Convention CIFRE 413 / 99 — LAAS-CNRS / Twam Informatique

Technical Report:

Avoiding State Explosion: A Brief Introduction to Binary Branching Diagrams and Petri Net Unfoldings

François Taïani[1, 2]
Mario Paludetto[1]
Thierry Cros[2]

[1] Laboratoire d'Analyse et d'Architecture des Systèmes
LAAS-CNRS, 7 av. du Colonel Roche
F-31077 Toulouse CEDEX 4 / France
{francois.taiani, mario.paludetto}@laas.fr

[2] Twam Informatique
Les Espaces de Balma bâtiment 15
16 av. Ch. de Gaulle
F-31138 Balma CEDEX / France
{taiani, tcros}@twam.com

Abstract

The use of formal verification methods has been studied for several decades now as a promising mean to improve software quality, particularly in the context of concurrent systems. Among them, approaches based on finite state-machine representations certainly belong to the most popular. Unfortunately, the state machine representation of a concurrent system grows exponentially in size with the number of its concurrent components. In this technical report we give an overview of two of the most important techniques proposed to fight against this combinatorial state proliferation, commonly known as the state space explosion.

The first one, based on Binary Decision Diagrams [BCM90], is a representative of symbolic approaches. These techniques use compact representations of state regions and transition relationships to reduce the memory requirement of verification algorithms. We first introduce general BDDs in an informal way using a small example. Then we show how those BDDs can be used to code a small state transition system.

The second state reduction technique we present is based on partial unfolding of Petri Nets [McMill95]. It belongs to the partial order reduction methods. Such approaches are based on the idea that in most cases the total interleaving of concurrent actions is not needed to check a given property on a system. In such cases, a total interleaving distinguishes between action sequences and their intermediary state counterparts that are equivalent from the point of view of the checked property. In this report we use a small concurrent system expressed with two state machines to show how the partial unfolding technique can be applied to it. We also introduce the informal concept of "unambiguous past / divergent future" to sum up the "self conflict freedom" and "forward conflict freedom" rules introduced in [McMill95].

1 Document Content

2 Introduction

Formal Verification techniques like model-checking are quite popular in concurrent environments (communication protocols, embedded software, distributed networks). As a matter of fact the quality requirements and the high complexity of these systems makes formal verification particularly interesting in spite of its inherent cost.

Such verification techniques are usually based on a finite state-machine representation of the system they are applied to. Unfortunately, the state machine representation of a concurrent system grows exponentially in size with the number of its concurrent components. This is the well known explosion problem. Because of that, techniques with extensive state spaces representations very rapidly run out of memory when confronted with industrial cases.

Since the beginning of the 90's numerous approaches have be proposed to alleviate this limitation, and allow the verification of larger systems. This technical report gives an introduction to two of those approaches: Binary Branching Diagrams (BDDs) and Partial Unfolding of Petri Nets. Both have been studied in a verification context by McMillan in its thesis [McMill92], and have enjoyed a constant attention by the research community since then.

We first look at BBDs in the next section. Using an example we give an informal description of what BDDs are, and try to point out the motivations underlying their introduction in [Bryant86]. In a second step based on the synthetic paper [BCM90], we explain with another example how BBDs can be used in a verification context to symbolically represent state sets and relationships.

In a second section we address the partial unfolding of Petri Nets, as it was presented in [McMill92] and [McMill95]. Here again we've favored an intuitive presentation of the method over a formal description of its details.

3 Compact Set Representation with Binary Branching Diagrams

3.1 Introduction

A synthetic presentation of the use of BDDs (Binary Decision Diagrams) in Model Checking can be found in [BCM90]: *"Symbolic Model Checking: 10^20 states and beyond"*. This article unifies several former techniques using BDDs in a synthetic method base on the μ-caculus. BDDs themselves were introduced in [Bryant86] to represent boolean functions and check integrated circuit components.

3.2 Principle of BDDs

Primarily, a BDD is a compact and canonical representation of a boolean function based on a decision graph. Canonical means any boolean function has a unique BDD representation. This characteristic is essential for verification, since it provides a direct way to compare two functions with respect to relationships such as equality, implication, or complementarity. Let's note in particular that the usual definition of a boolean function as a composition of boolean operators is not canonical.

For example let's consider the function $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$, where x_1, x_2, and x_3 are boolean variables, with value range {0; 1}. A first naive canonical representation of f could consist in a three level binary tree, each level corresponding to a variable, and each of the $2^3 = 8$ leaves to a possible variable combination. (Figure 1)

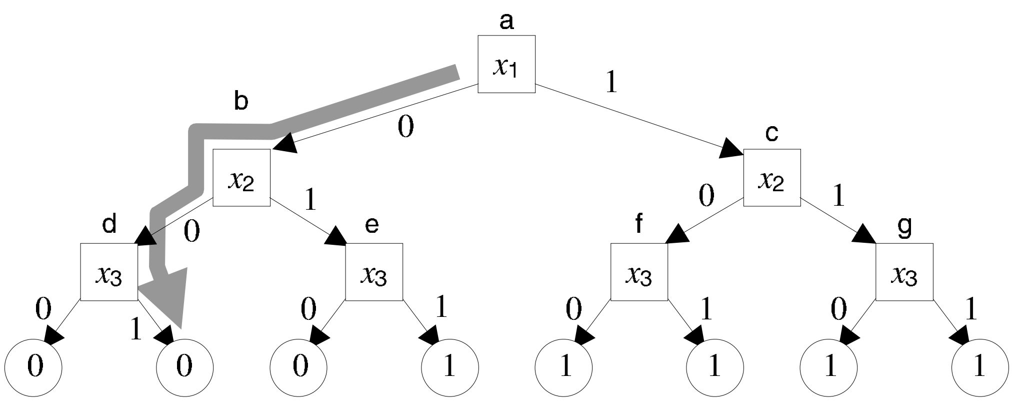

Figure 1: Expanded Tree Representation of $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$

In this representation each "upper" node (i.e. nodes which are not leaves) forms a decision point at which the corresponding variable is tested to decide of the further course of the evaluation. The two alternatives correspond to the subtrees of the node. The computation of f for a given valuation of the variables amounts to a traversal from the root node (**a**) to one of the leaf. Figure 1 illustrates such a computation for the valuation ($x_1= 0$, $x_2= 0$, $x_3= 1$), which yields the value 0. Note that a computation in Figure 1 always goes through the variables x_1, x_2, x_3 in this order.

The BDD representation of f amounts to the elimination of the redundant and irrelevant parts of the "naive" computation tree of Figure 1. Nodes that have no influence on the further progress of the computation (for example nodes **d**, **f**, and **g**) are taken away. Identical subtrees are folded up together. The resulting graph (Figure 2) is far more compact. The computation path of the same valuation ($x_1= 0$, $x_2= 0$, $x_3= 1$) as in Figure 1 has been represented. One sees that that path and hence the computation is shorter on the BDD because the variable x_3 is not taken into account any more. The order of the remaining variables along that path (x_1, and x_2) stays the same as in the computation tree. This is generally true of all the computation paths of the BDD: some variables may be absent, but the remaining ones keep the same ordering.

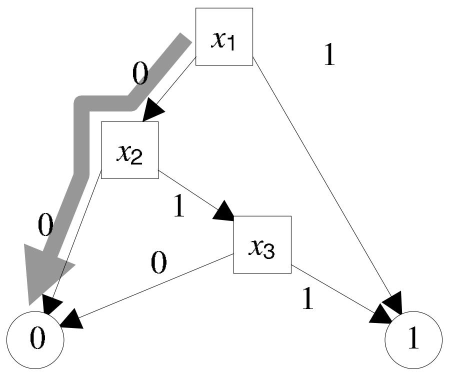

Figure 2: BDD Representation of $f(x_1, x_2, x_3) = x_1 \vee (x_2 \wedge x_3)$

On this short example, we see how efficient BDD storage can be compared to an extensive computation tree. Unfortunately the size of a BDD for a given function strongly depends on the ordering of the input variables. In the example of Figure 2 an ordering (x_2, x_3, x_1) would have yield a bigger BDD. We come back on that issue in more details in the next section.

3.3 Semi-formal Definition BDDs

3.3.1 Basic Structure

For a given ordered set of boolean variables $(x_1,x_2...,x_n)$, a BDD can be defined as a finite acyclic oriented binary graph with labeled nodes and edges (also called a binary decision graph). This graph can be seen as the representation of a partial order on its nodes (because of the acyclicity and the oriented edges). A node **a** is said to be smaller than a node **b** (or **b** bigger than **a**) if there is a path going from **a** to **b** (ordering condition on nodes). A BDD should have a unique minimal element (the graph has a unique “root”).

3.3.2 Labeling of the nodes

The leaves of the graph (i.e. its maximal elements) are labeled with the boolean values **0** and **1** (or **true** and **false**). The other nodes are labeled with one of the boolean variable $(x_1,x_2...,x_n)$. The labeling with the boolean variables is such that if a node **a** is smaller than a node **b**, then the label x_i of **a** should be smaller than the label x_j of **b**: i < j.

3.3.3 Labeling of the edges

The edges of the graph are labeled with **0** and **1** (or **true** and **false**). This labeling corresponds to the resolution to a particular value of the variable that labels the node.

3.3.4 Unicity of BDDs

To insure the canonical form of a BDD (i.e.: for a given ordering of boolean variables and a given boolean function there exists a unique BDD representing that function), a further condition is added: No node should have its 2 edges going to the same node, or its 2 edges going to two "identical" subtrees (i.e. the resolution of the variable to either true or false has no influence or the further computation of the function). This constraint is a sort of “singularity” condition and aims at avoiding redundancy in the BDD. It can be shown that as defined above and with that condition the BDD representation of boolean function is unique [Bryant86].

As mentioned above the fact that BDDs yield a canonical representation of a function is of critical importance for verification purposes since it allows efficient function comparison.

3.3.5 Computation of a function represented by a BDD

The computation of a boolean function with a BDD amounts to a traversal of the BDD from the root to one of the leaves, by choosing on each node the edge corresponding to the value of the variable labeling the node. Because of the ordering condition on nodes, variables are only evaluated once. Because of the singularity condition on BDDs only variables that influence the current computation are evaluated.

3.3.6 Size of a BDD

As mentioned previously, the size of BDDs strongly depends on the ordering of the variables. Finding an optimal ordering is an NP complete problem (i.e. today we need an exponential time in the problem's size to solve it). Even with an optimal ordering, some BDD representations may be exponentially large with respect to the number of variables (no silver bullet here) [Bryant86]. But in many practical and useful cases a good ordering (not necessarily the best one!) of the variables yields a BDD small enough for practical use.

3.3.7 Operations on BDDs

[Bryant86] present algorithms to calculate the BDD of "f_1 **operator** f_2", from the BDDs of the functions f_1 and f_2, for any binary boolean operator such as **and**, **or**, or **xor**. The BDD of a quantified boolean formula, such as $\exists v{:}\, f(v, x_1, \ldots, x_n)$, can be computed with the same algorithm using the following transformation [BCM90]:

$$\exists v{:}\, f(v, x_1 \ldots, x_n) \equiv f(0, x_1 \ldots, x_n) \text{ or } f(1, x_1 \ldots, x_n)$$

3.4 Example of the use of BDDs for model checking

As shown in the precedent part BDDs are a compact representation of boolean functions. If we consider a finite state transition system (*States*, *Transitions*), we may code each state **s** as a boolean string:

$$\begin{array}{ccc} \mathit{States} & \rightarrow & \{0, 1\}^n \\ \mathbf{s} & \rightarrow & (x_1, x_2, \ldots, x_n) \end{array}$$

With this coding a set of states can be coded as a boolean function over the boolean encoding of the states: the boolean function should return 1 whenever the state is in the set and 0 otherwise.

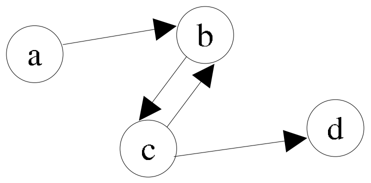

Figure 3: A simple State Transition System

For example, let's consider the set of states {b, c, d} in the State Transition System of Figure 3. If we choose the following boolean encoding of the states:

a	$\rightarrow$	$(x_1 = 0, x_2 = 0)$
b	$\rightarrow$	$(x_1 = 0, x_2 = 1)$
c	$\rightarrow$	$(x_1 = 1, x_2 = 0)$
d	$\rightarrow$	$(x_1 = 1, x_2 = 1)$

The set {b, c, d} will be coded with the following BDD:

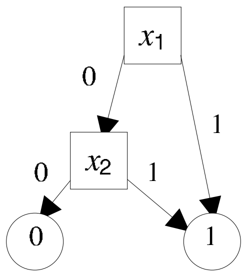

Figure 4: BDD Representation of the set {b, c, d} of the system of Figure 3.

In this particular case the BDD representation is about the same size as the extensive enumeration of the states of the set. In larger examples the BDD of a set may be tremendously smaller than extensive enumeration.

BDDs can be used to store relationships as well. If we consider the state machine of Figure 3, the transition relationship between the states can be expressed as an enumeration of state couples: { (a, b), (b, c), (c, b), (c, d) }. Using the previous state coding this enumeration can be coded in turn with a set of 4-bit boolean strings: { 0001, 0110, 1001, 1011 }. This set can be expressed with a BDD as done before.

Using such representations fix point algorithms can be used to compute the set of states of a state transition system that verifies a given LTL or CTL logic formula. (See [BCM90] for more details.)

4 Partial Unfolding of Safe Petri-Nets

4.1 The interleaving problem

The BDDs presented in the previous section permit avoiding redundancy in systems showing strong structural regularity and low data flows [BCM90]. In concurrent systems, another way to achieve compact representation consists in avoiding explicit interleaving of concurrent actions. Figure 5 shows 2 concurrent state machines that run independently. (I.e. there is no synchronization between them.) They are combined together to form a state machine of the complete system. In the resulting automata, the sequences of actions “**ab**” and “**ba**” are distinguished, as well as the corresponding intermediary states "**1a,2b**" and "**1b,2a**". For most properties (for example deadlock freedom), this distinction is useless since the elementary state machines don't interact with each other. Though it contributes heavily to the state space explosion. More generally this situation occurs when many concurrent problems are loosely synchronized.

A natural way to avoid that interleaving problem consists in using Petri Nets instead of the direct state machines product to represent concurrent processes. Figure 6 shows such a representation. The question is now: how can such a representation be used to check the validity of a logic formula on the system. A naive and brute force answer would consist in constructing the reachability graph of the net, and check the formula on the graph. In the case of Figure 6 this is equivalent with constructing the product machine.

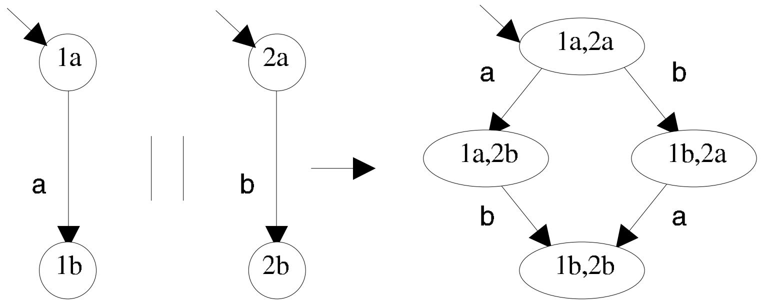

Figure 5: Total interleaving of actions (Here simultaneous actions have been discarded)

[McMill95] proposed an answer to that problem consisting in building a partial unfolding of a safe Petri Net. This partial unfolding is finite, and is particularly compact when many processes run concurrently with a weak synchronization.

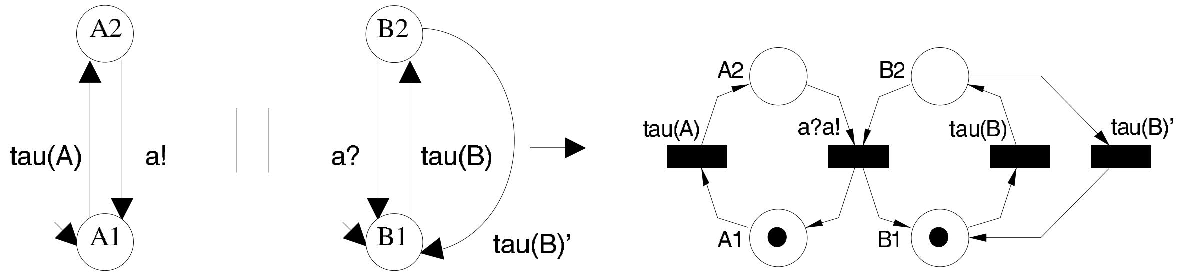

Figure 6: Synchronization product of 2 machines represented as a Petri Nets.

4.2 Unfolding of a Petri Net:

4.2.1 Occurrence Nets:

The unfolding of a Petri Net with respect to an initial marking is a special kind of a Petri Net called Occurrence Net. This Occurrence Net corresponds to a sort of "extensive" representation of all possible runs of the first Petri Net. Figure 7 shows the beginning of the occurrence net of Figure 6. In an Occurrence Net each place (resp. transition) is labeled with the name of a place (resp. transition) of the original PN. Note: In the occurrence net of the Figure 7, the labels are shown with running numbers to distinguish the individual places and transitions.

Intuitively two transitions of an occurrence net with the same label correspond to different "occurrences" (different "firings") of the same transition in the original Petri Net. For example the transitions **tau(A).1** and **tau(A).2** in the occurrence net of Figure 7 correspond to different firings of the same transition **tau(A)** in the Petri Net of Figure 6. To clearly distinguish the transitions of an occurrence net from those of a normal Petri Net, the transitions of an occurrence net are usually call 'events'.

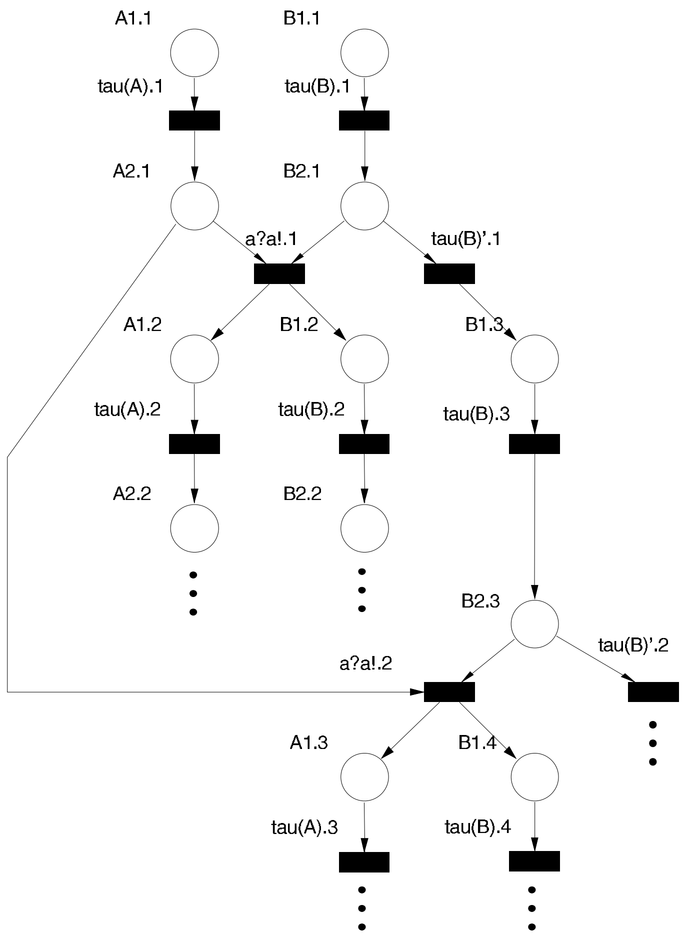

Figure 7: Infinite Unfolding of the Petri net of Figure 6

There exists a kind of "instantiation" relationship between an Occurrence Net and the represented Petri Net. Consequently the input places (resp. output places) of an event of an Occurrence Net should be labeled with the input (resp. output) places of the transitions of the original net represented by this event. For example in Figure 7 the event **a?a!.1** instantiates the transition **a?a!** in Figure 6. Because **a?a!** has 2 input places **A2** and **B2**, and 2 output places **A1** and **B1**, **a?a!.1** has 2 input places **A2.1** and B2.1, which are instances of the places **A2** and **B2**, and 2 output places **A1.2** and **B1.2**, which are instances of **A1** and **B1**.

An Occurrence Net must further have the following properties:

1) Acyclicity:

In a Petri Net each place induces a relationship between its output transitions and its input transitions. In an occurrence net the transitive closure of that relationship should be a partial order (i.e. no cycles should exist in the occurrence net considered as a bipartite oriented graph).

This partial order on events (i.e. transitions) of an occurrence net corresponds to a causality order between the firings of transitions in a run of the original PN. In an occurrence net, when you can travel from one event to another by a sequence of oriented arcs the second event is said to be posterior to the first one. For example in Figure 7 **a?a!.1** is posterior to **tau(A).1** and to **tau(B).1**, and **tau(A).3** is posterior to **tau(B).3**.

2) Existence of an initial marking:

The occurrence net should have a set of minimal events. (The causality order discussed above is said to be well-founded.) Thus it's not possible to infinitely step back on the time line. Starting from any event (i.e. any transition) in the occurrence net, it should always be possible to trace back to a (partial) initial marking.

The input places of the minimal events of an occurrence net correspond to the initial marking of the original PN. For example in Figure 7, the minimal events are **tau(A).1** and **tau(B).1**. Their input places are **A1.1** and **B1.1**, which correspond to the initial marking of the Petri Net of Figure 6.

3) Unambiguous Past:

Considering an event in an occurrence net, there should be a unique unambiguous 'sequence' of partially ordered events (firings) leading to that event from the initial marking. (Sequence is actually not the right word, since a sequence is usually supposed to be totally ordered. The word '*Configuration*' is usually preferred [McMill95].) This requirement is achieved by the forward conflict freedom property [McMill95]: Two different events of an occurrence net should not produce their token in the same place. (Note: in this discussion we're only considering safe PNs.)

4) Divergent Future:

Indeterminism in an occurrence net should lead to disjoint evolutions of the occurrence net. Stated more formally: if 2 events **a** and **b** are in structural conflict (i.e. they have at least one common input place like for example **a?a!.1** and **tau(B).1** in Figure 7), the events and places posterior to **a** should not be comparable with the events and places posterior to **b**. (The "evolution" of the net after **a** — the future of **a** — should not mix with the "evolution" of the net after **b** — the future of **b** —.)

Note that such interferences between conflicting futures may be of two kinds. In the first case, two conflicting futures mix for the first time on a common place. In this situation the past of the common place is not unique and this also breaks property 3 about "*Unambiguous Past*". In the second case the two conflicting futures mix for the first time through a common event. Property 3 does not cover this case: the past of the common event is actually unique. The problem is that that past is not realizable: the common event requires that two evolutions of the system that mutually exclude each other take place. This particular situation is called "*a self conflicting event*" in [McMill95].

The four properties presented here are rather informal. Interested readers are referred to [McMill95] for a more detailed and formal definition of occurrence nets for safe Petri Nets.

4.2.2 Reachability result for Occurrence Nets

An occurrence net is built in such a way, that a place of the original net is reachable from the initial marking if and only if it is instantiated at least once in the occurrence net. Thus partial reachability of a group of places $\{\mathbf{p}_1, \mathbf{p}_2 ..., \mathbf{p}_n\}$ can be tested by adding a transition **t** and a place **p,** with **t** taking $\{\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n\}$ as input places and **p** as output places. Testing the reachability of $\{\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n\}$ amounts to testing the reachability of **p**.

Most of the time occurrence nets are infinite and can't be directly checked for verification. Fortunately a finite prefix of an occurrence can be founded, which has exactly the same property with respect to reachability as the entire occurrence net. The definition of that finite prefix relies on the notion of cut-off point.

4.2.3 Cut-Off Points

Informally a cut off point is an event in an occurrence net which "locally" produces a marking that has "already" been produced by the occurrence net. Before defining more precisely a cut-off point the definition of the backward closure of an event must be defined, as well as the finite marking of a backward closure.

Given an event **a**, its backward closure is the "piece" of occurrence net which contains all the events prior to **a**, as well as the places linking those events (See [McMill95] for a formal definition). For example the backward closure of the event **tau(B).2** in Figure 7 is the subnet consisting of the transitions **tau(A).1**, **tau(B).1**, and **a?a!.1**, and of the places **A1.1**, **B1.1**, **A2.1**, **B2.1**, and **B1.2**.

The final marking of a backward closure is the set of places:

- either in which token are produced by one of the event of the backward closure, but are not consumed by any event of that closure
- or which are part of the initial marking, without being consumed by any event of the backward closure.

(Note: because we only consider safe PN, we may identify markings with set of places.)

For example the final marking of the backward closure of the event **tau(B).2** in Figure 7 is { **A1.2**, **B2.2** }.

A cut-off point of an occurrence net is an event **a** that fulfills one of the following conditions:

- There exists another event **b**, such that the backward closure of **b** is smaller than the one of **a** (with respect to cardinality), and the final marking of the backward closure of **b** is the same as the final marking of the backward closure of **a**.
- The final marking of the backward closure of **a** is the initial marking of the original net.

(Note: these 2 cases could be unified by adding a start transition to the original safe PN, which would produce one token in each place of the initial marking from one unique extra place.)

For example in Figure 7, **a?a!.1** is a cut-off point because its final marking is the initial marking.

4.2.4 Partial Unfolding of a Petri Net

A partial unfolding of a safe Petri-Net is the biggest subnet of the corresponding occurrence net with no cut-off points. Figure 8 shows an example of partial unfolding for the PN of Figure 6. [McMill95] shows that that partial unfolding contains all reachable places, and that all reachable places are in that partial unfolding.

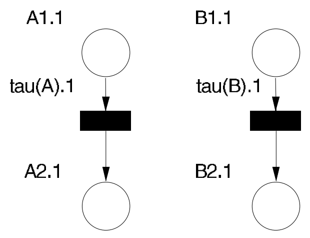

Figure 8: Partial Unfolding of the Petri Net of Figure 6.

4.2.5 Practical Results

The size of the partial Unfolding of a Safe Petri Net may grow exponentially with the number of concurrent processes. Nevertheless in cases with lots of concurrent processes that are only loosely synchronized, far better results can be reached. For example, the dining philosopher problem is treated in [McMill95] as an example, and proves to be very well adapted to this unfolding technique. Though the total size of the product state machine grows exponentially with respect to the number of philosophers, the size of the partial unfolding only grows linearly.

McMillan also gives interesting results for the checking of a Distributed Mutual Exclusion Protocol.

4.2.6 Extensions

Improvements to this technique have been proposed in the meanwhile, for example in [Kondra+96]. Some other approaches have extended the unfolding technique with time and clocks [Bieb+99].

5 Conclusion

In this report we have given a first introduction to Binary Branching Diagrams and Partial Petri Net Unfolding, two important approaches to avoid the state space explosion problem encountered in the formal verification of concurrent systems. In order to keep our presentation as understandable as possible, we have used small examples all along of the report, and we have avoided deep theoretical considerations whenever possible. Readers who would like to learn more on these topics are strongly encouraged to have a look at the references at the end of this article.

BDDs and similar techniques consisting in a compact representation of state space regions and relationships are sometimes referred to as "symbolic" methods. This is because they manipulate intensive representations ("symbols") of large structures, rather than extensive ones.

Partial Petri Net Unfolding on the other hand belongs to partial order techniques such as those presented in [Gode90] or [ProLi91]. Those methods have in common to avoid the total ordering of transitions in a concurrent system, which is usually induced by traditional tree search approaches, but don't reflect properly the nature of concurrency.

Besides symbolic and partial order reductions, other methods have been proposed to fight against the state space explosion problem. Among them on-the-fly model checking is certainly one of the most important. Instead of visiting the entire state space with compact representations like BDDs, or taking advantage of weak synchronization like Partial Order approaches, on-the-fly model checking focuses the search on the state region and the state information relevant to a property. The on-the-fly approach strongly relies on the connections between automata theory and logics. A synthetic approach to it can be found in [BMVW94].

Finally, some approaches have been proposed to combine the best of both worlds (space-efficiency and on-the-fly), for example as in [Henz$^+$96].

6 References

[Arn92] A. ARNOLD, *Systèmes de transitions finis et sémantique des processus communicants,* Collection Études et recherches en informatique, Editions Masson, France, 1992.

[BCM90] J. R. BURCH, E. M. CLARKE, K. L. MCMILLAN, D. L. DILL, L. J. HWANG, *"Symbolic Model Checking: 10^20 states and beyond",* Information and Computation, vol. 8, no. 2, June 1992, pp. 142-70.

[Bieb+99] B. BIEBER, H. FLEISCHHACK, *Model Checking of Petri Nets Based on Partial Order Semantics*, in Proceedings of the 10th International Conf. on Concurrency Theory, Eindhoven, August 1999, Lecture Notes in Computer Science n° 1664, Springer Verlag

[BMVW94] O. BERNHOLTZ, M.Y. VARDI, P.WOLPER, *An automata-theoretic approach to branching-time model checking.* In Computer Aided Verification, Proc. 6th Int. Workshop, volume 818 of Lecture Notes in Computer Science, pages 142--155, Stanford, California, June 1994. Springer-Verlag.

[Bryant86] R. E.BRYANT, *Graph-Based Algorithms for Boolean Function Manipulation,* IEEE Transaction on Computers, Vol. C-35, No. 8, August 1986

[Gode90] P. GODEFROID, *Using partial orders to improve automatic verification methods*. In Workshop on Computer Aided Verification, 1990

[Henz$^+$96] T.A. HENZINGER; O. KUPFERMAN; M.Y. VARDI, *A space-efficient on-the-fly algorithm for real-time model checking*, CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory (Aug.96, Pisa, Italy). Springer-Verlag, Berlin, Germany; 1996, p.514-529

[Kondr+96] A. KONDRATYEV; M. KISHINEVSKY; A. TAUBIN; S. TEN;
A structural approach for the analysis of Petri nets by reduced unfoldings,
Proceedings of 17th International Conference on Application and Theory of Petri Nets. 24-28 June 1996, Osaka, Japan, Springer-Verlag, Berlin, Germany; 1996; p.346-65

[McMill92] K. L. MCMILLAN, *Symbolic Model Checking, An approach to the state explosion problem*, Ph.D. Thesis, Carnegie Mellon University Technical Report CMU-CS-92-131, May 1992.

[McMill95] K. L. MCMILLAN, *A Technique of State Space Search Based on Unfoldings,* Formal Methods in System Design, 6, pp. 45-65, 1995.

[ProLi91] D.K. PROBST AND F.H. LI, *Partial Order model Checking: A guide for the perplexed.* In Third Workshop on Computer Aided Verification, pp.405-416, July 1991.