Modelling, Analysis and Synthesis of Asynchronous Control Circuits Using Petri Nets

A.V. Yakovlev
A.M. Koelmans
A. Semenov
Computing Science Department,

D.J. Kinniment
Department of Electrical
and Electronic Engineering,
University of Newcastle upon Tyne, UK

Abstract

In this tutorial paper we survey some of the existing techniques for modelling, analysis and synthesis of asynchronous control circuits. All these methods are based on the use of Petri nets as a tool for describing the behaviour of such circuits. The descriptive power of Petri nets allows them to model a wide range of asynchronous circuit components, whether they are built in the two-phase (micropipeline) or in the four-phase (logic gate based) design styles. We present three different approaches to verification of net-based models, and show their relative strengths and weaknesses. We advocate their complementary application for different classes of Petri nets and the properties verified. Two major synthesis approaches are demonstrated using the example of a modulo-N Up/Down counter. The first one is a combination of Petri net level decompositions and syntax-directed translation of nets into circuits. The second one is based on logic synthesis from Signal Transition Graph specifications.

1 Introduction

The design of asynchronous circuits of a reasonable size and complexity is still rare. The Amulet1 microprocessor, recently developed by Steve Furber's group at the University of Manchester [11], is one of the few examples. A specific feature of this design is that it has been almost entirely done within the micropipeline approach, originally presented by Ivan Sutherland in his Turing Award Lecture [43]. Micropipelining organises the control of the data path and the interaction between parts in a purely event-based way, using *2-phase* or *transition signalling* (more traditionally called *Non-Return-to-Zero*).

Amulet1 designers admit to the fact that most of the architectural and logic design of the chip has been done using standard design tools, without the use of formal models. No formal synthesis tools were used in the implementation. Formal techniques were applied at a later stage, but did not influence the design process. They were aimed at verifying some of the most sophisticated parts of the chip, in order to confirm that the circuit is free from deadlocks. The work was based on the CCS [24] process algebra and the associated CWB software tool [3]. For example, it was demonstrated that the Memory Address Interface, if built using one parallel data latch less in the Program Counter incrementing loop, would deadlock – a result already appreciated by the designers at an informal level. Similar conclusions were drawn within the team of designers themselves, through the use of Petri net models and the application of reachability analysis [27].

Despite being formally well-founded, the CCS modelling approach has certain shortcomings when compared with the use of models based on Petri nets. The graphical nature of the Petri net notation makes it more attractive to circuit designers than the algebraic notations, which are much less intuitive. Petri nets can also be used to check for potential hazards in circuits by associating them either with Petri net unsafety or non-persistence.

Another potential advantage of Petri nets is that they can be used as a modelling language to perform formal synthesis [49]. Practical asynchronous designs, which may not be entirely speed-independent, are increasingly becoming an issue, and here, again, the use of Petri nets appears to be profitable. An easy annotation of net elements with time attributes, in addition to their normal causality paradigm, makes them an excellent tool for timing analysis and synthesis under specific timing constraints [35, 9, 40, 20, 18].

However, it is important to note that net-based analysis of real designs by means of "brute force" reachability exploration is often prohibitively complex. This is due to combinatorial explosion of the state space for models with a high level of concurrency. It is therefore crucial to develop better verification techniques, such as net unfoldings. This allows concurrency to be captured in its natural *partial order* form, without unnecessary *interleaving* of concurrent events as in the reachability analysis methods.

This paper presents a systematic approach to the use of Petri nets for the modelling, verification and synthesis of asynchronous circuits with a micropipeline-based architecture. We first present a set of Petri net constructs to model the major transition-signalling elements such as the C-element. The use of nets is extended to the modelling of *level-based* circuits, because the ability to model mixed 2-phase and 4-phase circuits is crucial in the design of micropipeline architectures. We also show how Petri nets can be used for the modelling of simple data transfer micropipelines. Secondly, we examine the three major techniques that can be used for the verification of Petri net models. Two of these are based either on explicit and symbolic analysis of Petri net reachability space and one employs the partial order semantics produced by the Petri net unfolding algorithm. Each of these techniques has its own advantages and shortcomings but we attempt to view them as complementary rather than competing approaches.

Thirdly, we look at the two possible synthesis approaches. One is based on syntax-directed translation of Petri net specifications into self-timed circuits, where the circuit can be seen as a hardware implementation of the Petri net control flow structure. The other approach proceeds from a special type of interpretation of a Petri net, called Signal Transition Graph. We finally demonstrate how these techniques can be used in designing a modulo-N Up-Down counter. An important issue is the use of both approaches on the same design examples, where the syntax-directed translation is used as a rapid prototyping tool and the logic synthesis is used to improving the efficiency of the design implementation.

2 Modelling Asynchronous Circuits with Petri Nets

2.1 Petri Nets and Signal Transition Graphs

Petri nets and their properties. We briefly summarize the main features of Petri nets. For a more complete overview, see [26, 34]. An *ordinary Petri net* (often shortened to *net*) is a graph with two types of nodes: circles, called *places*, and bars or boxes, called *transitions*. Directed arcs between places and transitions, and vice versa, denote the *flow relation*. A *marking* of a Petri net is a multiset m of places. Usually, the *initial marking* is represented by tokens in places of the net. A marked net is therefore a net with its initial marking m_0. Any Petri net in this paper is assumed to be marked, unless specified otherwise.

Any Petri net may generate dynamic behaviour by means of the so-called *token game*. A transition is *enabled* in marking m if all its predecessor places are marked with tokens. An enabled transition may *fire*, yielding a new marking m', in which the number of tokens in each predecessor place (usually denoted as $\bullet t$) is decremented by one and in each successor place ($t\bullet$) is incremented by one. The

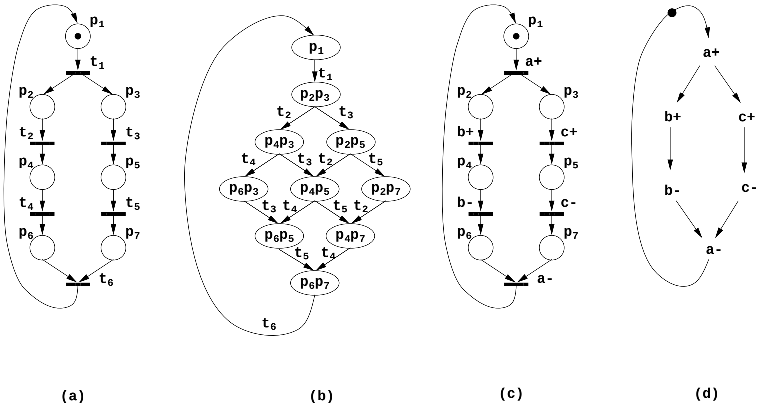

Figure 1: A Petri net, its reachability graph and a Signal Transition Graph built on this net.

number of tokens remains unchanged in places which are either both predecessor and successor to the transition or if they are not connected to it by the flow relation.

A (possibly empty) sequence σ of transitions including all intermediate transitions which have fired between two markings m and m' is called a *firing*, or *feasible* sequence. A marking m' *reachable* through σ from m is denoted as $m[\sigma\rangle m'$. A set of markings reachable from the initial marking m_0 (denoted as $m_0[\rangle$) is called the *reachability set* of the marked Petri net. It can be represented as a graph, called the *reachability graph* of the net, with the nodes labelled with the markings and the arcs labelled with transitions. An example Petri net and its reachability graph are shown in Figure 1.

A Petri net is called *k-bounded* iff in any reachable marking in $m_0[\rangle$, the number of tokens in any place is not greater than k. A net is called *bounded* if there exists a finite k for which it is k-bounded. It is clear that the reachability set of a bounded net is finite. A net which is 1-bounded is called *safe*. Because we will only be modelling hardware of finite size, we consider only bounded nets.

The following two properties, deadlocks and persistency, will be useful in checking correctness of circuits.

A marking m of a Petri net is called a *deadlock* if it does not enable any transition in the net. A marked net is called *deadlock-free* if its reachability set contains no deadlocks. Freedom from deadlocks characterises a system's model which never stops its activity, e.g. a circuit (possibly modelled together with its environment) whose every state has a signal which is ready to change its value. A somewhat related property, called *livelock* is defined as a strongly connected *proper subset* of reachable markings. The presence of a livelock usually implies that the system exhibits only partial behaviours, e.g. a circuit which operates in a cyclic manner but activates only a subset of its signals.

A marked net is called *persistent* with respect to some transition t if for any reachable marking enabling t we cannot fire another transition t' and reach another marking m' in which t is not enabled. If the net is non-persistent in transition t due to the firing of t', we say that t and t' are in *dynamic conflict*. It is clear that in order to be in dynamic conflict two transitions must share at least one predecessor place. This sharing is called a *structural conflict* between t and t'. Obviously, a structural conflict may not necessarily cause a dynamic conflict. A net is called persistent if it is persistent with respect to all its transitions. As will be shown later, persistency with respect to some transitions (associated with signal events on logic gates) often characterises absence of hazards in a circuit.

Signal Transition Graphs. A special type of *interpreted* or *labelled* Petri net is called a *Signal Transition Graph (STG)*, introduced independently in [36] and [4]. The transitions of an STG are labelled with the names of signal edges for a given set of signals A. Thus, for a signal a in set A, the allowed labelling of transitions is $a+$, meaning the rising edge, and $a-$, the falling edge, of a. An STG has an initial state, associated with the initial marking m_0, which is a binary vector of dimension

$n = |A|$. An example of an STG interpretation of the net from Figure 1(a) is shown in Figure 1(c). A "shorthand" notation is often used for depicting STGs whose underlying Petri nets have places with exactly one incoming arc and one outgoing arc. Every such place, together with the arcs incident on it, is replaced by a single arc. A token, whenever it should be associated with such a place, is put directly on its arc replacement. The STG of Figure 1(c) is shown in its shorthand form in Figure 1(d).

A feasible sequence σ of an STG is called *valid* iff for every signal a: (i) the next possible edge of signal a after $a+$ $(a-)$ can only be $a-$ $(a+)$, and (ii) the first change of signal a is consistent with the initial state of the STG, i.e.: if the value of a is 0(1) in m_0, then only $a+$ $(a-)$ can first appear in σ. An STG is *valid* iff every firing sequence generated from m_0 is valid.

It has been shown [36, 4] that any valid STG has a consistent binary state encoding for all of its reachable markings. In fact, the actual validity check can be performed through a consistency check of the state encoding $v : m_0[\rangle \to \{0,1\}^n$ of the reachability graph. Such an encoding is *consistent* iff for each edge $m' \to m''$ in the reachability graph, labelled with signal a:

- if the edge is labelled $a+$, then signal a is at logical 0 in $v(m')$ and 1 in $v(m'')$,
- if the edge is labelled $a-$, then signal a is at 1 in $v(m')$ and 0 in $v(m'')$,
- otherwise signal a has the same value in both $v(m')$ and $v(m'')$.

With the aid of a consistent encoding v, we can use the reachability graph in its binary encoded form, which is called the *state graph* of the STG.

2.2 Models of Basic Components

Modelling event-based (two-phase) elements. The original set of components used for building two-phase control circuits was proposed by Sutherland [43]. They include the following:

- **C-element**, which implements AND-causality or synchronisation between different processes. The admissible behaviour of this element is such that both inputs are allowed to change their values, say from logical 1 to logical 0, if the output change, from logical 0 to logical 1, has been produced for the previous input change. This component is described by the boolean equation: $Y = x_1x_2 + y(x_1 + x_2)$, where Y is the output and x_1 and x_2 are the inputs of the C-element. The variable y is used to distinguish the feedback wire from the main output.

- **Merge**, or **eXclusive-OR (XOR)**, which realises OR-causality between input changes, but requires that **only** one input can change at a time. Input changes must therefore arrive on a mutually exclusive basis.

- **Toggle**, which switches between two outputs for every input change, as a complementary flip-flop. It is used to *unconditionally* alternate between two possible directions of control flow.

- **Select**, which changes one of the two output signals but, unlike Toggle, does it *conditionally*. The output is selected depending on the state of another input, the *level-based* one. This component allows the construction of branches in control flow depending upon the state of the data path.

- **Call**, which operates like a control flow multiplexor. It transmits any of its alternative input requests to the single output request, and upon receiving the acknowledgement from the single acknowledgement input, transmits it to the acknowledgement output corresponding to the original request. This module therefore operates as an interface between different parts of the control flow and the single operational unit. It is crucial that input requests change in a mutually exclusive manner.

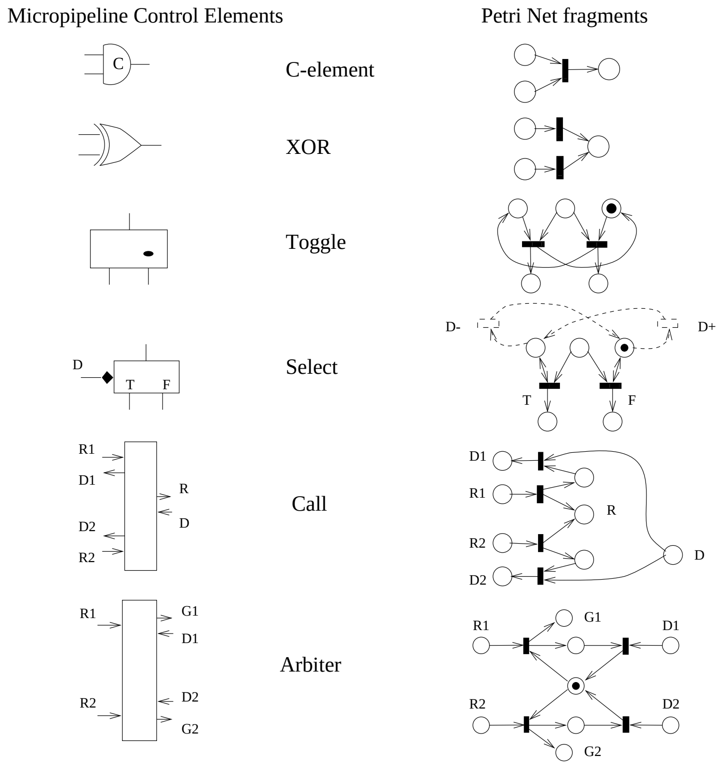

Figure 2: Basic two-phase control elements and their Petri net models.

- **Request-Grant-Done (RGD) Arbiter**, which arbitrates between two possibly concurrent input requests (R) and generates only one grant (G) at a time, using a built-in metastability resolution circuit. To indicate that one of the requestors has finished a *critical section* of the computation, it uses another input, called "Done" (D).

These components are modelled by the Petri net fragments shown in Figure 2. The main idea behind this type of modelling is that the places are associated with input/output wires and the transitions with signal events. Since we do not distinguish between rising and falling edges of transitions in the two-phase discipline, it is possible to associate one net transition with both.

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, also called *four-phase* signal, its edges, denoted as $D+$ and $D-$, are not "symmetric", and must be modelled by separate net transitions. The figure does not show the origins of the logic which switches D.

Modelling level-based (four-phase) logic. Any efficient circuit design requires a flexible combination of two signalling styles, two-phase and four-phase. We therefore present a general technique for modelling level-based circuit elements. Figure 3 shows two simple examples of such models, for an inverter and an OR-gate.

This type of modelling was called *Circuit Petri Nets* in [45]. It 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 logic. The actual input "guards" for these transitions are formed by using *self-loop* Petri net arcs from the places associated with the state of the input signals to the gate. 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

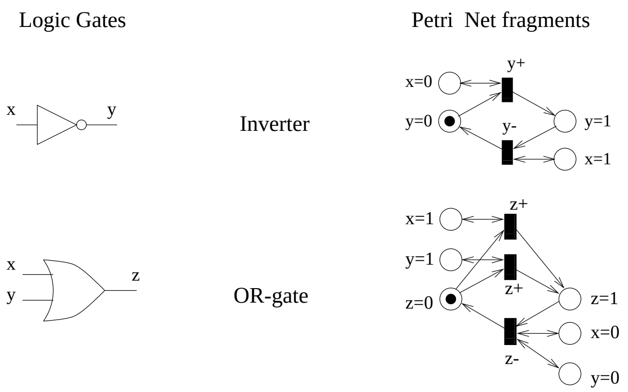

Figure 3: Modelling ordinary logic gates (level-based elements) with Petri nets.

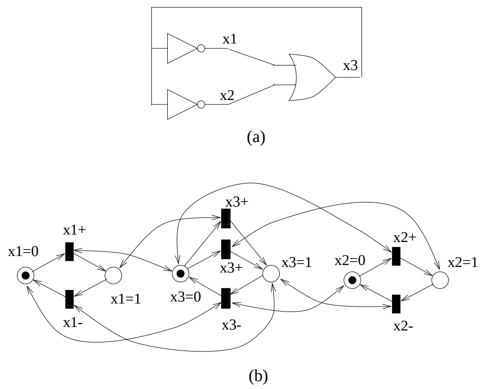

Figure 4: Example of a level-based circuit model.

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 by changed by the part of the net representing the circuit's environment.

As an example of the model of a level-based circuit, consider the one shown in Figure 4(a). This circuit is closed, i.e. it is autonomous and has no interconnections with its environment. Its behaviour can be analysed using the reachability graph of its Petri net, which is depicted in terms of the binary states corresponding to the markings of the Petri net as shown in Figure 4(b). It is easy to see that this circuit has hazards if the delay of one of the inverters (say, x_1) is greater or equal to the sum of the delays of the other inverter and the OR gate. In this case, the Petri net may start in state 000, in which both transitions x_1+ and x_2+ are enabled, then fire x_2+ and x_3+ in sequence, and finally enter state 011, with x_1+ disabled without firing. In the physical circuit, this corresponds to a potential hazard on signal x_1, while in Petri net terms, this is called *non-persistency* of the transition.

This example suggests a canonical way in which level-based circuits can be formally checked for hazard-freedom. Such a circuit has a hazard in signal x if the Petri net model is non-persistent with respect to a transition labelled with x. A similar interpretation of a Petri net property can be developed for circuits involving two-phase components. For example, we can check if the inputs of the Select element are mutually exclusive by checking persistency of its transitions corresponding to outputs for T and F. We should of course 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 in a safe manner.

In addition to (non-)persistency, another Petri net property, safeness, can be used to detect er-

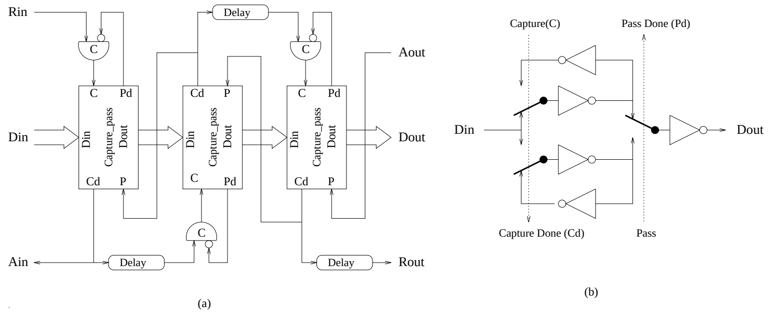

Figure 5: Sutherland's micropipeline FIFO (a) and Capture Pass storage element (b).

roneous behaviour of the modelled circuit. For example, consider a Merge (XOR) element with two inputs. If one of its inputs changes before the output has been able to respond to the change of its other input, then this manifests potentially hazardous behaviour of the XOR. In the corresponding Petri net, this would correspond to the arrival of two tokens – thus causing the net to be unsafe.

2.3 Modelling of a micropipeline FIFO

In this section we use the above techniques for modelling micropipeline FIFO structures. The basic two-phase FIFO structure was proposed by Sutherland [43]. It is based on a data path storage element called *Capture Pass Element*. Both are shown in Figure 5.

The synchronisation between data path and control in this FIFO is done using the *bundled data* technique, which requires using explicit *delay* elements between the stages. These compensate for delays and signal skewing in the bundles of data wires. Alternatively, one could use dual-rail signalling [38], but this option is often discarded in practice as overly expensive in terms of area and power consumption.

We only consider modelling of control flow in such pipelines here. The extracted control circuit for two adjacent FIFO stages is shown in Figure 6(a). The data path is abstracted away by means of delays inserted into the appropriate wires.

The operation of the FIFO stages is represented by the Petri net also shown in Figure 6(a). This net has a certain degree of redundancy that is caused by explicit modelling of the two delays between events on C and Cd and P and Pd, respectively. It is easy to reduce this net to a simple loop with two transitions, each modelling the C-element that synchronizes the request signal from the previous stage with an acknowledgement signal from the next stage. The reduced model, together with the models of the left hand side Sender and the right hand side Receiver, is shown Figure 6(b).

The Petri net model shown in Figure 6 is persistent and safe, hence the circuit modelled by it has no hazards.

The performance of the pipeline is determined by the following two parameters. The first is the *latency*, which is the time it takes to propagate a datum through the stages. In Figure 5(a) this would be the time from Rin to Rout. This can be evaluated by unfolding the Petri net model (Section 3 defines the Petri net unfolding), annotated with time, and calculating the delay of the corresponding critical path leading from Rin to Rout. Additionally, one may consider the *reverse latency*, which is applied to the complementary situation, the propagation of the acknowledgement through the stages (e.g., when the pipe is full of data). The second performance factor is the *cycle time*, or the time it takes one stage to process one value and accept the next one. The maximum cycle time of all the stages determines the overall *throughput*. The cycle time can be evaluated from the net using the technique described in [29]. This technique is, however, currently applicable to a restricted class of nets, called *marked graphs* [26], which can only model causality and concurrency but not choice.

Despite obvious performance gains achieved through the combined use of two-phase control with a Capture-Pass storage element, Amulet1 designers considered this design to be too costly in area and transistor count [32, 6]. They preferred to use conventional pass-transistor *Transparent Latch* circuits,

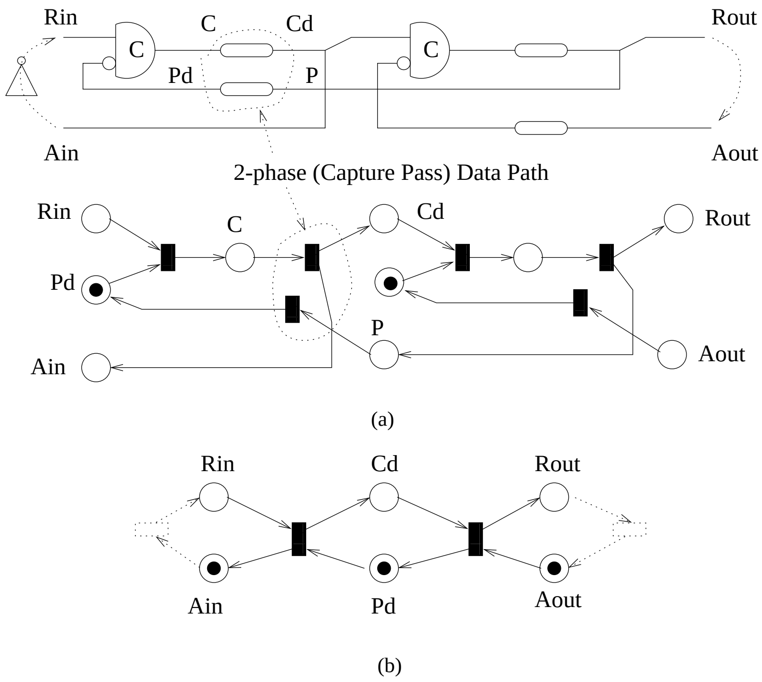

Figure 6: Petri net model of Sutherland's micropipeline FIFO

which are four-phase operated. The latch is controlled by two complementary *enabling* signals, En and nEn. As a result, more complex control circuitry has to be used to convert the two-phase control of the interface between the stages to the four-phase control of the latches inside the stages and back. The conversion is done by means of a combination of a Merge (XOR gate) and a Toggle.

We model here two possible designs from [32, 6], based on the four-phase data path. The first, the *standard* one, is shown in Figure 7 for one stage. It shows that the control circuit is delay-insensitive but has a long latency and cycle time. The second design, called *fast-forward pipeline*, has better performance (smaller forward latency) but at the cost of losing delay-independence. It is shown in Figure 8. This net, if analysed without taking timing parameters into consideration, is unsafe. The place that models the Merge gate can be marked with more than one token. To avoid this, we must guarantee that transitions $t1, t2$ and $t3$ fire, and remove the previous token from the Merge place, before $t4$ fires and adds another token. Such a constraint is satisfied by the design, as can be seen from the Petri net model comprising two adjacent stages and the separation time between the occurrence of $t4$ from that of the $t1$. It can be noticed that the reverse latency in this design still remains the same as in the previous design.

3 Analysis and verification techniques

In this section we briefly survey the various techniques for verification of Petri net models of circuits, consisting of two- and four-phase control elements. They can be divided into the following three major groups:

1. Reachability graph construction and analysis,
2. Symbolic (e.g., using Binary Decision Diagrams) traversal of reachable states,
3. Partial order (e.g., using net unfolding) construction and analysis.

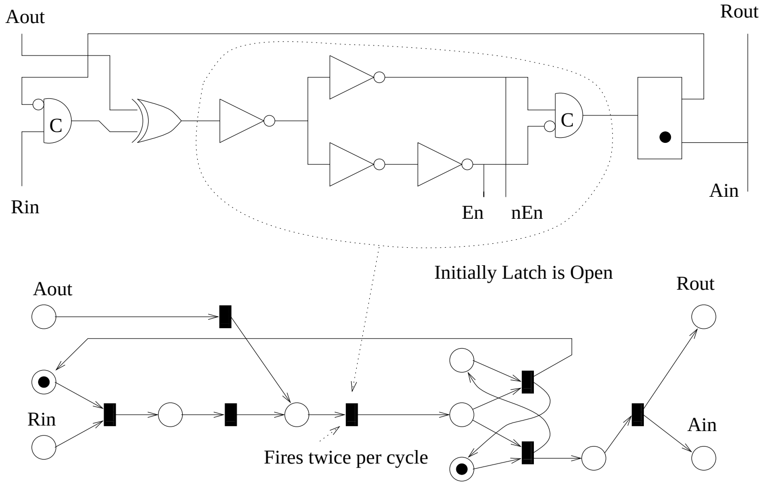

Figure 7: Model of the standard pipeline with four-phase latch control

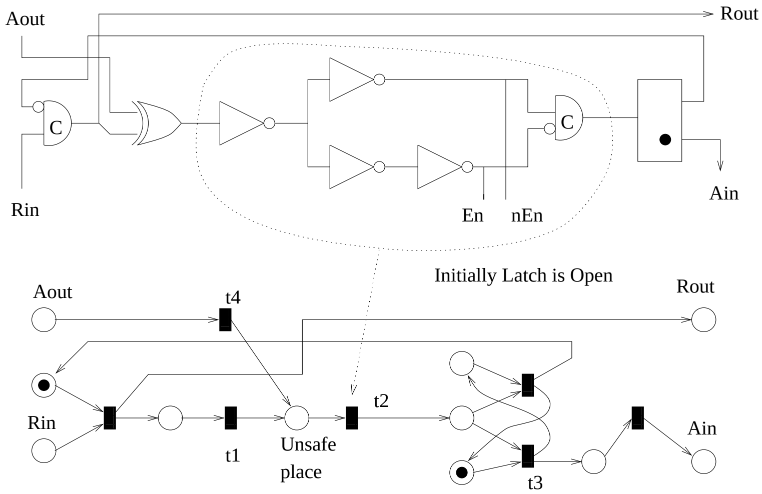

Figure 8: Model of the fast-forward pipeline with four-phase latch control

```
proc GenRRG()
  GRAPH = empty
  STACK = {m_0}
  GenStates(STACK)
end proc


proc GenStates(STACK)
  while STACK is not empty do
    m_i = pop(STACK)
    if m_i ∉ GRAPH do
      GRAPH = GRAPH + {m_i}
      T^st = findStubbornSet(T, m_i)
      for each transition t from T^st enabled at m_i do
        push(STACK, fire(m_i, t))
      end do
      GenStates(STACK)
    end do
  end do
end proc
```

Figure 9: Algorithm for generating a reduced reachability graph.

3.1 Reachability graph analysis

The first group of methods, including those defined by means of trace theory [7, 8], operate with the interleaving semantic model of concurrency. Such methods produce all states and thus capture all firing sequences. As a result, the fully built reachability graph (or state graph for an STG) may suffer from combinatorial explosion if the modelled system is highly parallel.

Reduced reachability analysis. For some properties, such as deadlock, one may only need a partial reachability set, containing only critical states. The technique, based on so-called *stubborn sets* of transitions, makes use of structural information about transitions and markings [44]. For every marking explicitly built, the set of transitions that are enabled is partitioned into subsets of transitions in which transitions from different subsets are mutually independent. Mutual independence means that the transitions do not share any input places and thus can fire concurrently. Thus picking up and firing only one such group of transitions is sufficient, since the enabled transitions from the remaining subsets will not be disabled (hence the term "stubborn") by such a firing. On the contrary, all transitions from one such subset are dependent and can disable each other. They must all be fired in the given marking in order not to lose any possible branch in the reachability graph. Applying the same procedure to every marking, we can generate only a part of the reachability graph. But, as was shown in [44], if the net has a deadlock, this deadlock will always be reached in such a graph. A pseudo-code description of the reduced reachability graph generation is given in Figure 9. As can be observed, it is an extension to the DFS reachability graph generation algorithm. The difference is that only enabled transitions belonging to a stubborn set of marking m_i are fired.

The reduced reachability graph, based on stubborn sets, for the example from Figure 1(a), is shown in Figure 10(a). It should be obvious that this net has no deadlocks and is persistent.

The advantage of full reachability analysis is the availability of complete information about all pos-

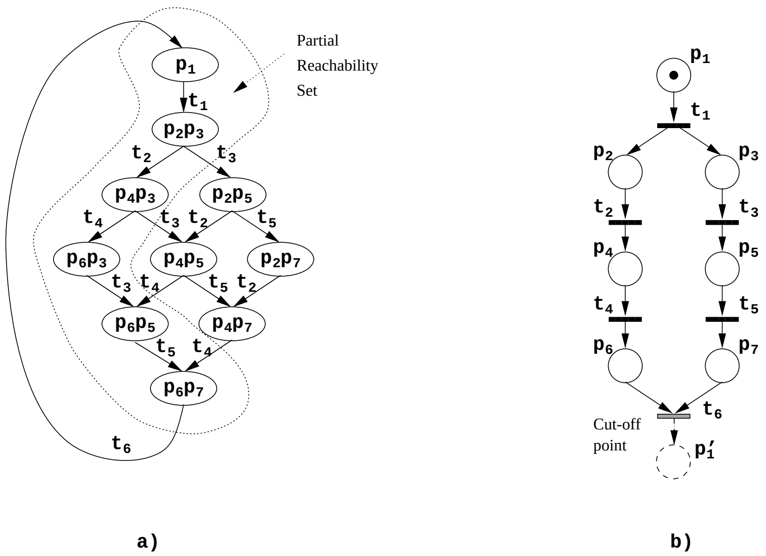

Figure 10: Illustration for the stubborn set (a) and unfolding methods(b).

sible state sequences of the modelled object. Such information may be necessary to check consistency of the state encoding or other properties. The disadvantage is the size of the full graph, which for large systems may be hard to generate and analyse. The use of partial representation may be restricted only to some properties. For example, it is insufficient to check consistency of the state encoding if we analyse the STG specification of the circuit. It may also be inadequate to perform timing analysis of the model, since, by operating with states, it hides the causal and ordering relations between events. On the other hand, the practicality of the stubborn sets method depends upon the efficiency of the computation and selection of the stubborn sets for each marking explicitly generated. We have successfully applied this analysis technique and its associated software tool, called PROD [12], to verify correctness of a self-timed (priority-based) token ring interface protocol [48]. Using the main deadlock detection framework, we have also been able to check other properties, such as mutual exclusion and correct priority order in the arbitration protocol, by reducing their checking to a deadlock detection problem.

3.2 Symbolic traversal

The second approach [30] constructs a representation of reachable markings or states by means of boolean characteristic functions. Such functions are defined on the set of variables which denote places. For any reachable marking m, a boolean function is built, in which each place p_i marked with a token is denoted by its literal p_i, whereas each empty place p_j is denoted by its complement literal p'_j.

For our example from Figure 1(a), the set of reachable states is represented by the following boolean function: $\mathcal{F}_{m_0[\rangle} = p_1p'_2p'_3p'_4p'_5p'_6p'_7 + p'_1p_2p_3p'_4p'_5p'_6p'_7 + p'_1p'_2p_3p_4p'_5p'_6p'_7 + p'_1p_2p'_3p'_4p_5p'_6p'_7 + p'_1p'_2p'_3p_4p_5p'_6p'_7 + p'_1p'_2p_3p'_4p'_5p_6p'_7 + p'_1p_2p'_3p'_4p'_5p'_6p_7 + p'_1p'_2p'_3p'_4p_5p_6p'_7 + p'_1p'_2p'_3p_4p'_5p'_6p_7 + p'_1p'_2p'_3p'_4p'_5p_6p_7$.

Characteristic function construction. This function can be represented by a Binary Decision Diagram (BDD) [30], which allows efficient operations on boolean functions. Such operations are necessary for performing checks on the properties of the reachable set. The actual construction of $\mathcal{F}_{m_0[\rangle}$ is carried out, starting from the initial marking's function, by cyclically applying the *transition function* to the set of new markings generated at the previous step. The sets of new markings are generated layer by layer. The transition function, a boolean characteristic function which defines the new layer from the previous layer of markings, is defined as a composition of two boolean operators:

boolean factorisation and boolean product. The transition function is calculated for $\mathcal{F}_{\mathcal{M}}$ (defining the set of markings M) and each transition t, by using the following formula:

$$[\mathcal{F}_{M_{E(t)}}\mathcal{P}(t)]_{\mathcal{S}(t)}\mathcal{A}(t)$$

Here,

$E(t)$ is a function describing the boolean condition under which transition t is enabled;

$\mathcal{F}_{M_{E(t)}}$ (called the cofactor of $\mathcal{F}_M$ with respect to $E(t)$) selects those markings in $\mathcal{F}_M$ in which t is enabled;

$\mathcal{P}(t)$ ("no predecessor is marked") helps to eliminate tokens from the predecessor places of t;

$\mathcal{S}(t)$ ("no successor is marked") helps to cofactor the function with respect to the successor places of t;

$\mathcal{A}(t)$ ("successor places") adds tokens in all the successor places of t.

We now show how the set $\mathcal{F}_{m_0[\rangle}$ is produced for our example. We start with the characteristic function of the initial set of markings, which is $\mathcal{F}_{m_0} = p_1p'_2p'_3p'_4p'_5p'_6p'_7$. For instance, let us calculate this function for transition t_1. $E(t_1) = p_1$, and $\mathcal{F}_{m_0}{}_{E(t_1)} = p'_2p'_3p'_4p'_5p'_6p'_7$. Then,

$$\begin{aligned}
\mathcal{F}_{m_0}{}_{E(t_1)}\mathcal{P}(t_1) &= p'_1p'_2p'_3p'_4p'_5p'_6p'_7, \\
[\mathcal{F}_{m_0}{}_{E(t_1)}\mathcal{P}(t_1)]_{\mathcal{S}(t_1)} &= p'_1p'_4p'_5p'_6p'_7, \\
[\mathcal{F}_{M_{E(t_1)}}\mathcal{P}(t_1)]_{\mathcal{S}(t_1)}\mathcal{A}(t_1) &= p'_1p_2p_3p'_4p'_5p'_6p'_7
\end{aligned}$$

where $\mathcal{A}(t_1) = p_2p_3$. The final step produces the function characterising the set of markings reachable from m_0 by firing t_1. It is easy to see that none of the other transitions generates any new markings from the $\mathcal{F}_{m_0}$ set. Thus at the next step, the same transition function is calculated for the newly produced set $p'_1p_2p_3p'_4p'_5p'_6p'_7$ and each transition, generating the set defined by $p'_1p'_2p_3p_4p'_5p'_6p'_7 + p'_1p_2p'_3p'_4p_5p'_6p'_7$. Then, the same action is applied to this set, and so on, until no more new markings are produced. The procedure subtracts the set of already generated markings, in order to avoid unnecessary repetition.

This technique is called BDD-traversal of the reachable set of markings. If a net is interpreted as an STG, it is not only possible to generate the symbolic image of the reachable markings, but also the binary codes of states associated with the reachability graph.

The result of a BDD traversal can be used to check for deadlocks and non-persistency by writing simple boolean functions characterising these properties [17]. For example, the set of deadlock markings in $\mathcal{F}_{m_0[\rangle}$ can be found by intersecting $\mathcal{F}_{m_0[\rangle}$ with the product of all $E'(t)$, for each transition t, where $E'(t)$ stands for the complement of $E(t)$, effectively denoting the condition under which t is **not** enabled.

The non-persistency check, with respect to some transition t, is performed by finding a set of markings in which t is enabled with any other transition t' and then calculating the set of markings reachable by firing t' and checking that t is not enabled in them.

The major advantage of this approach is that, although it "covers" the whole reachability set, it does not work with individual markings and edges between the marking nodes in the reachability graph. By operating with sets and layers of markings and using their boolean characterisation, the analysis process may achieve high performance. Existing BDD manipulation packages support such efficiency [23]. By covering all reachable states, one can guarantee to check the consistency of the state encoding for an STG check, and other properties required for synthesis of circuits from STGs (see Section 4).

The shortcoming of the approach is as follows. Without precise information about connectivity between individual markings, on cannot check the various liveness properties, including the absence of

```
while QUEUE is not empty do
  for each transition t in PN N do
    find an untried subset of independent occurrences of •t
    if such a set exists then do
      insert t' into the QUEUE in order of size of its local configuration
    end do
  end do
  pull the first transition t' from the QUEUE
  if t' is a cutoff point then do
    copy transition t and its t• into unfolding and connect them.
  end do
end do
```

Figure 11: Algorithm for unfolding generation.

livelocks (i.e. cyclic behaviours involving only a subset of transitions). Likewise, the partial reachability analysis cannot provide enough information about the causality and ordering relations between events required to perform timing analysis of circuit models.

The practical efficiency of this method depends on the size of the BDDs produced during the traversal and analysis. It was shown in [30] that performance strongly depends on the ordering of variables in the BDD used for the reachable set. A number of heuristics, based on the structure of the original net, have proved to be effective [17]. The actual calculation of these heuristics can however be a problem and in general requires a substantial knowledge of non-local information about the net, such as temporal relations between its transitions.

3.3 Partial order analysis

This approach draws upon the construction of the so-called *true concurrency* semantics of the Petri net model of a circuit. For any two concurrent transitions, the partial order representation does not need to interleave them in the same way as both of the above approaches do.

The partial order representation is given by the Petri net *unfolding*. Any Petri net can be unfolded into an acyclic Petri net, called occurrence net [28]. Since the unfolding may be an infinite object (for a Petri net with cyclic behaviour), it is not suitable for practical purposes of analysis. Therefore, some truncation of the net is applied, which is aimed at producing sufficient information to perform analysis. The notion of "sufficiency" is based on the idea of covering all reachable markings, which would have been produced if the reachability graph was generated instead of the unfolding. The method was proposed in [22]. It was shown that properties of the reachability set and its markings can be reformulated in terms of the unfolding.

Although some of the algorithms, such as deadlock detection, may require exponential time complexity, the actual representation of the net semantics is very compact – linear to the size of the original net.

For the net shown in Figure 1, the unfolding is shown in Figure 10(b).

Truncated unfolding. The original algorithm for building a truncated unfolding in [22] works for any class of net, and terminates if the net is bounded. Its truncation technique is based on the following criterion, called *cut-off condition*. Every time the algorithm generates an instance of a transition t_1', denoted in the unfolding by t_1' it first checks if the marking produced by firing this transition (called the *final state of the local configuration* of t') is equal to the marking generated by any of the transition occurrences, say t'_2, already constructed. For example, in the fragment shown in Figure 10(b), the transition occurrence t'_6 satisfies the first part of the cut-off condition, since its final state p_1 is equal to the initial marking $m_0 = \{p_1\}$. Note that the initial marking is declared to be the final state of the auxiliary *initial transition*, which is created purely for the sake of uniformity.

The second part of the cut-off condition for a transition instance t'_1 is that the size of the set of predecessors (called the *local configuration* of t'_1) is greater than the size of the local configuration of t'_2. It is important to note that the unfolding construction algorithm sorts all the transition occurrences that are due to be included on the basis of the size of their local configuration. This guarantees that by the time the candidate t'_1 is considered the t'_2 must have already been included.

It is easy to notice that t'_6 in our example satisfies the second part of the cut-off condition, which makes it an actual cut-off point. Note that this transition is the only cut-off point in this example.

It has been noted in [39] that the original cut-off condition is overly strong for some classes of Petri nets and may lead to construction of a truncated unfolding with redundant instances. The redundant instances will be added to the unfolding but do not add any new reachable markings to the set of already visited ones. A new cut-off point condition was suggested in [39] for general Petri nets and in [10] for safe Petri nets.

Since the unfolding contains all temporal relations between transition occurrences, one can use this information even if the actual property check is done by means of one of the previous approaches. The use of such relations may improve the efficiency of the symbolic traversal method. Indeed, one can always find a set of mutually concurrent places (that can be simultaneously marked with tokens) and partition the overall set of places onto the set of subsets of mutually inconcurrent ones. This information can then be applied to the ordering of the (place) variables in the BDD representations. Some successful results have been recently obtained [41] on this complementary use of net unfolding and symbolic traversal.

Timed Petri net unfolding. Analysis and verification of timed systems have recently attracted attention of the research community. Ordinary Petri nets and their related formalisms can be used for modelling of untimed systems. However, their application to modelling of untimed systems is often limited if not impossible. The size and complexity of such Petri net grows significantly even for simple examples. On the other hand, most of the asynchronous circuits today are designed with delay assumptions in mind. Additional circuitry is required to made an asynchronous circuit purely Delay Insensitive or Speed Independent. Thus the designers often go for shortcuts, and design their circuits whilst taking gate switching and environmental delays into account. An example of such a circuit is the Fast Forward Latch control circuit that can be found in [11].

Timed systems and hence asynchronous circuits with delay assumptions can be modelled with *Time Petri nets.* A Time Petri net incorporates timing information by assigning a range $[d(t_i), D(t_i)]$ to each transition of the underlying Petri net where $d(t_i)$ and $D(t_i)$ are called *earliest* and *latest* firing time respectively. We can view the whole system as a set of local clocks associated with each transition. These clocks are started when all its input places are marked, i.e. it is *untimed-enabled.* Each transition is said to be *time-enabled* if it is untimed-enable and it has been untimed-enabled for at least $d(t_i)$ units of time. A time-enabled transition must fire by $D(t_i)$ units of time from the moment of its untimed enabling unless it has been disabled by the firing of another transition. A time-enabled transition firing changes the current marking by removing tokens from its input places and inserting tokens into its output places. The firing also affects the set of started clocks by removing the clocks of fired transition and adding new clocks for all newly enabled ones. The firing itself is instantaneous.

Each state of a Time Petri net, or *time state*, consists of two parts: a marking m_i and a set of clock values of all enabled transitions. A state may change if some, time bounded by the firing times of enabled transitions, passes or a transition fires. There may be uncountable number of time states. In order to be able to represent them as a finite set and analyse them, a notion of *Time State Graph* is introduced. Each vertex of this graph is a class of equivalent time states. It has been shown that the set of classes of time states is finite for finite and bounded Time Petri net [2].

Asynchronous circuits were analysed using the Time State Graph in [51]. However, construction of the Time State Graph hits the problem of state explosion. The *Time Petri net unfolding* method has been applied to analysis of the Time Petri net based models of asynchronous circuits and their specifications in [40]. The idea behind this method is similar to untimed unfolding; the Time State Graph is represented implicitly by means of *time configurations*, a configuration along with a class of time states associated with it. The properties needed for correctness behaviour verification, such as safeness and signal persistency, can be detected from the unfolding by structural analysis.

Another modelling technique is based on Timed Petri nets, where the timing information is associated with places. A partial order based analysis technique has been described in [13]. This method constructs a set of alternative untimed processes and then applies an algebraic approach to separation times analysis.

4 Synthesis techniques

In Section 2 we identified ways in which asynchronous circuits and Petri nets can behaviourally mimic each other. The set of Sutherland's components can be formally mapped into a set of Petri net fragments (Figure 2), which appear to be generic enough to suggest the idea of a reverse mapping. Finding such a reverse mapping, or translation of a Petri net specification of a circuit into its logic implementation, is the objective of asynchronous circuit synthesis. In this section we review the major approaches to synthesis of asynchronous control circuits from Petri net specifications. These fall into the following two categories, reflecting the use of either the two-phase or the four-phase signalling discipline:

1. Syntax-directed translation of Petri nets into control circuits,
2. Synthesis from Signal Transition Graphs.

4.1 Syntax-directed synthesis

This approach starts with a Petri net specification and constructs a circuit which effectively "simulates" the net, by replacing its components with circuit elements. The net *must be safe* in order to make sure that the circuit produced is meaningful. Within this approach two alternative threads can be identified. They are described in the following subsections.

4.1.1 "Place-to-latch" circuit compilation

The first direct translation method is based on the idea of "physical simulation" of every reachable marking of a Petri net in terms of the state of the circuit. This is achieved by associating each place in the net with a memory latch, i.e. SR-flip-flop, and transitions with appropriate logic at the inputs of the latches. One of the examples of this style is described in [45]. Several types of flip-flop reflecting the AND-causal logic on transitions are shown in Figure 12. The state of signals corresponding to the markings of the Petri net fragments is shown in brackets, near the appropriate wire. The signals between adjacent cells that are currently in the process of switching are indicated by the associated transitions from logical 1 to logical 0.

The arrival of a token in a place, say $p1$ of Figure 12(a), is manifested in the circuit by setting the corresponding SR-flip-flop (whose direct and complementary outputs are labelled by $p1$ and $p1'$, respectively) to the state $p1 = 1, p1' = 0$. This state can be shifted into the next flip-flop $(p2, p2')$, thus modelling the arrival of a token into place $p2$, after which the state of $(p1, p1')$ will return to $p1 = 0, p1' = 1$.

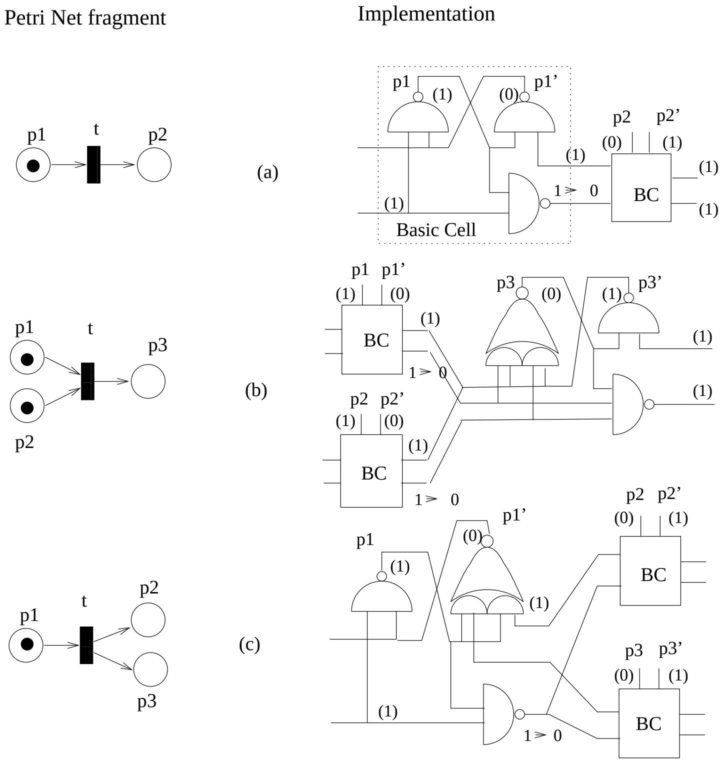

Figure 12: Syntax-directed translation from Petri nets based on a "place-latch" relationship.

The arrival of a token in place $p3$ in Figure 12(b) is dependent upon the presence of tokens in places $p1$ and $p2$. This is implemented by the appropriate sum-of-product logic at the gate that generates output $p3$. The resetting of the $(p1, p1')$ flip-flop in Figure 12(c) back to the state $p1 = 0, p1' = 1$ (corresponding to the situation when the token has been removed from place $p1$ after the transition has fired) is only possible after both $(p2, p2')$ and $(p3, p3')$ have been set to states in which $p2 = 1, p2' = 0$ and $p3 = 1, p3' = 0$. The sum-of-product logic of the gate implementing $p1'$ facilitates this effect.

When this implementation style is employed, it is crucial to note an important and inevitable discrepancy between the firing semantics of net transitions and the physical nature of the flip-flop switching process. The abstract character of transition firing assumes the removal of tokens from input places and addition of tokens to output places as a simultaneous and indivisible action. In circuits, this action is split into subactions. The output place flip-flops are first set to logical 1, and then the input place flip-flops are reset to logical 0. To cope with this discrepancy without further semantic complications, one has to make sure that the original net never reaches a marking in which any input place of a transition is marked with a token simultaneously with any of its output places.

For a practical example of the use of this translation method the reader is referred to [48]. Here, a major part of the speed-independent control logic of a self-timed token-ring adaptor is synthesised from a refined labelled Petri net description of the ring protocol. The method proved to be most effective for a controller of that level of complexity. Other techniques, e.g. those based on Finite State Machines or Signal Transition Graphs, would have been faced with the computationally hard problems of state assignment and hazard-free implementation. They would not guarantee a speed-independent solution whose size would be linear in the size of the Petri net specification. Compared to the methods described below, this synthesis technique is clearly more applicable to control circuits of relatively large size and which are not too critical with respect to speed and area optimality.

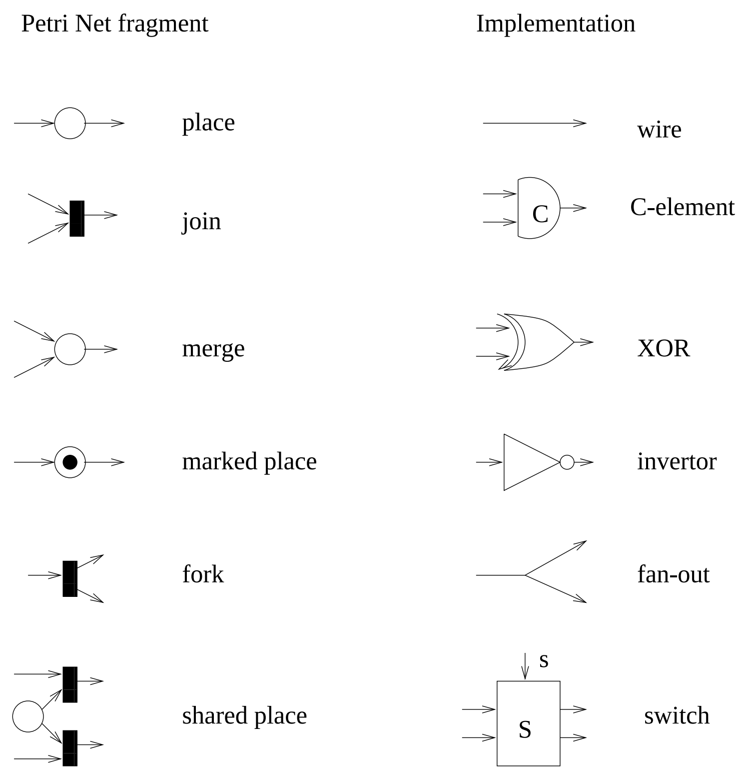

Figure 13: Syntax-directed translation based on the "transition-signal event" relationship.

4.1.2 Event-based (two-phase) circuit compilation

Another way to directly "simulate" the control flow in Petri nets is similar to the one described in Section 2, with the exception that we now have to view the circuit-to-net from the reverse perspective. This technique assumes that there is no semantic difference between the rising and falling transitions of the control signals. The firing of a transition is associated with the edge of the corresponding signal. This strategy originates from [31]. Figure 13 shows Patil's "mapping" of primitive fragments of Petri nets into event-based circuits.

Patil's mapping is applicable to a structural subclass of nets known as *Simple Nets*. Such nets are characterised by the following condition: for every transition, at most one input place may also be an input place for another transition. Figure 14 shows an example of a Petri net which is not a Simple Net. Its transition t has two input places, $p1$ and $p2$ which are both input places for other transitions. The reasons for this restriction are quite obvious. It is not possible to use an interconnection of two simple 2-way Switch components to implement a fragment such as the one shown in Figure 14.

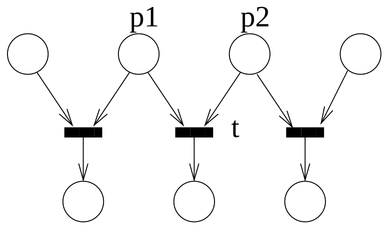

Figure 14: A fragment of a net which is not Simple Net

Extensions to Patil's mapping. There are certain conditions under which such a structurally non-simple fragment can be implemented in a different way (not as a Switch element). Under certain behavioural conditions a shared input place fragment of a simple and safe net can be implemented without the use of a Switch, whose internal structure requires a mutex element. In some cases this implementation degenerates into a pair of C-elements (see Figure 15,(a),(b)). This may not always work, since we cannot guarantee that the phases of input signals arriving at each C-element are appropriate. For example, the marking of a shared place of the two transitions $p2$ may not always occur in correspondence with the phase of the input y as required by the mutual phasing of two inputs of the same C-element $x1$ and y. For example, let us assume that transition $t1$ becomes enabled for the first time, and the corresponding C-gate is enabled when both $x1$ and y change their value from 0 to 1. Then, when a token arrives in $p2$ (this means that input y becomes 0 again) for the second time, it may either again assist in enabling t, if $p1$ is marked, or this time $p3$ may be marked. In the former case the upper C-element will switch back, restoring the output $z1$ back to 0. In the latter case, however, the inputs of the lower C-gate will be unmatched, $y = 0$ and $x2 = 1$. As a result, a deadlock may arise in the circuit which does not show up in the Petri net model. Furthermore, if the change of $x2$ to 1 arrives before the resetting of y to 0, a premature transition may be generated at the output $z2$.

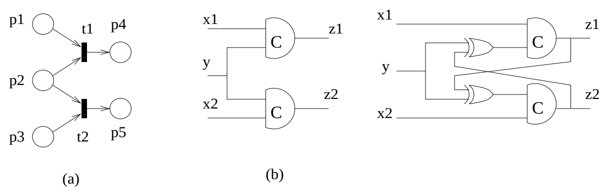

Figure 15: Problems with C-elements; need for Decision-Wait element

The above problems can be checked by means of behavioural analysis of the STG. This allows us to verify the polarity of the signals corresponding to a place marking. This can be extracted from the Petri net reachability analysis – only if $t1$ and $t2$ alternate at their even firings (i.e. $t1$ must fire even number of times before $t2$ is enabled, and vice versa), we can use C-gates. In all other cases, a more versatile component must be used, called *Decision-Wait (DW)*. It allows synchronisation of an event on one signal out of a group of mutually exclusive signals with an event on another signal out of another group of mutually exclusive signals. This synchronisation is done irrespective of the actual phases of signal transitions, purely on event basis. Figure 15(c) shows the internal implementation of the 2-by-1 Decision-Wait. The internal structure of this element is not fully speed-independent. The correctness of its actions depends on the delays of the XOR gates. These gates must have relatively small delay compared to the delay of the environment of the DW element, which must not switch inputs $x1$ and $x2$ too fast after outputs have changed. An arbitrary n-by-m DW can be quite complex internally, especially if one wants a totally speed-independent implementation [14]. Another useful application of a DW element is illustrated in Figure 16, where the initial net fragment is not a Simple Net. Here, provided that the shared input places which form structural conflicts can be split between two groups where the places are mutually exclusive, we can implement such a fragment by an appropriate n-by-m DW element.

With the syntax-directed approach, most transformations, such as decomposition and refinement, have to be done at the Petri net level. Correctness checks can then be performed by applying composition and verification techniques. Alternatively, one may try to decompose or optimise the design by performing transformations at the circuit level. This is rather risky since it can destroy the clear semantic relationship between the net and the circuit. Some of these transformations may optimise the overall design by recognising slightly more complex fragments of Petri nets and compiling them directly into 'chunkier' elements, such as Select, Toggle, Call (recall Section 2, Figure 2). Their possible implementations can be found elsewhere [32].

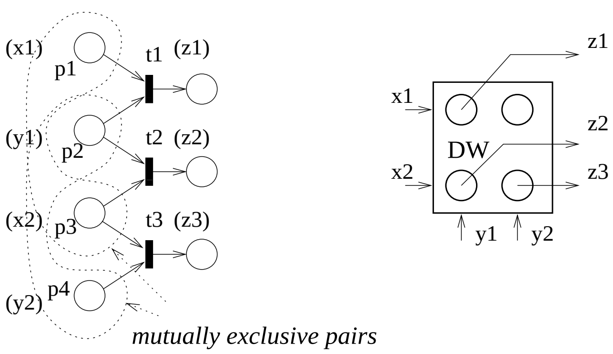

Figure 16: Use of Decision-Wait elements for non-simple nets

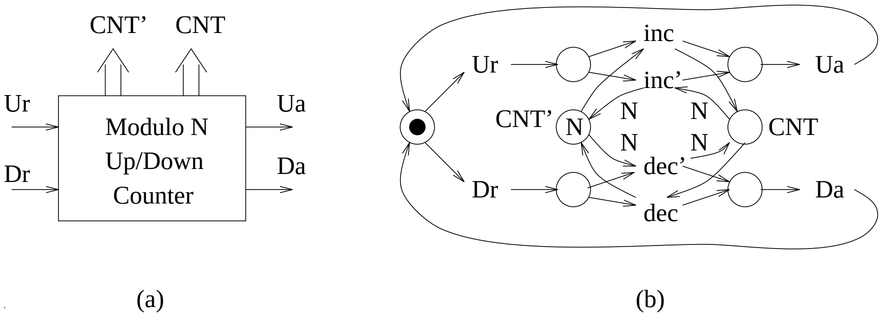

Figure 17: Top level Petri net model of a modulo-N Up/Down counter.

4.1.3 Direct synthesis example: modulo-N Up/Down counter

As an example, let us consider the synthesis of a modulo-N Up/Down counter that uses the event-based signalling discipline. We assume that the environment always guarantees mutual exclusion between sending requests for Up and Down operations.

Let N be a power of 2 for the sake of simplicity. The Petri net specification of the counter at the highest level of hierarchy is shown in Figure 17. This net models the case in which the counter increments its integer value represented by place CNT, and decrements its complement in place CNT', when a request signal arrives on input Ur (meaning a request to count Up). An acknowledgement is then produced on output Ua. This continues until the number of occurrences of Ur is not greater than the number of occurrences of Dr (meaning request to count Down) by N. If the latter happens, which is associated with the state in which CNT' contains no more tokens while CNT contains N tokens, the counter removes all N tokens from CNT back to CNT'. The counting is therefore cyclic, without indication of the "empty" and "full"' states. Similar cyclic action takes place if the counter decrements. To allow this, our net model employs so-called *weighted flow relation arcs*. A transition with weighted (say with N) input and/or output arcs is enabled if the corresponding input place contain at least the relevant number (N) of tokens. The process of firing such a transition removes N tokens from the input place and adds N tokens to the output place.

To implement this behaviour, we decompose the counter into a modulo-2 Up/Down counter stage and a modulo-N/2 Up/Down counter, as shown in Figure 18. The former, which produces the least significant bit of the counter, has to produce four completion signals, instead of just two for the complete counter. These four signals are formed by the two pairs (Ua1, Uc1) and (Da1,Dc1).

Signal Ua1 is an acknowledgement which does not need to be carried forward through the higher stages, where the counter is incremented in the state with the least significant bit equal to 0. Signal Uc1 is a carry signal which is produced when the current stage was incremented while being in the state 1. Signals Da1 and Dc1 have similar functionality, Ack and Carry, respectively, for the decrement operations.

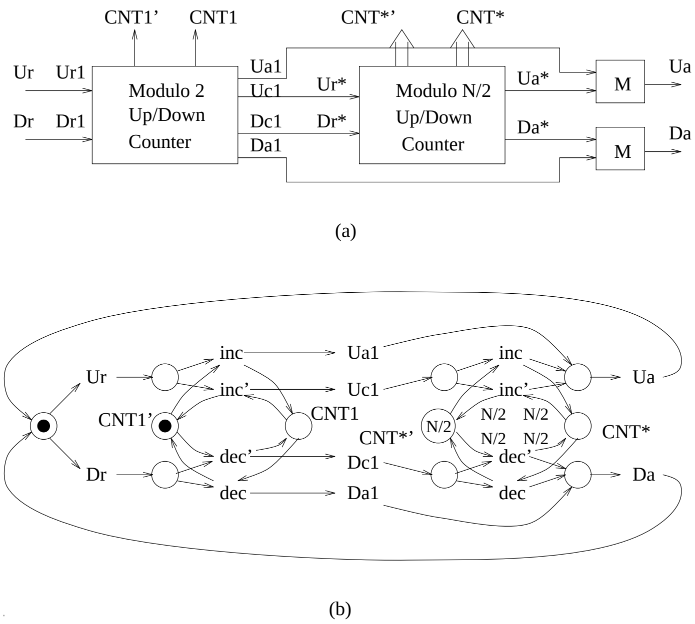

Figure 18: Decomposition of a modulo-N Up/Down counter.

The net model of the first stage of the circuit, factored out, is shown in Figure 19(a).

In order to implement this net by a circuit, we first transform it to an equivalent (with exactly the same behaviour on the set of its signal labels) net, shown in Figure 19(b). In this new net, each signal event has a unique transition associated with it and we can apply the syntax-directed translation of it into a circuit, using the component models of Figure 2. Note that in this circuit, shown in Figure 19(c), we use a 2-by-2 Decision-Wait (DW) element, which is a special case of a general n-by-m DW [14]. The general n-by-m DW is a matrix-shaped element with one input for each row and each column and one output for each combination of row and column. It expects exactly one of its row and one of its column signals, and produces the corresponding output in the intersection of the row and the column. In the 2-by-2 case, the DW synchronises an event occurring on exactly one of the two inputs B+ and B− with either event on Ur (Up-counting) or on Dr (Down-counting).

The various lower level implementations of the 2-by-2 DW can be found in [14].

It can be easily noticed that the output of the Merge can be used as a level signal CNT, thus making the task of combining the control and data parts very simple.

We can now, and further recursively, refine the modulo-N/2 counter in a similar way, until N/2 becomes equal to 1, so finally obtain the entire circuit implementation. This design is speed-independent (as guaranteed by our net decomposition) since the change of the level signal, CNT, in each bit stage is always acknowledged by the corresponding completion signals from the stage. The signals Ua1,Uc1,Dc1 and Da1 always change last in each stage.

4.2 Synthesis from Signal Transition Graphs

This approach takes the circuit specification in the form of an STG. Such an STG can be obtained either (i) independently as an initial specification model or (ii) from the original, more abstract specification by means of signal expansion.

In the first case, the STG is often the formal capture of a timing diagram description of the protocol of the interaction between the circuit and its environment. The overall process of STG-based synthesis is illustrated in Figure 20. The necessary and sufficient condition for the implementability of an STG (with bounded underlying Petri net) in the form of arbitrary logic gates, is two-fold:

1. validity, or consistency of the state coding of its State Graph, and

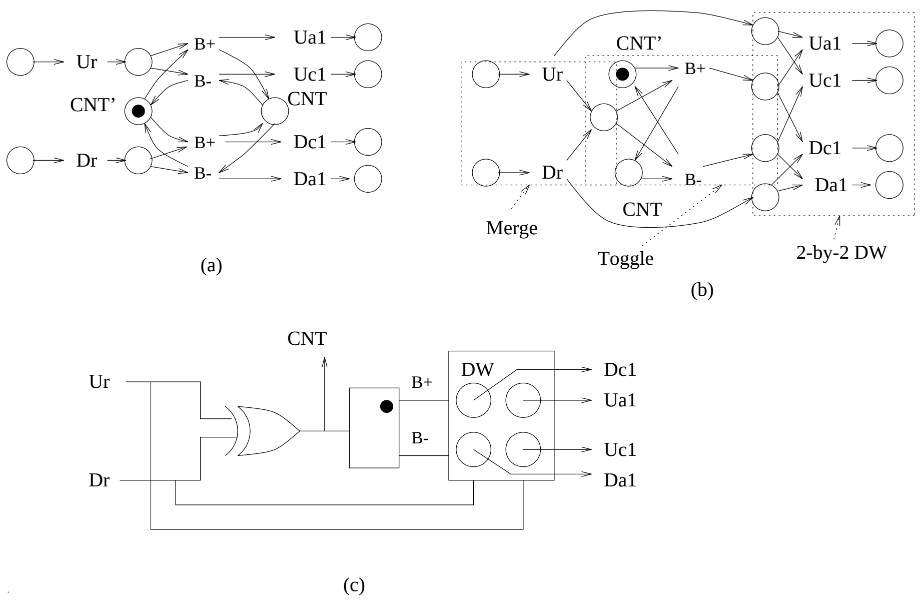

Figure 19: Circuit implementation for one bit Up/Down counter stage.

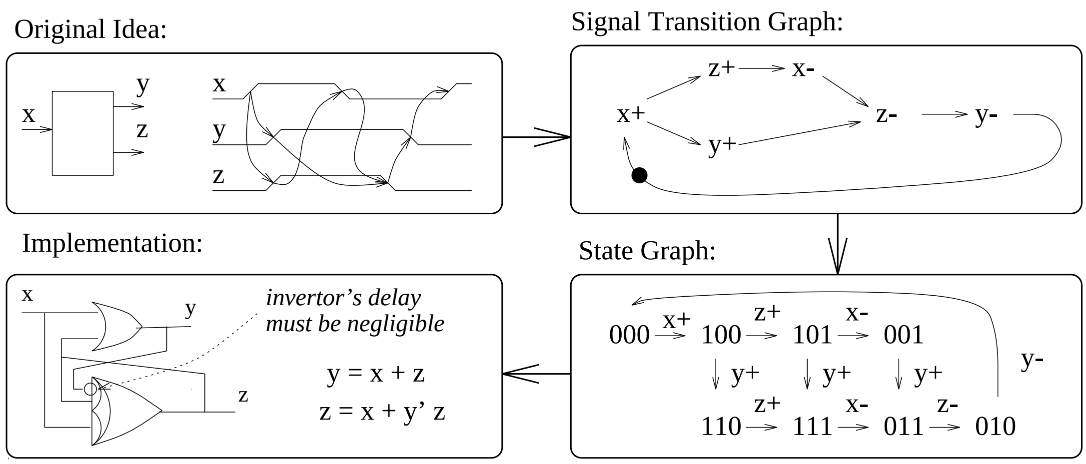

Figure 20: STG-based circuit synthesis.

2. Complete State Coding, i.e. the condition under which each state uniquely defines the *next-state* function for all non-input signals.

If the second condition does not hold for the original STG, it is usually possible to add extra internal signals into the model, without changing its original ordering of events. Once the Complete State Coding property is established, each non-input signal is derived as a boolean function from the State Graph, using boolean minimization techniques described, for example, in [21].

There are a number of methods, either at the STG, State Graph or even logic circuit level, to map such a rather abstract design into a specific implementation architecture. We do not present them here and refer the reader to [1, 25, 21, 16, 37].

Assume that the STG model is obtained from the original labelled Petri net through a signalling expansion [47]. Each signal event in the Petri net is explicitly represented by either a 0-to-1 or 1-to-0 transition of the corresponding binary variable. In order to separate the phases, thereby refining a two-phase model into a more general four-phase, one often has to unfold the original Petri net into two periods. With respect to any signal, each period stands either for the rising or the falling edge of this signal.

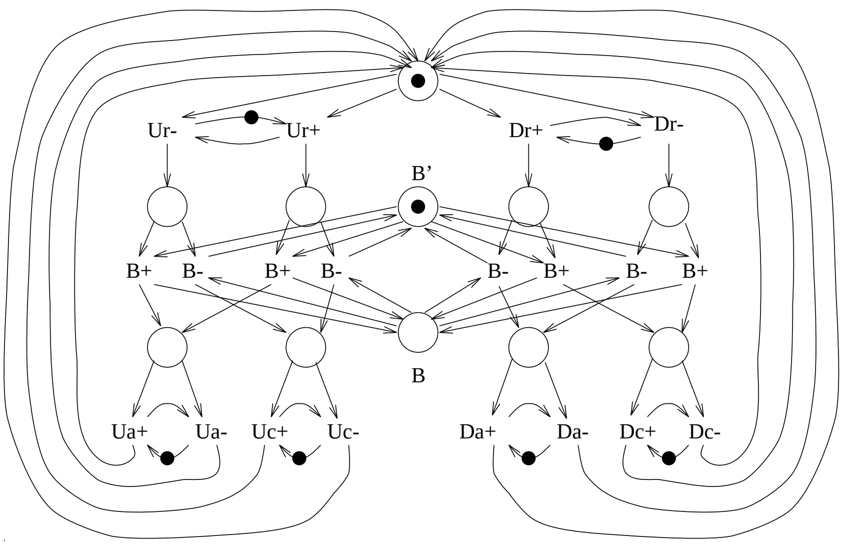

Figure 21: Signal transition graph for one bit counter stage

A circuit for a one bit stage of an Up/Down counter, different from the one presented in the previous section, can be obtained by converting the Petri net shown in Figure 19(a), into an STG with four-phase signalling. This is purely a syntactic transformation, which does not affect the actual operational idea of transition signalling. The STG for one stage of the counter is shown in Figure 21.

This STG generates, through the above-mentioned synthesis process, the following set of equations for the counter:

$$\begin{aligned}
U_a &= B\ (D_r\ U_c\ \overline{U_r} + \overline{D_r}\ \overline{U_c}\ U_r) + (\overline{B} + D_r\ U_c + \overline{D_r}\ \overline{U_c} + U_c\ \overline{U_r} + \overline{U_c}\ U_r)\ U_a \\
U_c &= \overline{B}\ (D_r\ \overline{U_a}\ U_r + \overline{D_r}\ U_a\ \overline{U_r}) + (B + D_r\ \overline{U_a} + \overline{D_r}\ U_a + U_a\ \overline{U_r} + \overline{U_a}\ U_r)\ U_c \\
D_c &= B\ (D_r\ \overline{D_a}\ \overline{U_r} + \overline{D_r}\ D_a\ U_r) + (\overline{B} + D_a\ \overline{D_r} + \overline{D_a}\ D_r + D_a\ U_r + \overline{D_a}\ \overline{U_r})\ D_c \\
D_a &= \overline{B}\ (D_r\ \overline{D_c}\ U_r + \overline{D_r}\ D_c\ \overline{U_r}) + (B + D_c\ \overline{D_r} + \overline{D_c}\ D_r + D_c\ \overline{U_r} + \overline{D_c}\ U_r)\ D_a \\
B &= D_r\ \overline{U_r} + \overline{D_r}\ U_r
\end{aligned}$$

These equations can be implemented either as *complex gates*[5], with internal feedbacks, or as a row of SR-latches with separate two-level or multi-level logic for the excitation functions S and R. These can easily be obtained from the above equations. For example,

$$\begin{aligned}
S_{U_a} &= B\ (D_r\ U_c\ \overline{U_r} + \overline{D_r}\ \overline{U_c}\ U_r) \\
R_{U_a} &= \overline{(\overline{B} + D_r\ U_c + \overline{D_r}\ \overline{U_c} + U_c\ \overline{U_r} + \overline{U_c}\ U_r)}
\end{aligned}$$

In the latter case, one cannot guarantee that the implementation is speed-independent with respect to the delays of the AND and OR gates involved in the implementation of the S and R functions. Special hazard analysis and elimination techniques must be employed [21].

To summarise, we should note that both direct translation and STG-based synthesis techniques are often complementary rather than competing to each other. Indeed, the capability of the existing automated STG-based synthesis tools, such as SIS [42] and ASSASSIN [50], do not in practice allow the designer to implement circuits with more than some 25-30 logic elements. The recent experiment with synthesis of control circuits for a Sproull Counterflow Pipeline has confirmed these limitations [46]. On the other hand, direct compilation has no such limit. Therefore, it appears to be quite natural to use the direct compilation approach at the higher design level, or at the level of fast prototyping, and then gradually optimise the design by re-synthesing its fragments by means of STG-based logic synthesis.

One of the examples of such an approach is described in [19, 20]. Using partitioning and 3D synthesis methods [52] this method has been shown successful in synthesysing large examples of "Huffman style" circuits. Combined use of both approaches, at different levels of control flow, has also been reported in [48] and in [33, 15].

5 Conclusion

We have presented a number of techniques that can be used in the application of Petri nets to the modelling, analysis and synthesis of asynchronous circuits. They exploit the event-based character of asynchronous circuits. Petri nets are shown to be equally applicable to the modelling of two-phase (micropipeline) and four-phase (based on logic gates) circuit components. Whether the former or the latter design style is predominant affects the decision as to which synthesis approach should be pursued. The use of micropipeline components tends to follow the way in which the Petri net specification is decomposed and finally "compiled" into the corresponding inteconnection of two-phase components. The use of logic gates implies circuit synthesis from a Signal Transition Graph interpretation of Petri nets. This is clearly demonstrated in our modulo-N Up/Down counter example.

Another important issue is the choice of a most appropriate method for the verification of the Petri net model of a circuit. This can be a behavioural specification intended to be implemented by a circuit using one of the synthesis techniques outlined in Section 4. Alternatively, it can be a net reconstructed from an already built circuit in the way shown in Section 2. In either case, the most important questions to be answered are what type of properties must be verified and what class of net is subjected to analysis. As we showed in Section 3, in some cases the various analysis techniques can be used in combination [41]. For example, the hazard properties of circuits can be verified by checking for net safety using the unfolding technique, while deadlock-freedom can be verified by means of the symbolic traversal of the reachable set of markings. The latter method may also benefit from the information supplied by the unfolding, which can can help in obtaining a more efficient variable ordering in the BDD representing the reachability space.

Our current research is aimed at creating a set of tools supporting automated design of asynchronous control circuits from Petri net specifications, at various levels of abstraction. We believe that there is a widely held view that asynchronous design is much more complex than synchronous. This problem cannot be resolved without the use of powerful tools for the graphical capture and manipulation of behavioural models.

References

[1] P. Beerel and T. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings of ICCD-92*, pages 581–586. Computer Science Press, 1992.

[2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[3] G. Birtwistle and Y. Liu. Specification of the Manchester Amulet1: Top Level Spec. Manuscript, November 1994.

[4] T.-A. Chu. On the models for designing VLSI asynchronous digital systems. *Integration: the VLSI journal*, 4:99–113, 1986.

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

[6] P. Day and J.V. Woods. Investigation into macripipeline latch design styles. *IEEE Transactions on VSLI Systems*, 3(2):264–272, June 1995.

[7] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* The MIT Press, Cambridge, Mass., 1988. An ACM Distinguished Dissertation 1988.

[8] J. C. Ebergen. *Translating programs into delay-insensitive circuits.* Centrum voor Wiskunde en Informatica, Amsterdam, 1989.

[9] M.A. Escalante and N.J. Dimopoulos. A probabilistic approach to timing analysis for synthesis and its application to microprocessor interface design. Technical Report ECE-94-6, The University of Victoria, Department of Electrical and Computer Engineering, 1994.

[10] J. Esparza, S. Römer, and W. Vogler. An inprovement of mcmillan's unfolding algorithm. Technical Report TUM-I9599, Insitute Für Informatik, Technische Universität München, 1995.

[11] S. Furber et al. Amulet1 workshop: Presentation materials. Technical report, Manchester University, Department of Computer Science, Amulet Group, Lake District, England, July 18-22 1994.

[12] P. Grönberg, M.Tiusanen, and K.Varpaaniemi. PROD: - A Pr/T-net reachability analysis tool. Series B: Technical Reports 11, Helsinki University of Technology, June 1993.

[13] H. Hulgaard. *Timing analysis and verification of timed asynchronous circuits.* PhD thesis, University of Washignton, December 1995.

[14] C.R. Jesshope, I.M. Nedelchev, and C.G. Huang. Compilation of process algebra expressions into delay insensitive circuits. *IEE Proceedings-E*, 140(5):261–268, September 1993.

[15] T. Kolks, S. Vercauteren, and B.Lin. Control resynthesis for control-dominated asynchronous designs. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 232–243, March 1996.

[16] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proceedings of DAC-31*, pages 56–62. Computer Science Press, 1994.

[17] A.K. Kondratyev, J. Cortadella, M. Kishinevsky, O. Roig, E. Pastor, A. Taubin, and A. Yakovlev. Checking stg implementability by symbolic BDD traversal. Accepted for EDAC'95, August 1994.

[18] P. Kudva and V. Akella. A technique to estimate power in asynchronous circuits. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async94)*, pages 166–175, November 1994.

[19] P. Kudva and G. Gopalakrishnan. A technique for synthesizing distributed burst-mode circuits. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU95).*, pages 221–222, November 1995.

[20] P. Kudva, G. Gopalakrishnan, and E. Brunvand. Performance analysis and optimization of asynchronous circuits. In *International Conference on Computer Design (ICCD)*, pages 442–446, October 1994.

[21] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits.* Kluwer Academic Publishers, 1993.

[22] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. *Formal Methods in System Design*, 1995. (to appear).

[23] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

[24] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, NJ, 1989.

[25] C. W. Moon, P. R. Stephan, and R. K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[26] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.

[27] S. Nicklin. Petri-net Modelling of the AMULET1 Address Interface. Lake District, England, July 18-22 1994.

[28] M. Nielsen, G. Plotkin, and Winskel G. Events structures and domains. *Theoretical Computer Science*, 13(1):85–108, 1980.

[29] C.D. Nilesen and M. Kishinevsky. Performance analysis based on timing simulation. In *Proceedings of the Design Automation Conference*, San Diego, CA, June 1994.

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

[31] S. S. Patil and J. B. Dennis. The description and realization of digital systems. In *Proceedings of the IEEE COMPCON*, pages 223–226, 1972.

[32] N.C. Paver. *The Design and Implementation of an Asynchronous Microprocessor.* PhD thesis, University of Manchester, 1994.

[33] M.A. Pena and J. Cortadella. Combining process algebras and petri nets for the specification and synthesis of asynchronous circuits. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 222–232, March 1996.

[34] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[35] T.G. Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, December 1993.

[36] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *International Workshop on Timed Petri Nets, Torino, Italy*, pages 199–207, 1985.

[37] M.H. Sawasaki, Ch. Ykman-Couvreur, and B. Lin. Externally hazard free implementations of asynchronous circuits. In *Design Automation Conference (DAC95)*, pages 718–724, June 1995.

[38] C.L. Seitz. *System Timing*, chapter (Chapter 7). Addison-Wesley, 1980.

[39] 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.

[40] A. Semenov and A. Yakovlev. Verification of asynchoronous circuits using time petri net unfolding. In *Design Automation Conference (DAC96)*, June 1996. Accepted paper.

[41] A. Semenov and A. Yakovlev. Combining partial orders and symbolic traversal for efficient verification of asynchronous circuits. In *Proceedings of the International Conference on Computer Hardware Description Languages (CHDL'95)*, Chiba, Japan, September 1995.

[42] E.M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R.Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of Californica, Berkeley, May 1992.

[43] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. Turing Award Lecture.

[44] A. Valmari. Stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322, 1991.

[45] V. Varshavsky, M. Kishinevsky, V. Marakhovsky, V. Peschansky, L. Rosenblum, A. Taubin, and B. Tzirlin. *Self-Timed Control of Concurrent Processes.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990. V.I. Varshavsky, Ed.

[46] A. Yakovlev. Designing Control Logic for Counterflow Pipeline Processor Using Petri Nets. Technical Report Series 522, Department of Computing Science, University of Newcastle upon Tyne, May 1995.

[47] A. Yakovlev, A.M. Koelmans, and L. Lavagno. High level modelling and design of asynchronous interface logic. *IEEE Design & Test of Computers*, 12(1):32–40, 1995.

[48] A. Yakovlev, V. Varshavsky, V. Marakhovsky, and A. Semenov. Designing an asynchronous pipeline token ring interface. In *Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, May 1995*, pages 32–41. IEEE Computer Society Press, May, 1995.

[49] A. V. Yakovlev and A. Petrov. Petri nets and parallel bus controller design. In *International Conference on Application and Theory of PetriNets, Paris, France*, pages 344–364. IEEE Computer Society, June 1990.

[50] C. Ykman-Couvreur, B. Lin, and H. De Man. ASSASSIN: A Synthesis System for Asynchronous Control Circuits. Technical report, IMEC vzm, Design Methodologies for VLSI Systems Division, 1995.

[51] T. Yoneda, I. Honma, and B.-H. Schlingloff. Verification of bounded delay asynchronous circuits with timed traces. Technical Report 94TR-0013, Tokyo Institute of Technology, August 1994.

[52] K.Y. Yun. *Synthesys of asynchronous controllers for heterogeneous systems.* PhD thesis, Stanford University, August 1994.