced by this approach allow for more efficient processing by the DPLL algorithm used in BMC, compared to those produced by the conventional bounded model checking approach. These formulæ are obtained by using only a restricted, safe subset of the transition relation based on POR considerations at many steps in the unfolding of the model.

In order to assess the correctness of our method, we define two general procedures for transforming a computation tree CT to a visible-bisimilar one. The partial-order reduction of CT, which reduces CT according to classical POR criteria, and the idle-extension of CT, which adds a finite number of idle transitions to each state. Then, we define the SBTP algorithm which is a particular instance of these transformations. Finally, we present the transformation of SBTP into a bounded model checking problem.

We extended a model checker which is currently under development at our university to support our method. We show on a simple case study that our method achieves an improvement in comparison to the classical bounded model checking algorithm. However, our method need to be tested on a larger range of case studies and to be compared with other methods and tools such as NuSMV [?] or FDR [?]. Furthermore, one could explore how to applied our method to those tools.

Our approach can be extended in the following ways:

- The SBTP algorithm 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 MathSat [?].
- When a partial $T_j^{idle}(s_i, a, s_{i+1})$ is applied, only local variables from p_j are modified, other variables y being constrained to remain the same ($y_i = y_{i+1}$). Based on that we could merge these variables and remove the corresponding constraints. This would amount to parallelizing safe transitions of different processes, approaching the result of Jussila's first method from a different angle.
- The heuristic used to determine the safe deterministic transitions is quite simple. Meanwhile, there exists a large body of literature on this subject. Based on that, we could explore better approximations that result in detecting more safe deterministic states.

Acknowledgements

This work is supported by project MoVES under the Interuniversity Attraction Poles Programme — Belgian State — Belgian Science Policy.

The authors are grateful for the fruitful discussions with Stefano Tonetta, Marco Roveri, and Alessandro Cimatti during a stay at the Fondazione Bruno Kessler. These discussions were the starting point of this work.