Combining partial orders and symbolic traversal for efficient verification of asynchronous circuits

Alexei Semenov* and Alexandre Yakovlev†
Department of Computing Science,
University of Newcastle upon Tyne, NE1 7RU, England

Technical Report No. 501

Abstract

Petri nets (PNs) are most adequate for the modelling of the event-triggered behaviour of asynchronous circuits, whose correctness is primarily concerned with freedom from hazards and deadlocks. A recently proposed method for the verification of Petri nets is based on implicit symbolic traversal of the net markings, which often yields better performance than using standard reachability graph analysis. It employs a Binary Decision Diagram (BDD) representation of the boolean functions characterising the state space of the model. This method may, however, suffer from the problem of a bad ordering of the BDD variables. In this paper, we propose an algorithm combining two approaches to PN verification: PN unfolding and BDD-based traversal. We introduce a new application of the PN unfolding method. The results of unfolding construction are used for obtaining the close to optimal ordering of BDD variables. The effect of this combination is demonstrated on a set of benchmarks. The overall framework has been used for the verification of circuits in an asynchronous microprocessor.

Keywords: asynchronous circuits, Binary Decision Diagrams, Petri Net unfolding, variable ordering, verification

1 Introduction

Petri Nets (PNs) are widely used for modelling concurrent systems [11]. The list of applications of PNs is growing rapidly and includes operating systems, asynchronous circuits, distributed systems etc. An event-triggered paridigm in asynchronous circuit operation [19] makes them an excellent testbed for Petri net modelling and analysis techniques.

Once a circuit has been represented in terms of a PN, the designer wants to check if its behaviour meets his/her requirements and expectations. Several methods exist that can be applied for the verification of PN models. These methods can generally be divided into the following categories: the reachability graph (reachability tree) approach; the linear algebraic approach; the state space reduction methods; the partial order approach and the BDD-based symbolic traversal methodswhich has recently acquired special attention. The latter have been shown to be capable of verifying large state spaces at a relatively low cost. This feature has made this method attractive and its application is currently being investigated.

Despite being powerful, the BDD-based verification techniques [14, 7] may suffer from the problem of bad ordering of the BDD variables. Using a proper variable ordering can yield significant reduction in the size of BDD, which in the worst case can be exponential to the number of variables. Usually, it is assumed that the designer, using additional knowledge about the system, can provide proper variable ordering. This obviously cannot be assumed in general.

*Supported by grants from the Research Committee of the University of Newcastle upon Tyne and CVCP.
†Supported by SERC grant No. GR/J 52327

In this paper, we propose a technique combining two approaches to the verification of PN-based models: a specific type of the partial order approach, using PN unfoldings, and the PN reachability symbolic traversal approach. We employ useful properties of the PN unfolding algorithm such as boundedness and safety checks as well as the properties of its signal transition interpretation, relevant to the hazard-freedom of the analyzed circuit. However, in the new framework, we find a new role for the PN unfolding, to produce a more efficient variable ordering.

This paper thus complements the contribution of [14, 7] and [9], and at the same targets its application effort at the verification of circuit designs rather than their initial specifications. We have modelled and analysed a part of the circuitry in an asynchronous microprocessor, AMULET1. This device has been recently developed at the University of Manchester by the group led by Steve Furber. The design has been almost entirely carried out in the style of micropipelines, originally presented by Ivan Sutherland in his Turing Award Lecture in 1989 [19]. Micropipeline circuit components are easily converted into generic fragments of Petri nets, thereby providing a close correspondence between the properties of circuits and those of their net models.

The paper is organised as follows. In section 2 we give a brief introduction to PNs and PN unfoldings. Section 3 describes PN verification methods using the BDD-based traversal technique. Section 4 describes the algorithm for obtaining variable ordering for a BDD generated for the boolean function of the reachability set. Section 5 includes experimental results. The description of the overall verification framework is given in Section 6. Finally, section 7 briefly outlines the application of our modelling techniques and tools for the verification of circuits in asynchronous microprocessor Amulet1.

2 Petri Nets and Unfolding

A marked PN is a tuple $N = \langle P, T, F, m_0 \rangle$ where P and T are non-empty sets of places and transitions respectively, F is a flow relation and m_0 is the initial marking. A PN is represented as a graph with two types of nodes: places (circles) and transitions (bars or boxes). A marking of a PN is denoted with tokens (thick dots). A transition is said to be enabled if all places that input to it (the set of input places is denoted as $\bullet t$) contain at least one token. An enabled transition can fire producing a new marking. The firing of a transition removes one token from each input places and adds one token in to each output place (the set of output places is denoted as $t\bullet$). The set of markings that can be reached from the initial marking via all possible firings of transitions is called the *reachability set*.

A PN is said to be *finite* if sets P and T are finite.

A PN is said to be *k-bounded* (or simply bounded) if there exists such k that at any reachable marking the number of tokens in any place is not greater than k. A 1-bounded PN is called *safe PN*. Further in this paper we will consider only finite safe PNs.

Usually a PN needs to be analysed for having certain properties such as boundedness, deadlock freedom etc. Analysis of a PN can be done by building the reachability set. This can be done in the form of the *reachability tree* denoting all possible sequences of fired transitions (also called *firing sequences*) or in the form of the *reachability graph*. The properties of a PN can be easily verified on the reachability set, e.g. a deadlock is a node in the reachability graph with no arcs going out. The major drawback of building the reachability graph in explicit form is that the number of its nodes can be exponential to the number of transitions in the PN. As a way to analyse the PN behaviour avoiding exponential explosion we can use the unfolding technique.

Analysis based on the PN unfolding (or partial runs of the PN) has been the subject of study in a number of papers (e.g. [13, 6, 10, 17]). One of the most elaborated models is Change Diagrams [6]. Implicitly imposing a FIFO ordering discipline on the token consumption from a place, this technique has been proved to be able to analyse behaviour at the complexity of $O(n^3)$ in the number of events. Unfortunately, Change Diagrams do not have any means of representing non-determinism. An attractive technique introduced recently by McMillan [10] allows the reachability graph for an arbitrary PN to be represented in the form of unfolding. We view the unfolding as a representation keeping concurrency relations between transitions and places of the original PN.

Here we only briefly introduce the unfolding; with some useful notions and notation associated with it.

Formally, the unfolding obtained from a PN, N, is an *occurrence* net $N' = \langle P', T', F', L' \rangle$ where P', T' and F' are the set of places, the set of transitions and the flow relation of the unfolding respectively and $L' : (P' \times T') \rightarrow (P \times T)$ is the labelling function which labels every place and transition of the unfolding as an occurrence of the corresponding place or transition of the original PN.

We can define the following notions representing the causal relations between occurrences.

• The *history of an element* of the unfolding is the minimal backward closed subset of elements of the unfolding including the element itself.

• Two elements of the unfolding are said to be in *conflict* if the histories of these two elements include two distinct transitions such that $\bullet t'_1 \cap \bullet t'_2 \neq \emptyset$. We will say that two elements are *independent* (or concurrent) iff they are not in conflict and are not included into the histories of each other.

The notions of conflict and history of an element allow us to develop an algorithm for constructing the unfolding from an arbitrary PN as follows:

while at least one transition added to UNFOLDING **do**
    for each transition t in T of PN N **do**
        find untried set independent occurrences of $\bullet t$
        if such set of occurrences exists **then do**
            make a copy t' of transition t in the UNFOLDING, copy all output places of t
            connect generated transition t' with appropriate occurrences of input and output places
        end do
    end do
end do

• A *configuration* is the backwards closed non-conflicting subset T'. A special case of configuration is *local configuration* of transition t' which is a minimal configuration including t'. In other words it is a set of all transitions from the history of t'. A local configuration of t' is denoted as $\lceil t' \rceil$.

• We can also define the *final state* of a configuration as a marking of the original PN which is reached by firing transitions whose occurrences are in the configuration. Obviously, a configuration may represent several firing sequences but nevertheless, due to the non-conflictness property, any firing sequence will lead to the same marking. It has been shown [10] that for any reachable marking m of the original PN there exists a configuration in the unfolding such that its final state is equal to m and vice versa. Note that, since a local configuration is a particular case of configuration, it also has the final state.

It is easy to note that the algorithm given above may produce an infinite unfolding, which will make its analysis impossible. Therefore we would like to obtain some truncated form of the unfolding. At the same time we require that all interesting properties of the original PN detectable on the infinite unfolding are preserved.

In [10] an algorithm for obtaining the truncated unfolding was given. We reproduce it below in a somewhat simplified form:

```
while QUEUE is not empty do
    for each transition t in PN N do
        find an untried subset of non-conflicting occurrences of •t
        if such set of occurrences exists then do
            insert t' into the Queue in order of |⌈t'⌉|
        end do
    end do
    pull the first transition t' from the QUEUE
    if there is no other occurrence t'_c with the same final state such that |⌈t'_c⌉| < |⌈t'⌉| then do
        make a copy t' of transition t in the UNFOLDING, copy all output places of t
        connect generated transition t' with appropriate occurrences of input and output places
    end do
end do
```

The above algorithm is guaranteed to terminate because no new transition such that $|\lceil t'_c \rceil| < |\lceil t' \rceil|$ and its successors will be added to the unfolding. This condition of termination is called the *cutoff condition*.

It has been noted in [17] that the truncated unfolding may not contain occurrences of all transitions from the original PN. This can be solved by adding cutoff points to the truncated unfolding although still not exploring the unfolding after them.

As has been shown elsewhere [17], the above algorithm may produce a truncated unfolding which may contain redundant occurrences of transitions. Redundant occurrences are those whose final state is equal to a marking already represented in the unfolding (i.e. there exists another configuration in the unfolding whose final state is exactly the same) and their presence in the truncated unfolding does not add any information. Two reasons have been identified for this: i) some occurrences have final states of their local configurations equal to the final states of other occurrences; ii) some occurrences have final states of their local configurations equal to the final states of other (non-local) configurations. An attempt to avoid the latter requires checking a newly added final state against the final states of all existing configurations. This is equivalent to building the reachability graph. The former redundancy can be avoided for certain classes of the PNs [17, 8] by relaxing the cutoff condition. However, in this work, we are not concerned with the cutoff condition itself and will assume that the right cutoff condition is used for our examples. To work with PNs the algorithm for obtaining the *segment of unfolding* used for our purposes is as follows:

```
while QUEUE is not empty do
    for each transition t in PN N do
        find an untried subset of non-conflicting occurrences of •t
        if such set of occurrences exists then do
            insert t' into QUEUE in the order of |⌈t'⌉|
        end do
    end do
    pull the first transition t' from QUEUE
    make a copy t' of transition t in the UNFOLDING, copy all output places of t
    connect generated transition t' with appropriate occurrences of input and output places
    if there exists another occurrence t'_c such that t' is a cutoff then do
        mark transition t' and its t'• as cutoff point
    end do
end do
```

Naturally, the cutoff condition is chosen for a PN so that the segment of unfolding obtained using the above algorithm fully represents the reachability graph and that the concurrency relations between transitions and places can be determined from such a segment. At the same time the size

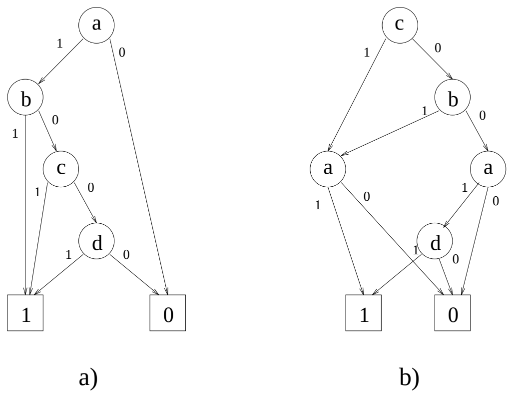

Figure 1: An example of different ordering of variables.

of the unfolding segment and the number of the actually visited markings is often much less (equal for state-machine PNs) than the total size of the reachability graph (see section 5).

In Section 4 we will show how the segment of the unfolding can be used for ordering the variables in the BDD for symbolic traversal of the PN behaviour.

3 Symbolic Traversal of Petri Net State Space

There are several methods of representing logic functions such as truth tables, Karnaugh maps, minterm canonical form or the sum of products form. Operating with these representations is inefficient for relatively big logic functions.

Binary Decision Diagrams (BDDs) were proposed as a means of canonical representation of logic functions in a graphical form. For a detailed introduction to BDDs and their basic manipulations, the reader is referred to [2].

We will only briefly introduce BDDs and explain how they can be used for PN analysis.

Consider a logic function given below:

$$f = a \cdot b + a \cdot c + a \cdot d$$

We can construct a Binary Decision Tree for this function and order its variables $a < b < c < d$. By operations on the Binary Decision Tree (merging equivalent nodes and eliminating the redundant ones) we will arrive at the BDD given in figure 1a. Evaluating all three representations (boolean function given above, Binary Decision Tree and the BDD given in figure 1a) of this function will show that all these representations are identical. The number of the nodes in BDD for this variable ordering is 4 while the number of nodes in Binary Decision Tree is 16.

Boolean binary operations on two functions represented by BDDs can be performed in polynomial time in the size of BDDs [2].

It has been noted [2] that the size of BDD depends heavily on the order of the variables in the function. For example using another order ($c < b < a < d$) for the same function will give a BDD shown in figure 1b. In general, the size of BDD can be exponential in the number of variables, however, in practical examples the BDD has usually a smaller size when the appropriate ordering of its variables is used.

The use of BDDs for analysis of PNs has been explained in [14]. A marking of a PN N can be represented by means of a Boolean vector $V \in 2^{|P|}$. Then the fact that a place p_i is marked is denoted by the value of corresponding element $V[i]$ is asserted to TRUE. A reachable marking m_n corresponds to a vector V_n and a Boolean function $R_n(V_i)$ which evaluates TRUE for m_n. Hence the reachability set of a given PN can be represented symbolically as Boolean function $R = \bigcup_{j=1}^{n} R_j$.

From this representation, using structural information about a PN and standard Boolean functions such as quantification and substitution, we can obtain all markings reachable from the initial one via firing the transitions enabled at m. A detailed algorithm was developed [14] which uses the BDD representation of the reached markings and iteratively constructs the symbolic representation in the form of Boolean function. This method is called the *symbolic traversal of PN*. For clarity, we will reproduce this algorithm here in slightly different form. We denote by $\mathcal{T}(f)$ a function which returns a BDD representing the set of markings reachable from the markings represented by f. Then the algorithm will look as follows:

```
NEW = BDD(m_0)
REACHED = BDD(FALSE)
while NEW ≠ BDD(FALSE) do
    REACHED = REACHED + NEW
    NEW = T(NEW)
end do
```

The time of traversing a PN and the size of the BDD constructed during traversal strongly depends on the order of variables which are defined on the places of the PN. If this ordering is unsatisfactory, the use of symbolic traversal may simply be useless. Thus, finding an optimal ordering is a crucial task for the symbolic approach. Avoiding a completely greedy enumeration of all possible orderings, it is impossible to find a good ordering without referring to extra information available from the net. In the next section, we will introduce an algorithm which uses the unfolding segment built from the original PN and helps to obtain an ordering yielding a BDD whose size is close to the optimum.

4 Variable Ordering by Means of Unfolding

Using the BDDs for PN symbolic traversal analysis provides a powerful tool for analysis of the PN specified behaviour. As indicated earlier, obtaining the ordering of variables such that the size of BDD will be close to optimal is very important for efficient PN symbolic traversal.

Consider the BDD built for formula:

$$f = a \cdot \bar{b} \cdot \bar{c} \cdot \bar{d} + \bar{a} \cdot b \cdot \bar{c} \cdot \bar{d} + \bar{a} \cdot \bar{b} \cdot c \cdot \bar{d} + \bar{a} \cdot \bar{b} \cdot \bar{c} \cdot d$$

which is given in figure 2. Each term of this has one variable in normal form and all the others in complementary form. In general, these type of functions can be written in the following form:

$$f = \sum_{j=1}^{n} a_j \cdot \prod_{i=1}^{n} (a_i \ : \ i \neq j)$$

If each of the variables is associated with only one place in a PN, then such formula will evaluate to TRUE for the markings which have *only* one place marked. This type of formulas is also used in *one-hot encoding* technique [7]. Note, that the size of the BDD does not depend on the order of the variables used in it. The size and structure of the BDD remain the same for any possible orderings and furthermore it is essentially the same BDD which has some nodes renamed.

According to the algorithm of PN symbolic traversal, each of the places of the original PN corresponds to a variable in the traverse function. Hence, any subset of places which can never be marked at the same time of that PN will have its traverse function in the form given above. We define an *ME-cluster* (or simply cluster) as a set of places of the original PN that cannot be marked simultaneously. The problem of dividing the places into clusters is NP-hard. Here we suggest heuristics which allows a more efficient calculation of clusters of places to use for variable ordering.

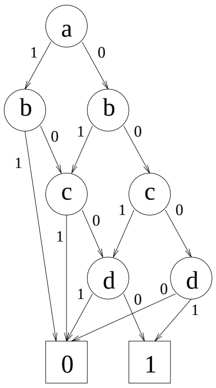

Figure 2: An example of BDD.

The PN unfolding represents the concurrency relation between the transitions of a PN. Similarly, we can obtain the concurrency relation between the places from the unfolding segment. This is stated in the following proposition.

Proposition 1 Two places of a PN can be marked simultaneously iff in the unfolding segment obtained from the PN there exist two instances of these places that are concurrent.

Proof: [*if*] Obvious.

[*only if*] Since the concurrency relation between any pair of transitions can be determined from the unfolding segment and for any occurrence all its output places are included in to the unfolding segment, then any pair of places that can be marked simultaneously in the PN will have corresponding concurrent occurrences.

Two concurrent places (transitions) are denoted as $p_i \| p_j$ ($t_i \| t_j$). We can define two places to be in *orthogonality relation* iff they are not concurrent. According to the above proposition we can easily built a table, Δ, of orthogonality relations between any two places which is calculated as follows:

$$\Delta[i, j] = \begin{cases} 0 & \text{if } p_i \| p_j \\ 1 & \text{otherwise} \end{cases}$$

From this table we can obtain clusters of mutually exclusive places.

For our heuristics we use an algorithm similar to the graph colouring algorithm given in [4] as follows:

```
for each place p in P do
    find a cluster C in CLUSTERS such that p is orthogonal to every p_i in C
        if such cluster C found then do
            add p to C
        else
            add new cluster C_n = {p} to CLUSTERS
    end do
end do
```

This is a greedy algorithm which does not check all the possible clustering of the PN. As it can be easily seen, not every clustering of places will yield the smaller size of BDD. We can observe that if the clusters are *balanced* in number of places contained in them, then such clustering produces better results.

One of the possible ways to reach balanced clustering within our approach is to order the places of the PN in the ascending order of their number of outgoing arcs. Then the places which can be included into the largest number clusters will be considered last [15].

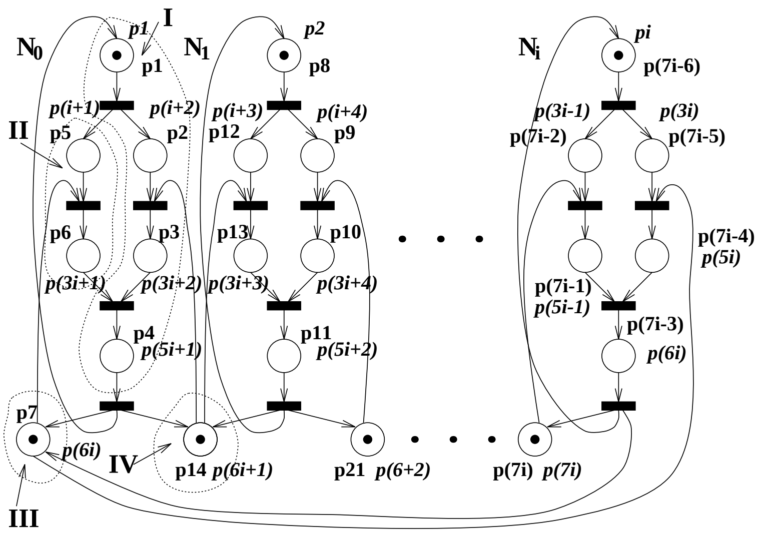

Figure 3: Dining philosophers benchmark.

Obtaining clusters themselves is obviously not enough. Clusters should be also ordered with the same goal — to minimise the size of BDDs. In order to achieve this we will use the notion of the *degree of orthogonality* between two clusters P_i and P_j which is defined from the matrix of orthogonality relations Δ as:

$$\delta_{ij} = |\{\{p_{i_k}, p_{j_n}\} \ : \ \Delta[p_{i_k}, p_{j_n}] = 1\}|$$

In other words, the degree of orthogonality is calculated as a number of mutually exclusive pairs of places between two clusters. We will demonstrate this on the following example. Consider a subnet, N_0, taken alone from the PN shown in figure 3. After clustering we will obtain clusters of places which are encircled by doted lines. Now we can calculate the degree of orthogonality between every pair of clusters. We represent the resulting as a graph in figure 4a, where each node corresponds to a cluster and the arcs are inscribed by the value of degree of orthogonality. Note that we left only those arcs which represent that two cluster are connected. That is, there exists at least one pair of places (p_1, p_2) belonging to two different clusters that exists a transition t such that $p_1 \in \bullet t$ and $p_2 \in t\bullet$.

At this point we can apply a simple greedy algorithm which will order the clusters according to their degree of orthogonality. The algorithm itself is given below:

choose cluster with highest degree of orthogonality
add chosen cluster to the LIST
while not all clusters are in the LIST **do**
    choose cluster which has highest degree of orthogonality with already
        chosen clusters and is connected to larger number of chosen
        clusters
    add the chosen cluster at the end the LIST
end do

The illustration of our algorithm for our example is given in figure 4. Shaded nodes represent the clusters added to the list. After the algorithm terminates we will obtain the following ordering for places: $p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_{14}$.

There should not be any illusions about the complexity of the suggested method. In the worst case the complexity of building the unfolding for an arbitrary PN is exponential. However, it should be taken into consideration that for most of the practical examples the unfolding construction can

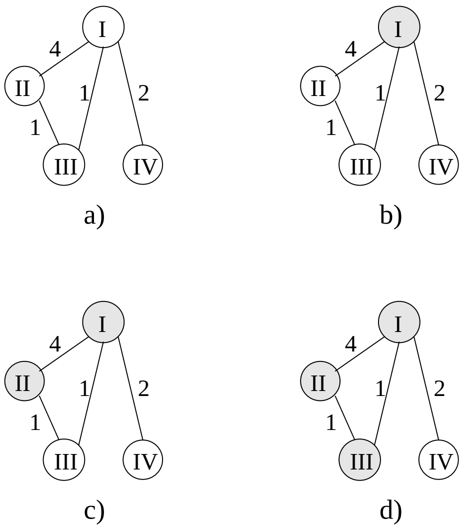

Figure 4: Steps of cluster ordering algorithm.

Benchmark	Arbitrary order		Clustering order		Dynamic reorder	
	Time	BDD size	Time	BDD size	Time	BDD size
10 phil	3.99	554	2.39	357	4.45	357
20 phil	18.33	1174	10.03	737	19.39	737
30 phil	42.93	1794	23.30	1117	44.48	1117
40 phil	76.93	2414	41.13	1497	79.18	1497
50 phil	122.05	3034	64.11	1877	126.37	1877
15 pipe	22.49	1639	12.17	715	22.87	1153
30 pipe	754.81	6694	352.13	2635	786.38	4518
45 pipe	6827.03	15149	2753.51	5755	6988.88	10665

Table 1: Experimental results

be done in $O(n^3)$. Also, the suggested colouring algorithm has $O(n^2)$ complexity. Both finding the connectivity matrix and the matrix containing the degree of orthogonality will require $O(n^2)$. Finally, the ordering of clusters has complexity O(n). Thus, the complexity of our ordering algorithm using clustering will be $O(n^2)$ in the number of places of PN. Thus, in practical examples, the complexity of the obtaining the variable ordering for BDD symbolic traversal of PNs is $O(n^3)$. This situation is similar to the complexity of the reduced state space construction using the persistent set [5] (or stubborn set [20]) methods.

5 Experimental results

We have implemented the unfolding algorithm in C++ running under Solaris2.3 on a Sun SPARC5. The PN symbolic traversal software was developed in UPC[1] using the CMU[2] BDD package.

In order to show the practicality of the developed algorithm we applied it to a set of benchmarks [15] which included such examples as dining philosophers (figure 3) and Muller pipelines. Both types of models are scalable and can be easily grown by simply instantiating an additional number of generic fragments. The results of our experiments on PN symbolic traversal and ordering algorithm using PN unfolding can be observed from tables 1 and 2 respectively.

[1] Universitat Politècnica de Catalunya, Spain

[2] Carnegie-Mellon University, USA

Benchmark	Unfolding clustering			
	Time	No. trans.	Final states	Total States
10 phil	0.60	50	41	4.86×10^6
20 phil	1.73	100	81	2.19×10^{13}
30 phil	3.66	150	121	1.03×10^{20}
40 phil	6.68	200	161	$\approx 10^{27}$
50 phil	10.74	250	201	$\approx 10^{34}$
15 pipe	1.11	70	66	6006
30 pipe	11.61	240	231	6.01×10^7
45 pipe	76.91	510	496	6.90×10^{11}

Table 2: Experimental results

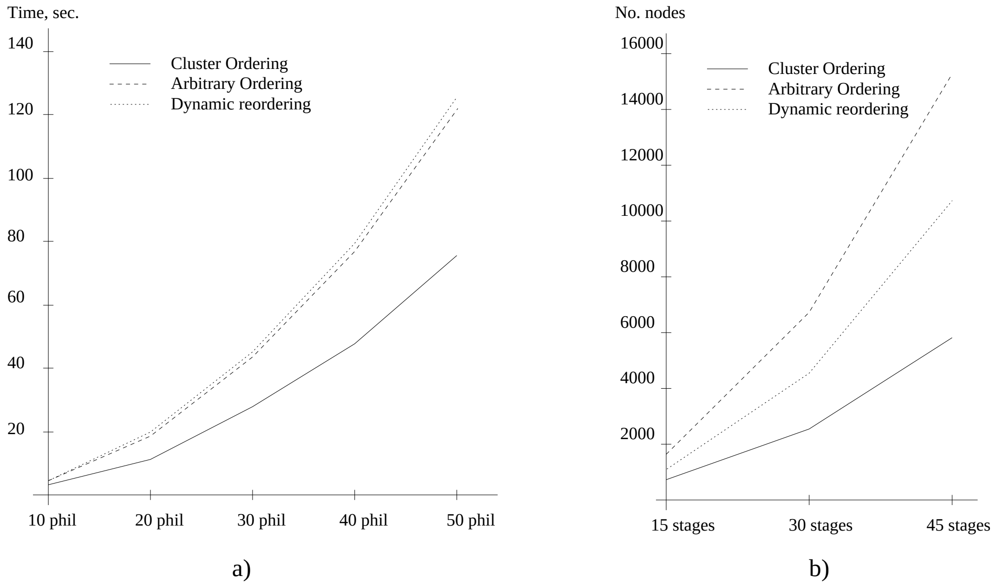

Figure 5: Results of the PN symbolic traversal of dining philosophers benchmark (a) and Muller pipeline (b).

We have compared the results of the PN traversing using the ordering obtained by our algorithm (the times include time for obtaining the new ordering and PN symbolic traversal) and some other arbitrary, although not the worst, ordering of places. The third set of experiments includes the results of variable reordering using the *sift* dynamic reordering procedure supplied with CMU BDD package. Note that there exists an ordering of places in 10 dining philosophers benchmark for which the traversal algorithm fails. This is the ordering which is given in italics in figure 3.

The comparison of the total times of traversing a PN with arbitrary ordering and traversing on PN together with obtaining an unfolding segment and ordering on the places can be seen from the graph in figure 5a. These are given for the dining philosophers benchmark. The results of dynamic ordering are also given. In figure 5b we can observe the growth of BDD sizes for all three methods for Muller pipeline benchmarks. Note that dynamic reordering yields bigger sizes of the BDDs in this case.

6 Discussion and Overall Verification Framework

The results given in the previous section show that the ordering obtained using clustering with the unfolding segment yields reduction in time and space while traversing BDD.

Benchmark	PN unfolding (time sec.)	PN traversal (time sec.)
10 phil	0.18	0.17
20 phil	1.32	0.85
30 phil	4.34	2.01
40 phil	10.40	3.66
50 phil	20.58	6.04

Table 3: Experimental results (deadlock detection)

A reasonable question arises: When is the application of unfolding pre-processing justified for a better obtaining ordering?

Obviously, if the PN is an state-machine PN, then there is no need to run the pre-processing. In this case there will be no concurrent places and the BDD will not depend on ordering.

On the other hand, if a PN has concurrent places and no specific "good" pre-ordering has been made, which is often the case when the net description is generated as output data from an automatic tool, then the unfolding pre-processing for obtaining the ordering according to the clustering approach may reduce significantly the time spent on the PN analysis.

However, it is known that the PN unfolding segment is not well-suited for the verification of purely state-based properties such as deadlock freedom or unique state encoding in circuit synthesis [7]. An algorithm for deadlock detection on truncated unfolding has been given in [10]. This algorithm can be applied to the PN unfolding segment without any changes. The deadlock can be identified as a maximal configuration of the PN unfolding segment which is in conflict with all cutoff points. It has been shown that the problem of deadlock detection on unfolding can, in general case, be exponential to the number of cutoff points, although in practice it gives good results.

The comparison of times needed for deadlock-freedom verification, for both approaches, are shown in table 3. We observe that once the BDD in PN symbolic traversal has been built, we can check for deadlock freedom faster than using the PN unfolding segment. Note however, that the PN symbolic traversal method is able to determine the existence of the deadlock in the PN and show the deadlock marking but is not able to produce the firing sequence (commonly called *trace*) leading into this deadlock. In the PN unfolding segment, due to the fact that the representation keeps the relations between transitions, we can not only identify the deadlock but produce at least one trace leading to this deadlock.

Another problem which can be efficiently solved using the PN symbolic traversal is *Complete State Coding* (CSC) problem. This problem is more known to asynchronous circuits designers working with the PN-based formalism — *Signal Transition Graph* (STG) ([16, 3]). Informally, an STG is a PN whose transitions are labelled with signal value transitions. Many useful results of PN theory have been successfully used in STG verification. The designed STG is usually verified for its implementability — if it is possible to implement a given STG as an asynchronous circuit. This is checked on so a called State Graph (or Full State Graph [17]) which is essentially a reachability graph with consistently assigned binary state vectors. Since the PN symbolic traversal represents symbolically the whole reachability graph, this property can be checked more efficiently using this method [7].

The above discussion shows that both methods considered in this paper should be used in conjunction, complementing each other (see figure 6). For relatively small examples, the PN unfolding segment can be used both for obtaining the variable ordering for future PN symbolic traversal and deadlock detection. When the examples grow larger, and there is more confidence that deadlock detection is rather a formality, it is more efficient to obtain the variable ordering and then proceed to PN symbolic traversal. At the same time, the PN unfolding segment can be kept for later "back-tracking" in the case the deadlock has been found during symbolic traversal, for the identification

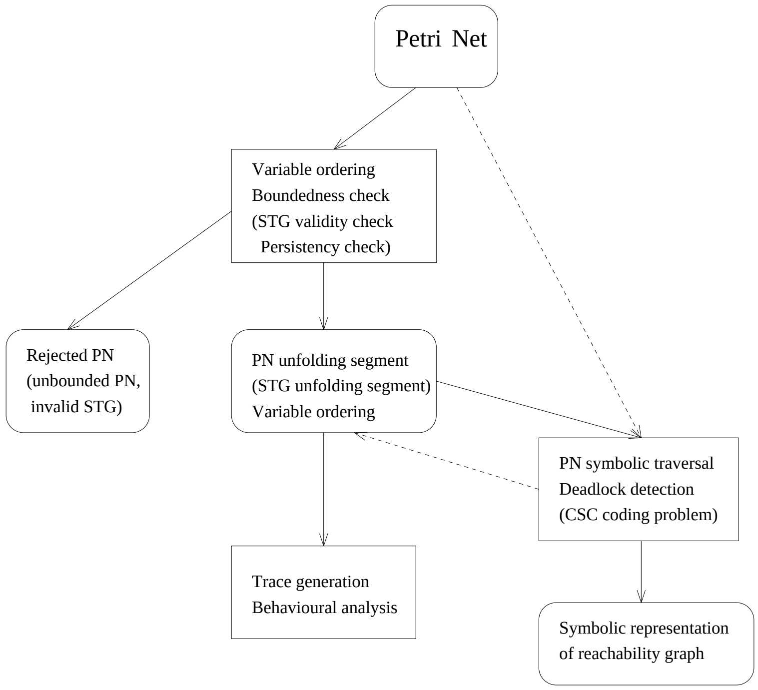

Figure 6: Analysis of PN using both methods.

of the offending trace.

While building the PN unfolding segment we also check for boundedness of the given PN. In the case of unbounded PN the trace leading to unboundedness will be reported at the pre-processing period without wasting time on more expensive traversal. Note that using the similar segment (called *STG unfolding segment* [18]) in STG analysis also allows us to check the validity of STG [17] "on-the-fly", i.e. while building the segment itself. Thus, all the traces invalidating the behaviour described by the STG will be reported at the earlier stage of the analysis.

7 Application to Asynchronous Circuit Verification

7.1 Modeling approach

The most advocated approach for the design of asynchronous circuits is a top-down design. This refers to two major styles currently pursued by researchers. One is based on logic synthesis of circuits from behavioural models, such as STGs. The other is based on decomposition of behavioural specifications until such specifications are sufficiently simple that they can be converted into some pre-designed circuit components. In the first case, the Petri net model underlying the STG is verified for its logical circuit implementability [7]. In the second case, designers often skip formal top-down refinements and instead resort to a bottom-up techniques. The final implementation is thus obtained in an "ad hoc" manner, by associating the behavioural paradigms directly with the structural components of the circuit.

In our application we have been looking at exactly these types of designs. The existing circuits need to be verified for their correct functioning. As a result of going "bottom-up", the issue of correctness as a conformance between the specification and the implementation is not obvious. Rather, the correctness criteria are more generic, such as ensuring that the circuit does not halt at some state or does not produce hazardous spikes.

The use of a currently popular micropipeline approach [19] allows the designer to compose the circuit from a set of primitive components, whose behaviour is purely event-triggered. This means that all that matters in such circuits is the act of switching of a circuit signal from one level (say,

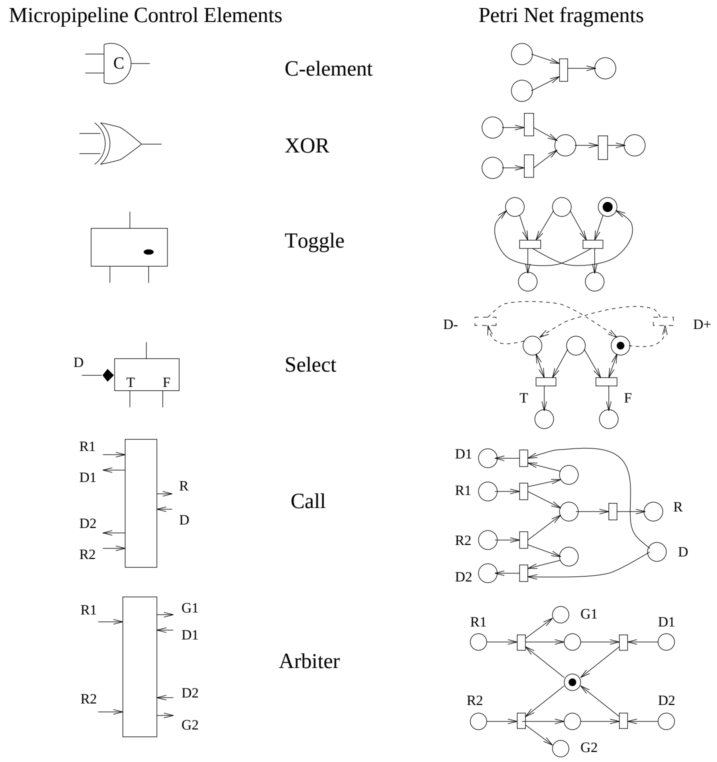

Figure 7: Translation of the circuit components into PNs

logical 0) to another level (logical 1). The actual level is not important as long as the circuit is kept in a proper operation cycle. Due to this nature of signal interpretation, we can associate most of the elements of the asynchronous micropipeline circuits with corresponding PN fragments. Some of the examples of the asynchronous circuit element representation are given in figure 7.

The main idea of this type of modelling is that the places are associated with wires and the transitions with events on these wires. Since we do not distinguish between rising and falling signal events in the event-triggered discipline, it is possible to associate one net transition with both. It is also possible to optimise the model by deleting redundant transitions and places when the fragments are connected together. For example, the XOR module can often be represented by only two places (one for *merging* and the other to model the output wire), and one transition in between. The effect of two or more inputs is made possible by connecting the places which correspond to the sources of the inputs for the XOR directly to its merging place.

The model of the Select element has a special feature. The complete model shows the effect of the environment which changes the state of input D (meaning “data”). Since D is a *level-based* signal, its edges, denoted as $D+$ and $D-$, are not “symmetric” as for event-based signals, and must be modelled by separate net transitions. The figure does not show the origins of the logic that switches D.

It is often the case that a good design is in fact a combination of the event-based and level-based signalling styles therefore two types of components must be provided. In addition to the above micropipeline elements, the designer may need to use logic gates. Figure 8 shows two simple examples of such models, for an invertor and an OR-gate. This modelling is in fact a specific type of STG, in which each signal y is associated with two places, representing its two logical states. the groups of transitions labelled with $y+$ and $y-$ are connected to these places in such a way that the enabling/firing AND semantics of Petri net transitions, “corrected” through the appropriate labelling mechanism, adequately represents either AND or OR conditions in the circuit logic. The actual input “guards” for these transitions are formed using *self-loop* Petri net arcs from the places associated with the state of the input signals to the gate.

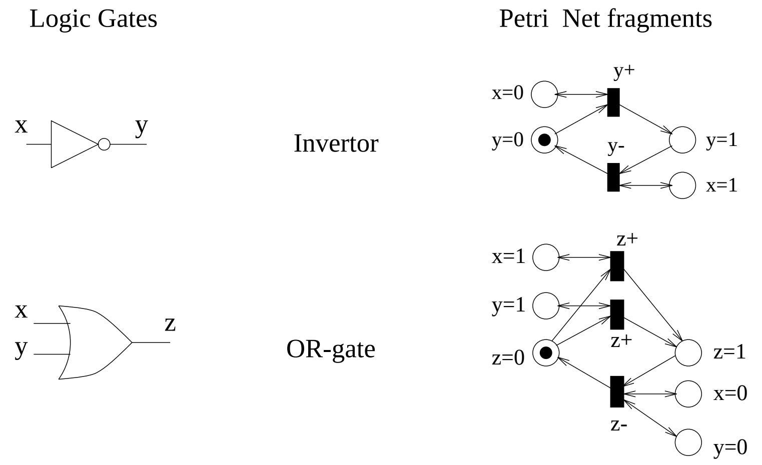

Figure 8: Modelling level-based elements with Petri nets

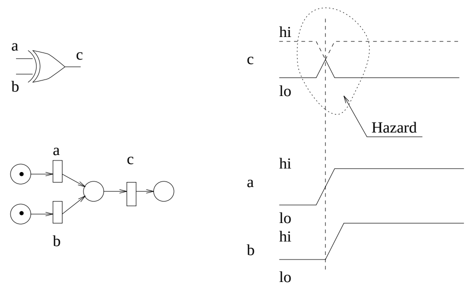

Figure 9: Hazardous behaviour of XOR element described in PN

The use of self-loops, rather than "normal" input arcs is essential to this modelling method. It only allows tokens to be moved from the state-holding places associated with signals by firing transitions of the elements whose outputs are modelled by these inputs. Therefore, if one models a circuit with inputs and outputs, the Petri net model of the circuit can only change the state of the places associated with its outputs. The marking of the places for the input signals can only be changed by the part of the net representing the circuit's environment.

7.2 Analysis of circuit behaviour

After the circuit elements have been translated into the PN net fragments, we can apply simple composition (through places) to obtain the PN corresponding to the whole circuit. Then the PN model of the whole circuit is verified along with the model of the environment.

What are the properties that need to be verified on the PN built from the asynchronous circuit? If the circuit was built following the event-based signalling protocol, then we need to verify *safeness* of the obtained PN. The safeness implies that the circuit does not have any hazards in it. This can be easily seen from the following example.

The XOR element is supposed to have only one input changed at a time. If two inputs are enabled concurrently, then this situation will lead into a hazardous state. Indeed, the first change of the signal a will cause the output signal to change its value whereas the next one will force it to go back to the original one. Since none of the signals is restricted in time, they can happen close enough to cause the spike at the output of the XOR gate. It can be shown in a similar way, by considering an example of a Petri net model of a level-based circuit, that hazards on level based signals are interpreted as *non-persistency* of Petri net transitions labelled with the corresponding signal level changes. This also applies to some event-based signals. For example, we can check if the inputs of the Select element (see Figure 7) are mutually exclusive by checking persistency of

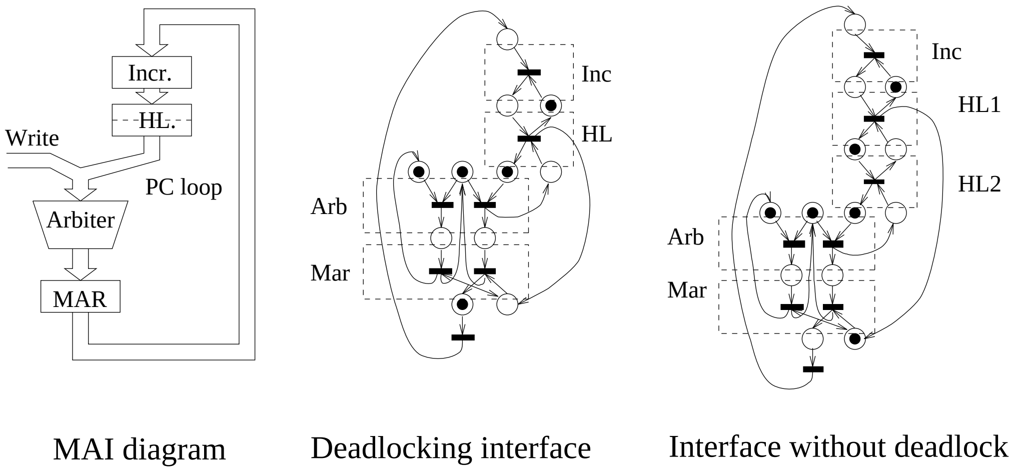

Figure 10: Memory Address Interface

its transitions corresponding to outputs T and F.

We should however bear in mind that some signal transitions must be allowed to be non-persistent. These are associated with the model of the Arbiter, where the effect of transition disabling does not lead to hazards due to special analogue circuitry inside the Arbiter to resolve *metastability* situation in a safe way.

As a real example, we have analysed the Memory Address Interface (MAI) of the AMULET [1] asynchronous microprocessor (one of the most significant design examples of the recent time). The simplified diagram of the MAI is given in figure 10. It shows one of the main functionalities of this circuit. The Memory Address Register (MAR) is represented by two transitions due to existence of the select block in front of the MAR [12]. A simple analysis of this PN yields that the MAI containing only one holding latch (HL) has a potential deadlock in it. This is the situation the arbiter is won by the side write request. It can be observed that the MAI which has two holding latches does not have such a problem.

By using the modelling approach described in the previous section, we have converted the schematics of the MAI into a Petri net. This net was run through the verification framework (combination of the unfolding and symbolic traversal techniques) outlined in Section 6. The results of the verification showed that the circuit was correct both in terms of freedom from hazards and deadlocks. Note that in the course of anlaysis for hazards (through checking both safety and persistency with respect to non-arbitrating signals) we had to apply some special timing assumptions, realistically placed by the designers, as the actual circuit was not intended to be completely speed-independent [6]. They were added in the form of additional causality constraints (extra places) into the Petri net model.

8 Conclusion

We have developed a new approach to the verification of Petri nets based on the combination of partial order (PN unfolding) and symbolic traversal techniques. This approach uses an algorithm for obtaining the ordering of variables in the BDD employed for symbolic traversal of the PN state space. Experimental results show that our approach is practical for known set of benchmarks.

BDDs can represent big state spaces of PNs at a relatively small cost provided that proper ordering has been found. The application of the PN unfolding segment as a pre-processing stage gives a possible solution to this problem especially when the ordering supplied with a PN is unclear.

Our approach has been applied in the verification of circuits in an asynchronous microprocessor built on the micropipeline principles. Such principles allow a relatively straightforward conversion of circuits into Petri nets and a coherent interpretation of circuit correctness in terms of well-known

Petri net properties, such as safeness, persistency and deadlock-freedom.

Further application should be found in the analysis of Signal Transition Graph specifications. This will also speed up the existing technique of verification of implementability of STG [7]. An attractive issue here is that while building the unfolding we can easily check correctness of an arbitrary STG. Such properties as boundedness/safeness, validity of the STG [17], persistency of transitions and signals can be checked effectively during building the PN (or STG [17]) segment. Thus the BDD will be constructed only for correct STGs and afterwards only implementability of correct STGs will be checked.

9 Acknowledgements

The authors would like to acknowledge help in carrying out this work from Oriol Roig, Jordi Cortadella and Enric Pastor. We are also grateful to the AMULET group for providing us with their schematics and answering our numerous queries. Alex Kondratyev, Alexander Taubin and Mike Kishinevsky are also acknowledged for their useful discussions about the Petri Net analysis by means of unfoldings.

References

[1] Amulet1 group workshop: Presentation materials. Technical report, Manchester University, Department of Computer Science, Amulet Group, Lake District, England, July 18-22 1994.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulations. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] T.A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications.* PhD thesis, MIT, 1987.

[4] Gavril F. Algorithms for minimum colouring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Jornal on Computing*, 1(4):180–187, December 1972.

[5] P. Godefroid and P. Wolper. Using partial orderes for efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[6] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* John Wiley and Sons, London, 1993.

[7] A. Kondratyev, J. Cortadella, M. Kishinevsly, E. Pastor, O. Roig, and A. Yakovlev. Checking Signal Transition Graph implementability by symbolic BDD traversal. In *EDAC-95*, 1995. Accepted for publication.

[8] A. Kondratyev and A. Taubin. On verification of the speed-independent circuits by STG unfoldings. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah, USA*, November 1994. To appear.

[9] A. Kondratyev, A. Taubin, M. Kishinevsky, S. Ten, and A. Yakovlev. Analysis of petri nets by ordering relations. Draft submitted to 16th International Conference on Application and Theory of Petri Nets, November 1994.

[10] K.L. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. Manuscript, February 1993. The earlier version of this paper was presented on the 4th Workshop on Computer Aided Verification, Montreal, 1992.

[11] T. Murata. Petri nets: Properties, analysis and application. *Proceedings of IEEE*, 77(4):541–574, April 1989.

[12] S. Nicklin. Personal communications.

[13] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. part I. *Theoretical Computer Science*, 13:85–108, 1981.

[14] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulations. In *Proceedings of 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain, June 1994*, pages 416–435, 1994.

[15] O. Roig. Personal communications.

[16] L.Ya. Rosenblum and A.V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets, Torino, Italy, July 1985*, pages 199–207. IEEE Computer Society, 1985.

[17] A. Semenov and A. Yakovlev. Event-based framework for verification of high-level models of asynchronous circuits. Technical Report 487, University of Newcastle upon Tyne, 1994.

[18] A. Semenov and A. Yakovlev. Full state graph and its verification for asynchronous circuits. Manuscript, November 1994.

[19] I.E. Sutherland. Micropipelines. *Communications of ACM*, 32(6):720–738, 1989.

[20] A. Valmari. Stubborn Sets for Reduced State Space Generation. *Advances in Petri Nets 1990, ed.G.Rozenberg, LNCS 483*, pages 491–515, 1991.