Petri Nets and Digital Hardware Design

Alexandre V. Yakovlev and Albert M. Koelmans

Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU, England

Abstract. Petri nets are a powerful language for describing processes in digital hardware, and particularly asynchronous or self-timed circuits. Self-timed circuits are designed to operate without the use of a global clock signal. Applications for such circuits are likely to increase during the next decade, due to problems with on-chip event coordination as VLSI technology approaches a density of one hundred million transistors per chip. Designing such circuits without help of formal tools does not seem to be possible. We present an overview of the methods for specification, verification and synthesis of asynchronous circuits with the aid of Petri nets. We present a number of design examples which are used to illustrate the authors' belief that Petri nets could become widely accepted by digital system designers as a design method.

1 Introduction

1.1 Role of hardware in modern systems

Modern computers, telecommunication systems, items of consumer electronics, and many other examples of systems controlled by a "silicon brain", have complex, multilayered architectures, implemented partly in hardware and partly in software. The hardware and software parts used to be clearly separated both in the tasks they had to perform and in the way they were designed, implemented and tested. This situation is now changing rapidly. Software design technologies have moved into the abstractions of object-orientation and distribution, and hardware design methodologies are following. Consider for instance the world of consumer electronics. The challenges posed by these applications are enormous. Such systems must for example be able to perform a billion operations per second to cope with video applications. To allow this, many of their functions must be embedded in hardware. Making them as cheap as possible means designing and fabricating them in a short time, say with a concept-to-manufacture cycle of less than one year.

1.2 Role of hardware design tools

With the advent of submicron VLSI technology, which will soon enable hundreds of millions of transistors to be placed on a single chip, hardware design is facing new challenges [113, 143]. To cope with this complexity, and with the need to produce new VLSI designs efficiently, designers must be provided with adequate

development techniques and tools. What is needed is a well-integrated design system that combines simulation, synthesis, verification and testing capabilities. Such an environment must allow examination of the behaviour of a whole system, redesign of some of its parts, and reconfiguration of other parts if necessary. It must perform all of these tasks efficiently and without loss of accuracy. At present, the electronics industry often lacks feedback in the design process. It is often realised that previous design decisions were wrong at a point in the design cycle when it is too late or too costly to change anything. The designer should be able to model the system at most levels of detail before commiting to a particular design route. An important issue is that of the reusability of design components. While in software design the notion of a reusable component is hardly new, the circuit designer has always been faced with the problem that circuit details are heavily dependent on the details of a particular silicon technology. It is therefore crucial for design methods to support the necessary modularity and technology independence.

1.3 Role of modelling language

Design methods and tools require appropriate modelling and specification techniques. These techniques must be formal and rigorous, but also easy to comprehend. In the past, logic designers used to draw logic gate diagrams directly from semi-formal specifications, defined by timing diagrams. The only way to prove that the circuit performed correctly was to observe its behaviour with a logic analyser. Today, the specification may not necessarily be defined at the level of signal waveforms. It may be presented in a much more abstract form, similar to a software specification. It is the designer's responsibility to refine the original specification in such a way that functional equivalence is preserved. Those refinements may differ in speed (say, degree of parallelism between individual component actions) and implementation cost (e.g., the number of gates or layout area). These factors should be allowed to influence the design process, and the modelling language must account for such non-functional design qualities.

1.4 Why Petri Nets

Why are Petri nets a good formalism to assist the designer and support hardware design tools? The language of Finite State Machines (FSM) has also been used by digital designers. Unlike timing waveforms, FSMs allow formal specification of and reasoning about hardware. Many existing software tools support the synthesis, verification and testing of systems using FSM representations [114]. So, why are FSMs not sufficient, and for what type of hardware are Petri nets more appropriate?

Why not Finite State Machines? The main characteristic of the FSM-based approach is that the system defined as an FSM is sequential. For a given input signal and a particular state, the system may move to another state and produce an output signal. While the system is performing a state transition, its inputs are

assumed to be stable. The process of changing state is seen as an atomic action. Even if the FSM representation allows modelling of non-sequential effects such as races, which are caused by concurrent transitions of state vector components, the model still uses the notion of a global state, and any concurrent signal transitions are modelled as a set of possible interleavings of state transitions.

The FSM approach suggests the notion of an FSM *composition* for the modelling of large systems. Corresponding verification techniques and tools are essentially based on the theory of FSM products, which usually leads to a combinatorial state explosion. In order to bridle the complexity of the composition, the FSM approach must avoid the explicit construction of a product FSM, but then it faces the problem of adequate interpretation of concurrency, parallellism, and synchronisation between transitions in the different FSMs.

Petri nets can act as FSMs if the modelled system is totally sequential. However, if there is an explicit need to model concurrency without showing it in its interleaving form, even for the purposes of a more compact representation, Petri nets are adequate for doing so.

What are the advantages of Petri Nets? The area of hardware design has traditionally been a fertile application area for research in concurrency and Petri nets using new ideas in modelling and verification [51, 86, 72, 73]. Similarly, the theory and practice of digital system design has always recognised Petri nets as a powerful and easy-to-comprehend modelling tool [66, 93]. In this paper, when we talk about the use of Petri nets and other models of concurrency in hardware design, we assume almost everywhere *asynchronous* circuit design[1]. The reasons for this are twofold. Firstly, any *synchronous* circuit, i.e. a circuit operating under the control of a global clock signal, can be considered as a special case of an asynchronous system, where the clock signal is just an additional event-generating component. The second reason is due to a crucial similarity between asynchronous hardware and Petri nets. The paradigms of asynchrony and concurrency are intrinsic to the behaviour of both. It would be very difficult to talk about concurrency in the presence of a clock signal used as a global event scheduler. Thus, it would probably be less interesting to apply Petri net modelling techniques to the analysis of clocked circuits.

1.5 Historical Review

We present here a brief historical outline of the relationship between Petri nets and hardware design developed in the last four decades.

1950's and 1960's: Foundations. The earliest work was done by D.E. Muller on the theory of asynchronous circuits. The notions of concurrency, conflicts, convergence etc. were developed by hardware researchers a few years before they learned about the net formalism proposed by C.A. Petri. The theory of

[1] This paper also presents (Section 7) a brief overview of existing techniques for the synthesis of *synchronous* controllers from Petri nets.

speed-independent circuits presented by D.E. Muller and W.C. Bartky [85, 76] introduced ideas of feasible sequences, final equivalence classes, confluence, semi-modularity and distributivity. Muller's work was based on the "state-transition" modelling paradigm with its interleaving semantics, because it was mostly an analysis-oriented investigation. However, it gave rise to new ideas in synthesis, too. For example, the language of change charts [36, 119], apparently the first "condition-event" approach to circuit specification, was formally related to Muller's state-based circuit classification [76]. The 1960's were therefore the time when the idea of expressing concurrency in its natural form was fostered amongst digital design theoreticians fairly independently from the first work on net theory. Eventually, the elegance and simplicity of a net form was duly appreciated by circuit designers. For example, C. Molnar and his colleagues began to use Petri nets, with signal names annotating net transitions, to specify the interface behaviour of a circuit. At the same time, work of R.M. Karp and R.E. Miller [50] on parallel program schemata established a very important link between a formal model of concurrency and its interpretation (which could be arbitrary, e.g. that of an asynchronous logic network).

1970's: Towards Parallel Computations. When research into Petri nets grew in the 1970's, it quickly became the choice formalism for research into data flow computers and distributed architectures. Several illuminating structural methods for logic synthesis with Petri nets [79, 97, 34] and related formalisms, such as parallel flow graphs [26], were developed. These methods originated from the seminal MAC project at M.I.T. led by J.B. Dennis. Almost simultaneously, Petri nets gave rise to an alternative structural approach, developed at the Aerospace Research Centre in Toulouse [15]. Such structural techniques are now usually refered to as methods of *direct translation of a (behavioural) specification into the circuit implementation*, so as to differentiate them from the logic synthesis methods developed later. Work by J.R. Jump and P.S. Thiagarajan [48, 49] played an important role in bridging the idea of interfacing speed-independent circuits and the notion of composition in a class of labelled marked graphs. Another good example is work of M. Yoeli [142]. In parallel, new techniques for designing asynchronous control structures, very much in the style of Petri net based methods, had emerged [10]. Structural methods were studied and enhanced with additional modelling constructs and circuit components in the USSR-based work on aperiodic automata [2, 127], led by V.V. Varshavsky. In the UK, work on an asynchronous computer [83] and a design language called MUDL at Manchester University stimulated the use of Petri net models [52]. An interesting method for modelling and analysis of switching circuits with Petri nets was proposed in Germany [37]. Timed Petri nets were developed and applied [100] for the purpose of performance analysis. Despite their elegance and formal clarity, these methods were not very efficient from the point of view of system size and performance. They also completely relied on the designer's experience if model optimisation was required. They were not supported by software tools for the exploration of large state spaces and the solving of complex optimisation tasks.

1980's: First progress in VLSI design. The first book on very large scale integrated (VLSI) systems design, written by C. Mead and L. Conway, appeared in 1980 and quickly became a bible on such design. Notably it included a special chapter on self-timed circuits, written by Ch. Seitz [105], which prophesied the increasing role of asynchronous systems in future generations of hardware, and called for models and methods to make their design efficient. It was an inspiring call for Petri net users. At the same time, the 1980's saw Petri nets gradually evolving into an independent computer science subject. One of the most remarkable lines of research, into the semantics of concurrency [45, 89], led to the exploration of similarities and differences between Petri nets and logic circuits. For example, the notion of atomicity in transition firing and conflict resolution in Petri nets was a high level abstraction of physical effects in circuits. This generated a number of theoretical problems affecting the use of Petri nets for the verification and synthesis of asynchronous circuits [120]. By the end of the 1980's, which saw enormous progress of VLSI technologies and the emergence of powerful software for logic synthesis and verification, Petri nets had attracted attention as a potential practical tool for hardware design. The first work on Signal Transition Graphs, both in the USSR [104, 63] and the USA [20, 18, 19], laid a foundation for their long-term exploitation in VLSI design. Initial attempts to design asynchronous designs with timing constraints specified in Petri net models were also made in [104]. The design-oriented links between Petri nets and self-time circuits were demonstrated in one of the most comprehensive monographs on asynchronous design [127].

1990's: Towards powerful design tools. In the 1990's, strengthened both descriptively (high-level nets) and analytically (new semantics and related verification methods), Petri nets are being used ever more widely. For instance, high-level nets have already helped to tackle the modelling and verification of very complex hardware [115]. Signal Transition Graphs and their close relative, Change Diagrams [55], have uncovered numerous problems relating to the synthesis of asynchronous circuits under bounded and unbounded delays and their hazard-free implementation. These problems required new methods and algorithms for checking various properties of interpreted Petri nets and their respective state graphs, such as consistency and completeness of state assignment, and monotonicity of boolean covers. In pursuit of efficient analysis and synthesis procedures, new methods were developed, such as structural analysis [94], symbolic traversal [96], and partial order (unfoldings) [75, 74]. Analysis of Petri net models with time annotation, a traditionally challenging area of research, has found its application in the analysis and synthesis of hardware designs with timing constraints [44, 87]. The problem of producing hardware implementable event-based specifications has been greatly assisted by the progress in the theory of regions and Petri net synthesis from transition systems [31, 30, 27, 90, 23].

This overview underlines the importance of the long-term relationship between Petri nets and hardware design, and its benefits for bridging the gap between computer science and electronic engineering. Some of the techniques, especially those concerned with modular synthesis of circuits by means of syntax-

direct translation from Petri nets, are only familiar to a limited audience. Recent research has focused on the logic synthesis approach, under the assumption that this is where the real power of Petri nets and Signal Transition Graphs lies. Other approaches, such as Communicating Processes and Process Algebras, are often seen as better suited to design at higher level of abstraction, and hence are predominant in the area of syntax-directed synthesis. Such a subdivision of the "spheres of influence" is in our opinion unfair, and restricts the genuine potential of Petri nets.

The remainder of this tutorial is organised as follows. Section 2 presents a general introduction to the principles underlying asynchronous circuits. Section 3 introduces design transformations, which are used as a first step towards the synthesis of the final circuit. Section 4 gives an overview of the abstract design stage. In section 5, logic synthesis is discussed in detail. Finally, in sections 6 and 7, we briefly discuss software tools and synchronous design strategies.

2 Asynchronous Circuits

This section presents an introduction to the principles of circuits designed to operate *without a clock signal*. Such circuits or systems, traditionally called *asynchronous*, are also called *self-timed* [105] or *self-clocking* [2]. This section will firstly present an informal overview of what an asynchronous circuit is. Then, a number of advantages of such circuits over their clocked counterparts will be examined. This will be followed by reasons why the main focus should be on control logic rather than datapath logic. We will conclude this section with a classification of asynchronous design stages and a presentation of examples. This section is therefore mostly addressed to readers with a limited background in digital design.

2.1 What is an asynchronous circuit?

An asynchronous circuit can be regarded as a hardwired version of a parallel distributed program [4, 16], in which statements or actions are activated if their preconditions are true. However, unlike parallel programs, which normally exist on top of some run-time mechanism, asynchronous circuits do not need an underlying mechanism. Their "statements" are their own physical components, such as logic gates, memory latches, or complex hierarchical modules. These components have inputs and outputs which are connected by means of wires. The role of the data exchanged between them is played by switching events that occur on the interconnection signals. The conditions that activate these modules are *caused* by similar events on their inputs. These conditions are evaluated by

[2] We hope that the reader, particularly the reader without a special hardware background, will appreciate the difference between this interpretation of "synchronous versus asynchronous" and the one often used in referring to different types of interaction between system components. For instance, in the area of real-time systems, the term "synchronous" is usually connected with the "rendez-vous" type of interaction.

the components in much the same way as the above-mentioned preconditions in parallel programs.

Physical level. Although we talk about "parallel programs" of the lowest possible level, this level is still a logical abstraction. Systems built from logic gates are themselves models of the real hardware, which behaves according to the laws of physics! Strictly speaking, in order to fully investigate the dynamic behaviour of switching processes in hardware, one should refer to the physical models of the circuits [105]. This can be done by means of systems of differential equations that describe a circuit as a dynamical system [9, 38]. It is convenient to sacrifice some modelling accuracy because of the complexity of the analogue models. These grow enormously with the size of the circuit, which makes analysis of systems consisting of more than a few gates infeasible.

Fortunately, in most cases it is possible to apply a discrete-event abstraction mechanism to asynchronous hardware. We only consider systems at the discrete level, with signals encoded as Boolean variables and switching events as transitions from logical 0 to logical 1 and vice versa, called *up and down transitions*, respectively. There is of course a class of behaviours, traditionally seen as *anomalous phenomena* in digital hardware, which is referred to when the above-mentioned assumptions cannot be guaranteed. Examples of such phenomena are *hazards* and *metastability*. Calling them "anomalous" is not really fair because they are "necessary elements" of concurrency in electronic systems. They can be approximated in discrete terms but only under certain assumptions (see e.g. [12, 137]). However, given the discrete nature of Petri nets, this body of research falls outside the scope of this tutorial. The interested reader may refer to [69, 17]).

Logical level. At the logical level, the behaviour of an asynchronous circuit can be characterised by sequences of up and down transitions on the inputs and outputs of its components. The order between these transitions is not prescribed by any global scheduler or clock, and is determined by the *local* causal relationships between transitions. Such an order cannot be total, due to the locality of dependence between signals, and hence should be considered as partial. When a component is ready to switch its outputs, it does so without any additional enabling factor. By contrast, in synchronous devices switching can only take place when the enabling signal from the clock arrives. Designing a circuit with the ability to act completely on the basis of causal relations between switching events is the essential principle of self-timed design. In many ways, this behaviour resembles that of a Petri net.

Figure 1(a) illustrates the principles involved in the design of a simple asynchronous circuit. The circuit performs the calculation $out = (a+b)*(x+y)$. The major part of the circuit, the *data path*, consists of two adders and a multiplier. In addition, there is a *control* element called C, the *Muller C-element*, that controls the operation of the data path. In order to allow control of the data path, the adders and the multiplier have an extra input called 'req' (for *request*) and an extra output called 'ack' (for *acknowledgement*). A logic block is triggered

when the appropriate signal arrives on the 'req' input. Once the operation is completed, the 'ack' signal is asserted. Since adders have variable completion times, which depend on the values on the input signals and the length of the carry path they generate, the Muller C-element is used to trigger the multiplier only when both adders have completed. Figure 1(b) and 1(c) show the Petri nets for the data path and control logic, respectively. Figure 2 shows an nMos circuit implementation for the C-element. Its functionality is described by the following Boolean equation: $OUT = ab + (a + b)out$, where OUT is the new value of the output, while *out* denotes its previous value, arriving as a feedback at the input. In this circuit, the two cross coupled transistors T1 and T2 form a memory element (a latch). The output of the circuit assumes the value of the inputs when both inputs are equal. The latch preserves the output when one of the inputs changes. So, the output of the circuit changes only when both inputs have changed.

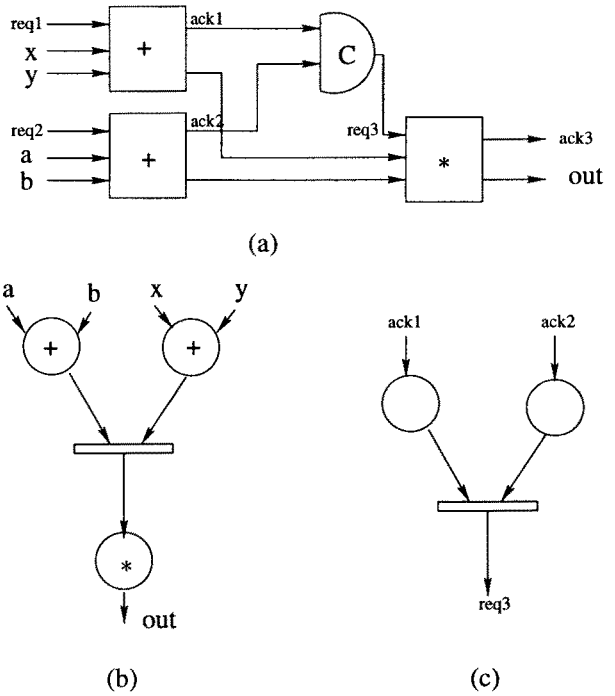


Fig. 1. (a) example circuit (b) data path Petri net (c) control Petri net

Speed-independent and delay-insensitive circuits. Note that self-timed circuits such as the C-element are generally much more robust to variations of delays in their components, gates and wires, than synchronous circuits. The ability to preserve the same partial ordering in their behaviour regardless of component

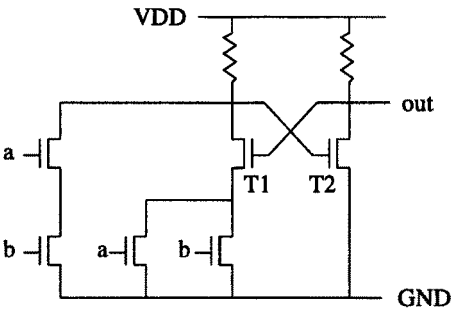


Fig. 2. Circuit implementation of the C-element

delays and variable completion times in the data path is an essential feature of self-timed circuits, making them indeed similar to Petri nets. Depending on the level of delay insensitivity of their behaviour, asynchronous circuits are often subdivided into classes. The most well-known historically is the class of *speed-independent circuits*. Their behaviour is insensitive to variable delays at the outputs of logic gates, although they can be sensitive to variations in the delays of the interconnections between gates. In other words, speed-independent circuits are *hazard-free under the unbounded gate delay model*. A hazard is an anomalous behaviour of the circuit, i.e. a deviation from its normal functioning. A more restricted subclass of circuits, whose behaviour is independent of *both* gate and wire delays, is called the class of *delay-insensitive circuits*. A less restricted class of circuits, which operate with some delay assumptions, is called the class of *asynchronous bounded-delay circuits*. Synthesis and verification of these classes has attracted most of the research in the last decade. Other taxonomies of asynchronous circuit design, such as classes of delay models, different switching semantics, types of causality and their relationship with Petri nets, may be found in [137, 132].

A synchronous implementation of the example circuit of Figure 1 would leave out the C-element and the 'req' and 'ack' signals. Whenever new input values would arrive, the adders would generate new outputs, typically at different times. This in turn would lead to *glitches* on the output of the multiplier, as it would be presented with new input values in rapid succession. The circuit designer would have to ensure that the clock signal of the overall circuit was slow enough to ensure that all glitches had disappeared at the start of the next clock cycle. So, in a synchronous implementation, the clock signal would have to take into account the worst possible delay through the adders, even though the input patterns that would generate such delays would occur rarely. The circuit would thus be idle for significant periods of time. The glitches in the circuit would consume power, which would be wasted. The clock signal itself would use typically half of the power consumed by the entire chip. By contrast, the asynchronous implementation would run at average speed, since it would continue as soon as the adders had completed. Power consumption would be only a fraction of the synchronous

implementation since there are no spurious glitches and no clock.

Let us discuss the arguments in favour of the design of hardware using the priciples of self-timing in more detail.

2.2 Why go asynchronous?

It should be quite clear from the above simple example that implementing the idea of synchronisation between two independent operands with the aid of a clock signal is less natural than with a self-timed two-input C-latch. There are a number of arguments in favour of asynchronous circuits:

- *Performance.* In clocked circuits, the logic is designed to operate in stages. Latches are used to hold input and output data between stages. Data transfer between stages takes place under the control of the clock signal. The period of the clock must be set to the worst case delay in these stages. In asynchronous circuits, modules propagate their switching conditions by themselves. As a result, their activity is limited by actual, not worst case, delays.
- *Power efficiency.* A clocked chip dissipates power even when it does no useful work, simply because the clock beats away and generates enable signals to all parts of the circuit. Typically, the clock will consume half of the total power requirements of a chip. Gating the clock from the idle parts of the logic, e.g. by means of a special "sleep-mode" control signal, alleviates the problem, but cannot solve it radically. An asynchronous chip achieves near-zero standby power in the idle state.
- *Clock skew.* Reliable clock distribution is a big problem in complex VLSI chips because of the clock skew effect. It is caused by variations in wiring delays to different parts of the chip. It is assumed that the clock signal fires off the different stages of the chip simultaneously. However, as chips get more complex and logic gates reduce in size, the ratio between gate delays and wire delays changes so that the latter begin to affect significantly the operation of the circuit. Asynchronous circuits need not deal with clock skew problems, and although they can also be subject to the bigger effect of wire delays, those problems are solved at a much more local level.
- *Metastability.* All synchronous chips interact with the outside world, e.g. via interrupt signals. This interaction is inherently asynchronous. A synchronisation failure may occur when an unstable asynchronous signal is sampled by a clock pulse into a memory latch. Due to the dynamic properties of an electronic device that contains internal feedback, the latch may, with nonzero probability, hang in a metastable (somewhere in between logical 0 and 1) state for a theoretically indefinite period of time. Although in practice this time is always bounded, it is much longer than the clock period. As a result, the metastable state may cause an unpredictable interpretation in the adjacent logic when the next clock pulse arrives. Self-timed circuits wait until metastability resolves. Even though in some (e.g. real-time) applications this may still cause failures, their probabilty is very much lower than in clocked systems, which must trade-off reliability against speed.

- *Modularity.* Different parts of a digital system are usually designed separately. These different parts tend to have different timing constraints. Combining them into a single synchronous circuit can be very difficult, and may result in a complete redesign of the entire system. By contrast, asynchronous designs can be much more easily combined into a single circuit. The only requirement is to make sure that the functional and causal interfaces between the modules are well defined. Since such interfaces are often based on delay-independent *handshake protocols* (cf. the 'req' and 'ack' pairs in Figure 1), self-timed designs can be much more independent of the implementation technology, and thus support the idea of hardware component re-use.
- *Electromagnetic compatibility (EMC).* The clock signal is a major cause of electromagnetic radiation emissions, which are widely regarded as a health hazard or a source of interference, and are becoming subject to strict legislation in many countries. EMC problems are caused by radiation from the metal tracks that connect the clocked chip to the power supply and target devices, and from the fact that on-chip switching activity tends to be concentrated towards the end of the clock cycle. These strong emissions, thus being at the harmonics of the clock frequency, may severely affect radio equipment. This is why it is sometimes not allowed to use portable computers on aircraft. Asynchronous circuits emit much less radiation than synchronous ones.

2.3 Why control logic?

It is quite customary in hardware design to separate the design of *control logic* from that of *datapath logic*. The control logic implements the control flow of the algorithm of the problem specification, while the datapath logic deals with the operational part of the algorithm. In many ways, such a distinction is not absolute. It is perfectly acceptable to consider an application where the datapath may have its own elements of control flow. Some hardware design examples, e.g. an asynchronous bus or ring interface adapter [135, 70, 136], a tree arbiter [35, 138] or a modulo-n counter [29, 131], can be control-flow dominated, with a fairly simple datapath logic. Their control would be usually specified by a combination of partially ordered sets of events. Other examples, such as an asynchronous register bank or a parallel n-bit arithmetic-logic unit [98, 53, 71, 88], would have a fairly simple control behaviour but may be quite complex from the functional point of view.

2.4 Role of Petri nets

For obvious reasons, Petri nets have traditionally been used to aid the design of control logic. Hence the focus of our discussion will be the control flow. In order to design an asynchronous datapath unit, the designer could follow existing structural methods outlined elsewhere [127, 42]. Additionally, the designer would need to specify the protocols between the datapath and the control, using Petri nets. Another reason why control circuits are our main concern here is that their design is particularly difficult. They are behaviourally much more

diverse than datapaths, and hence the use of structural approaches is rather limited. Virtually every new algorithm requires the design of a new controller. This puts tremendous demands on the effectiveness of the tools for verification and synthesis. Compared to other "asynchronous process" languages, Petri nets are in a very advantageous position, because of their ability to represent the paradigms of causality, concurrency, deterministic and non-deterministic choice at any level of granularity and abstraction [137]. They also allow specification of hierarchy and compositionality. For example, when designing an asynchronous FIFO buffer, the move from a fairly abstract level of specification, in terms of actions "put a data item" and "get a data item", to a much lower level, in terms of signal transition events, can be done quite comfortably through changing the basic Petri net notation. Support from existing theory is provided by (i) the *composition* of labelled Petri nets, (ii) the *signalling refinement* of the event annotation, and (iii) the use of *observational equivalence*. Since Petri nets have a clear link with the state-transition notation [90], they provide a semantically rigorous bridge between other description languages and existing asynchronous circuit synthesis tools [23].

2.5 Asynchronous design: abstraction levels and design stages

The overall design flow in a Petri net based system for designing asynchronous circuits is shown in Figure 3. Such a design normally distinguishes between two levels of abstraction and modelling, which are applied during the corresponding design stages. The higher level is associated with the *abstract design* of the control flow. This level deals primarily with behavioural descriptions; the notion of the system structure comes only from the datapath and the way it is referenced in the specification of the control flow. The internal structure of the control path is usually determined by the structure of the behavioural specification of the control flow, its level of compositionality, and the specific interpretation of the abstract actions in terms of the lower level design.

The lower level design stage, called *logic design*, is focused on the transformation of the abstract model of the control flow into the asynchronous control circuit, i.e. into an interconnected set of circuit elements (gates). This transformation consists of two major parts:

- the *signalling refinement* of the abstract behavioural model into its binary signal "equivalent"; this is based on the definition of an actual interface between the control logic and datapath, as well as interface between the abstract components of the control flow in terms of the lower level protocols for up and down transitions of binary signals.
- the *circuit implementation* of the signal-refined behaviour; this part may proceed either as a direct syntax-based translation of the behavioural model or using some logical synthesis techniques; the latter often give a more efficient (in terms of silicon area and performance) implementation than the direct translation methods.

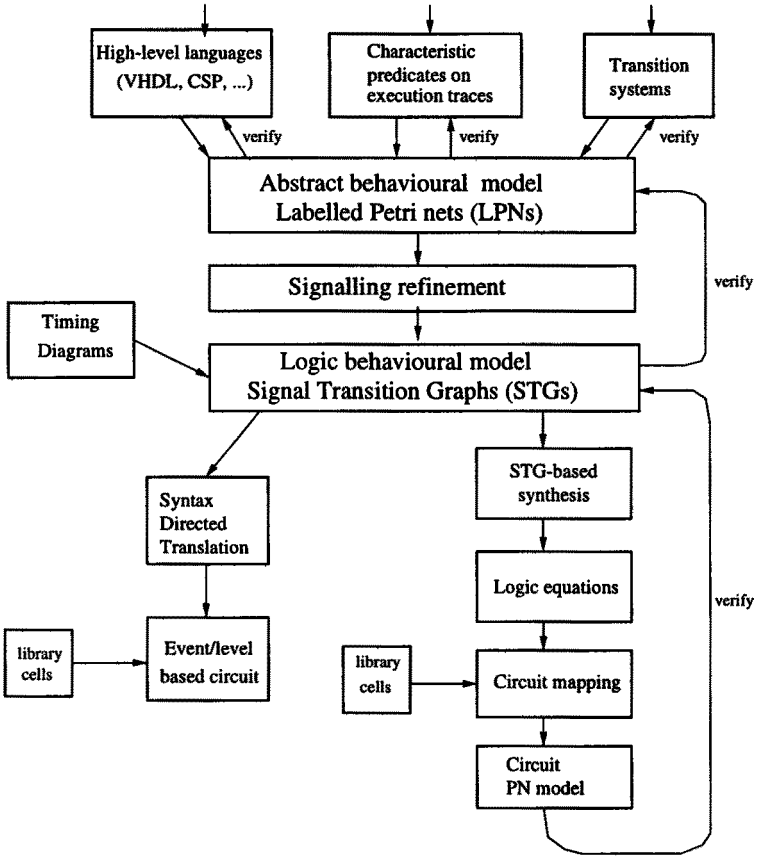


Fig. 3. Design flow of Petri net design system.

To give an initial flavour of the use of Petri nets as a modelling tool at the above-mentioned levels of abstraction, let us consider two relatively simple examples.

2.6 Design examples

Asynchronous processor. At the higher design level the behaviour is defined in terms of an asynchronous process that can be represented by a labelled Petri net. The transitions of such a net can be labelled with the names of relatively abstract operations on datapath or control components. For example, let us consider a high-level design model of an asynchronous processor shown in Fig. 4. At the top abstraction level, the behaviour of a processor consists of two actions, Instruction Fetching (IF) and Instruction Execution (IE), which alternate and are therefore performed sequentially.

We can now refine these actions into subactions according to our ideas about

the processor architecture. Thus, the IF action can be seen as a process, i.e. a Petri net fragment, consisting of the following subactions: incrementing a Program Counter (PC), loading a Memory Address Register with the new address for memory reading (MAR_r), and reading the new instruction word from Memory (Mem). The IE action can be refined into a process (another Petri net fragment) involving other subactions: loading an Instruction Register (IR), decoding, activating and executing the fetched instruction for two possible instruction formats, a one word instruction (1WdInst and 1WdEx) and a two word instruction (2WdInst and 1WdEx). The part of the process concerned with two word instruction execution requires two memory cycles. As can be observed from the analysis of this Petri net, the initial sequential operation between IF and IE has been refined into a model which allows concurrency between actions with smaller granularity. For example, the PC action can be executed concurrently with instruction reading, decoding and execution. Another paradigm appearing at this level is that of choice between two types of instruction execution. The refined model can be subjected to verification (e.g. for absence of deadlocks or undesirable conflicts between actions) and/or performance analysis (e.g., estimation of the degree of concurrency between transitions, evaluating critical paths, simulation). The process of refining the design can be continued until the designer realises that the abstract behavioural model satisfies the desired functional and quantitative requirements. The result of this design stage is a specification of the control flow in such a form that its actions, i.e. transitions in the labelled Petri net model, can be easily mapped onto the primitive operations of the datapath units. This part of the design process is described in detail in [107].

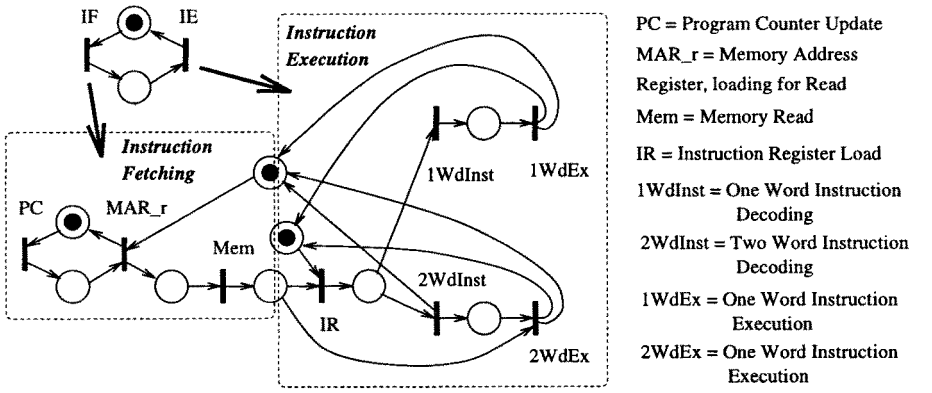


Fig. 4. Petri net example: a high level behavioural model of an asynchronous processor.

VME bus adapter. The second example presents a Petri net model for a logic design level specification. The transitions of such a net will be labelled with the names of binary signal transitions. The example is a simplified version of a hard-

ware adapter which interfaces the VMEbus with a "slave" device, e.g. a memory chip. Such interfaces are typically described directly at the logic design level, by means of timing diagrams, as shown in Figure 5. Informally, the adapter's function is to synchronise two *handshake (request-acknowledgement) protocols*, one at the VMEbus link and other at the link with the device. The first handshake involves bus data strobe signals DSR (read operation) or DSW (write operation) and acknowledgement DTACK. The second handshake involves the local data strobe command LDS and local acknowledgement LDTACK. The process of synchronisation includes an additional signal, DEN, to control data bus buffers. The order of the signal transitions is established in the corresponding timing diagrams by means of arrows. The solid arrows stand for *causality conditions* to be implemented by the adapter circuit. The dashed arrows designate causal relations implemented by the environment, through the above-mentioned handshakes.

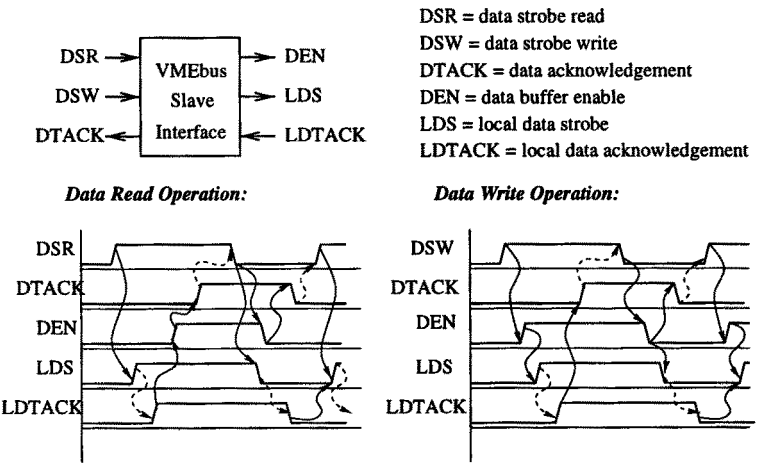


Fig. 5. VME bus adapter example: timing diagrams.

The above behavioural specification can be converted into the Petri net shown in Figure 6. Each transition is labelled with the name of a signal followed by either + or −, depending on whether this is a rising or falling edge. Such a net is called a Signal Transition Graph (STG). The notation used for depicting STGs is essentially a short-hand Petri net notation, where a place with a single input and single output transition is simply replaced by an arc. Note also that transitions are simply represented by their label.

This net, or STG, combines both Read and Write operations into a single model. This is due to the ability of Petri nets to model choice using places with several incident output transitions (e.g. place incident to transitions DSR+ and DSW-). The STG also captures potential concurrency by allowing some transitions to fire independently. For example, the release of signal DTACK followed

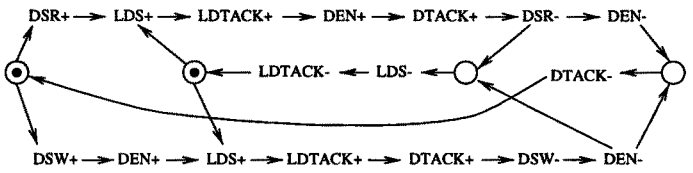
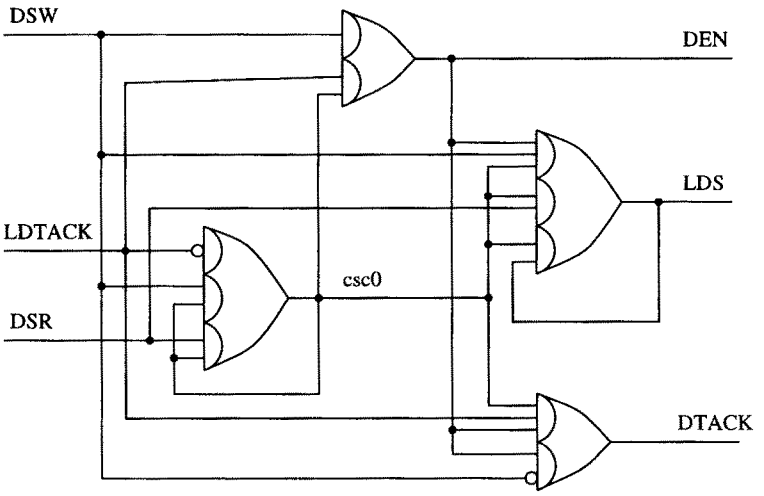


Fig. 6. VME bus adapter example: Signal Transition Graph.

by the assertion of a new strobe signal, DSR or DSW, can be done concurrently with the release of signals in the device handshake, LDS and LDTACK. The completion of the latter is synchronised only at the point where the new activation of signal LDS is required. We recommend that the reader traverse the firing sequences of the net and compare them with the original timing diagram model. This Signal Transition Graph can be implemented in logic using synthesis tools such as SIS or Petrify (see Section 6). The solution, which involves an additional state signal *csc*0, inserted by Petrify for the purpose of appropriate state encoding (see Section 5.4), is shown in Figure 7.



Logic equations (csc0 is additional state signal):

DEN = DSW + LDTACK * csc0
DTACK = DEN * (DSW' + LDTACK * csc0)
LDS = csc0 * (DEN * DSW + DSR + LDS)
csc0 = LDTACK' + csc0 * (DSR + DSW)

Fig. 7. VME bus adapter example: logic implementation obtained by an automatic synthesis tool.

The process of constructing Signal Transition Graphs for this kind of hard-

ware and synthesis of their logic implementations, is described in more detail in [135].

3 Overview of design transformations

In this section we briefly outline the major ideas underlying the overall two stage design process. This outline presents the main design steps involved in applying model transformations. We will use a very simple example which appeals purely to the reader's intuition, and does not require formal knowledge. We then proceed to a more detailed examination of these design steps with the help of formal models.

3.1 Transformations for Abstract Design

The basic idea behind model transformations in abstract design is depicted in Figure 8. Here, the initial requirement is that two actions [3] a and b can proceed in parallel but only once, i.e. for a (or b) to occur again it must wait for the completion of b (a). The *circuit semantics* of the model, used in a subsequent refinement, assumes that actions a and b are started by the designed control circuit. This means that these actions can be refined into so-called *active* handshakes [123]. In such a handshake the first transition (e.g., a rising edge) is produced on the output *request* signal, and it is acknowledged by the environment of the circuit with a transition on the *acknowledgement* wire.

We consider here two possible threads of abstract design. One, called *compositional design*, corresponds to the original idea of control flow being captured in the form of causality constraints between individual actions. It then proceeds through transformation of this knowledge into the form of a labelled Petri net model via steps (1.1) and (1.2). The other thread, called *synthesis from state-based specification*, assumes that the original description is given in an FSM-like form, by a transition system. This model is used as a source for synthesis of a labelled Petri net by means of the theory of regions; these transformations are shown as (1.3) and (1.4). Note that both threads are complementary; we may allow for the application of both at different levels. Indeed, the first one is essentially based on a compositional approach, and is probably more natural to be used at a higher level. Thus, the target labelled Petri net model can be built as a parallel composition of labelled Petri nets for smaller scale control elements. These simpler elements can themselves be built using either transformation thread. Let us look at these threads in more detail.

Compositional approach (Section 4.3). Transformation (1.1) involves the construction of a labelled Petri net model of the control flow from the initial capture of causality constraints. In this example we have two such constraints, which are specified as characteristic predicates on the *numbers of occurrences*

[3] Unless specified otherwise, the terms "action" and "event" are equivalent.

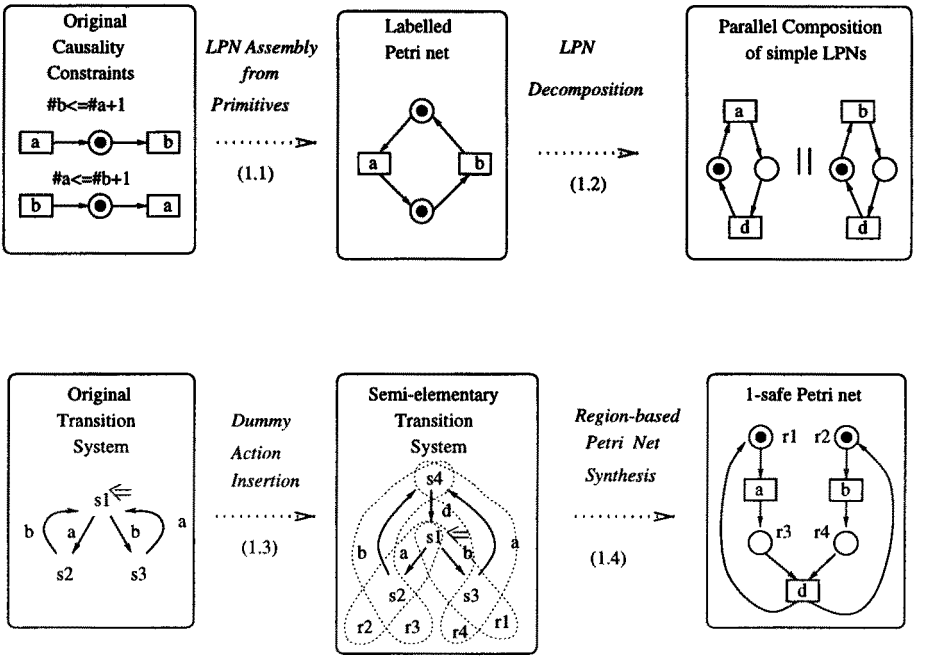


Fig. 8. Model transformations for abstract design.

of events a and b in execution traces (denoted by the symbol $\#$). The first constraint, $\#b \leq \#a + 1$, says that the number of occurrences of event b in any execution trace cannot exceed that of a plus 1. The second condition is symmetrical. Each such constraint can be conveniently captured by a *single place labelled Petri net primitive*. These primitives are composed together by means of merging the transitions corresponding to the same event label. This merge reflects the lowest level at which the parallel composition of nets via transition synchronisation is applied.

Transformation (1.2) illustrates the process of decomposing the labelled Petri net model into a set of nets each of which has a simpler behaviour than the initially obtained net (in step (1.1)). This decomposition is again based on the idea of a parallel composition of labelled nets with synchronisation on transitions with the same label. In our example, the initial net model, whose reachability set consists of three states (cf. states in the transition system used for the second thread) can be decomposed into two nets, each of which has only two states. The nets consist of two transitions each. One of those two transitions in each net is labelled with the same name, d. Thus their parallel composition exploits synchronisation on this label. Furthermore, the original net is behaviourally 2-safe (each place can keep two tokens in some markings), whereas the simpler nets are 1-safe. The notion of 1-safeness is important for the application of some logic design procedures. Note that transformation (1.2) may be based on intuitive ideas

about the control logic structure. For example, each simple net is implemented by its own control logic unit; these units can interact through a handshake port implementing the common transition. Thus, since we do not generally apply any kind of correctness-by-construction principle to this decomposition, and rely on the intuition of the designer, we should assume that the resulting net needs to be verified against the original one. We will show that this verification can be based on the notion of *observational equivalence* [77] between labelled Petri nets. This notion fits well with the concept of conformance tests between the implementation and specification models.

Synthesis from state-based description (Section 4.4). Let us consider the second thread. Transformation (1.3) is applied to a Transition System [4] which does not satisfy a *semi-elementarity* condition, defined in Section 4.4. This is a necessary and sufficient condition for applying further transformations (1.4). To satisfy this condition, we insert at stage (1.3) additional events into the model. These events can be regarded as *dummy* (sometimes also called "silent"[77]) actions. In the same way as labelled Petri nets, the correctness of this transformation will be taken in the sense of observational equivalence between the original and the resultant Transition Systems, which is sufficiently powerful for the purpose of asynchronous design. Both notions (for Transition Systems and Labelled Petri nets) are formally defined below. In our example, an auxiliary event d helps satisfy the semi-elementarity conditions. The new Transition System is observationally equivalent to the original one with respect to the set of events $\{a, b\}$. The reader may note the similarity between the idea of introducing dummy events and that of new transitions with shared labels in the compositional approach (transformation (1.2)).

Transformation (1.4) is based on the notion of regions in a Transition System [90], which are sets of states corresponding to places in the synthesised 1-safe Petri net. If the Transition System satisfies the condition of semi-elementarity, the synthesised net generates a reachability graph which is isomorphic to the Transition System. Thus, due to the property of transformation (1.3), the Petri net should be observationally equivalent to the original description. Note that the event labels of transitions in the original Transition System are used as the unique labels of the events.

3.2 Transformations for Logic Design

The logic design transformations shown in Figure 9 use a labelled Petri net for each control logic unit. Note that each such net may be only a part of the overall net model – due to the compositional approach.

[4] The term "Transition System" is used as a synonym to "State Graph". Only if it may cause confusion, we will apply the latter term in a more specific sense than the former. Namely, a State Graph is a Transition System which has a binary encoding. This follows the terminological tradition established in the asynchronous design community.

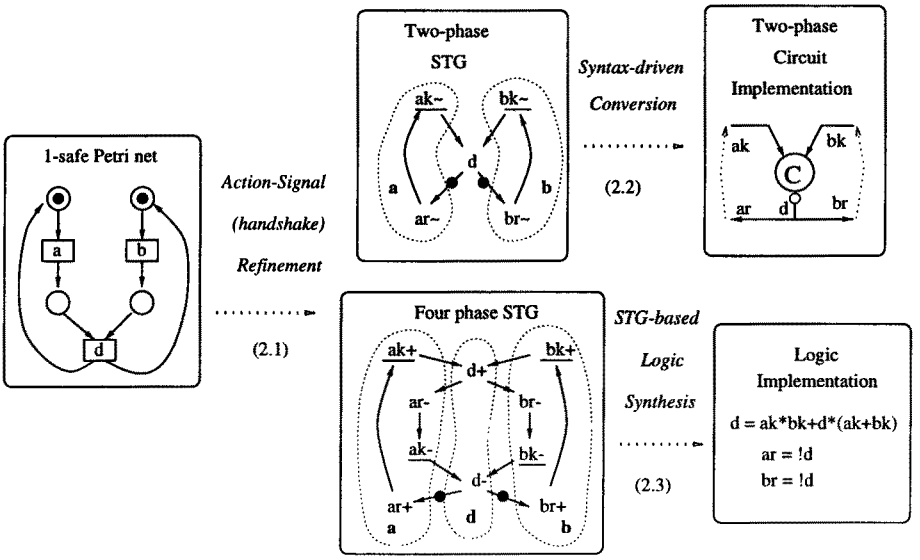


Fig. 9. Model transformations for logic design.

Action refinement (Section 5.2). Transformation (2.1) is an action refinement. It is, however, different from the insertion of dummies in (1.3), since it involves associating an original event name with a set of events. Furthermore, it is performed at the Petri net level. In order to cast it into the notion of observational equivalence, we need to establish a mapping between the set of refined actions and the original actions. For every original action such a mapping should select a *critical* event from the refined set while other events must be regarded as silent actions. The idea of such refinements for labelled Petri nets has been defined in [133, 134]. The refinement can be done in two ways that lead to circuit implementations (2.2) and (2.3). Note that for the example shown in Figure 9 those implementations produce the same result, which is of course not true in general; an alternative implementation, (2.3), is shown in Figure 10.

Direct translation (Section 5.3). The (2.2) label is assigned to the implementation type in which the circuit is obtained by direct, syntax-based, conversion of Petri net fragments into corresponding macromodules in the style of [118] or [97]. The class of 1-safe *simple* [86] Petri nets is sufficient to perform such a conversion [97]. The net, called a two-phase STG in Figure 9, is obtained from the original net in the (2.1) transformation stage by means of: (i) expanding abstract events into pairs of handshake signals (*handshake expansion*) in a *two-phase protocol* (also known as a Non-Return-to-Zero, NRZ, protocol[5]) [118], and (ii) for resolving conflicts with *output signal non-persistency*, by inserting

[5] In such a protocol, both the rising and the falling edges of a signal have equal significance from the semantical point of view.

semaphore actions which are implemented with arbitration elements [25]. In our example, the circuit semantics of events a and b in the original model is such that they correspond to two *active* handshakes. Therefore, they are refined into two pairs of signal transitions ($ar\sim, ak\sim$) (respectively, ($br\sim, bk\sim$)), meaning a request to execute action a (b) and an acknowledgement of its completion. The fact that the request part is leading in those handshakes (since they are both active) is reflected in the relative position of the tokens, i.e. before $ar\sim$ and $br\sim$. (Note also that the input transitions are underlined in the STGs of Figure 9.)

Logic synthesis from STGs (Section 5.4). The (2.3) stage is concerned with synthesis of a logic gate implementation, which is called a four-phase implementation because it is synthesised from an STG in which signals are refined according to a *four-phase protocol* (also known as a Return-to-Zero, RZ, protocol[6]). Similar to (2.2), the (2.3) implementation also requires from the (2.1) refinement that abstract events are expanded into handshakes, and that explicit arbitration actions [25] are inserted. Unlike (2.2), the actual derivation of logic is performed by means of logic synthesis from the STG. This is done with the aid of software tools such as SIS or Petrify [114, 22], which themselves access the logic minimisation package Espresso [8]. In the example, we refine both handshakes into an STG for its four-phase logic synthesis, in a way that is not much different from two-phase signalling. The purpose of this is to benefit from the existence of the auxiliary event d, which can itself be interpreted as an extra state signal, and refined into a pair of transitions $d+$ and $d-$. These are used to help solving the *Complete State Coding* problem, which is a necessary condition for obtaining logic equations for the output signals. Alternatively, by refining only the a and b handshakes, we could completely rely on a synthesis tool, which could solve both the state coding and logic synthesis issues. This is illustrated in Figure 10, where three additional state signals ($csc0, csc1$ and $csc2$) have been added for Complete State Coding.

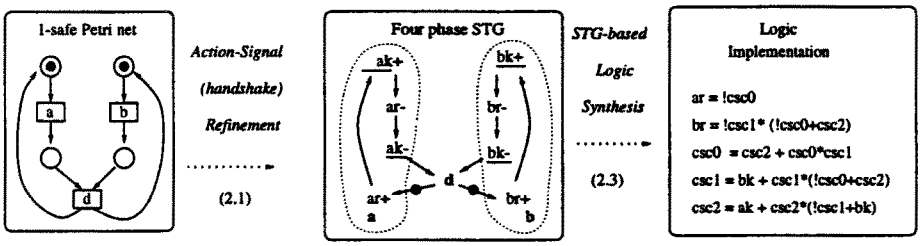


Fig. 10. Alternative four-phase STG refinement.

[6] Here, the process control semantics of the rising and falling edges of a signal is different. Only the rising edge can be significant, say, to indicate that data is valid, while the other edge simply carries out a "resetting" function.

4 Abstract Design

4.1 Modelling foundations

In this section we introduce the major modelling principles that support asynchronous design with Petri nets. As has already been noted, the main description language for behaviour design is that of labelled Petri nets. The simple processor design example (Section 2.6) shows that the designer can simply use Petri nets to capture the control flow. In other words, Petri nets appear as an immediate syntactic form for a certain semantical knowledge. However, Petri nets represent concurrency in its true form (i.e. by defining a partial order of actions), and this could be a problem for a logic designer, who is traditionally inclined to think in a sequential manner. Furthermore, the designer should be allowed to represent the behaviour partially, by defining a set of separate views which must be combined by means of a behavioural composition to produce the overall description. These views can even be regarded as "external" observations of the system's functioning that the designer wishes to realise. An often more natural way to capture the control flow behaviour is to characterise a *set of execution sequences or traces of actions* produced by the system at the interface with its environment. A convenient approach for representing trace sets in a finite form is to use characteristic predicates defined on some quantitative metrics on traces, for example the *number of occurences* of actions in traces. A Petri net model can be constructed directly from such predicates, where each predicate can be represented by a simple net fragment and composition of nets. We do not intend to consider all types of net composition [6] here, and just illustrate the above idea using the *parallel composition of nets via labelled transition synchronisation.*

Another way to capture the control flow behaviour sequentially is to use state-transition graphs (i.e. FSMs), or transition systems. Therefore, one of the important tasks in providing the designer with tools for behavioural design is to develop a method of extracting a Petri net model from a state-transition specification. We rely on the recently developed theory of regions [31, 30, 90, 84, 5]. This theory provides an important link between certain subsets of states in the transition system and places in the Petri net. It thereby makes it possible to synthesise Petri nets from state graphs [23].

We first define two modelling ingredients that will be used to discuss possible design approaches. These are Transition Systems (TSs) and Labelled Petri Nets (LPNs), as well as the notion of observational equivalence for TSs and LPNs, based on weak bisimulation. The latter seems to be sufficient for most hardware design purposes, involving both verification and synthesis aspects. We will then outline the two main approaches for constructing behavioural specifications of control logic in labelled Petri nets. The first approach is based on the synthesis of Petri nets from simple fragments based on parallel composition. The second approach is associated with the synthesis of Petri nets from Transition Systems. We illustrate our ideas with two hardware design examples, a k-place buffer and a counterflow pipeline processor control unit.

4.2 Principal Models

Transition Systems and their behavioural equivalence. A convenient way of capturing the interleaving semantics of a control behaviour in finite form is by means of a state-transition representation. This kind of model is often preferred by hardware designers.

Transition systems. A *transition system* (TS) is a quadruple $\mathrm{TS} = (S, E, T, s_{in})$, where S is a non-empty set of states, E is a set (alphabet) of events, $T \subseteq S \times E \times S$ is the transition relation, and s_{in} is the initial state.

A TS is represented by a directed graph in which every arc connecting a pair of states is labelled with a name of an event. Such a labelled arc is called a *transition* [7]. One state is marked as the initial state. We assume that any TS satisfies the following *basic conditions* [90, 23]:

A1. For every $(s, e, s') \in T, s \neq s'$, i.e., no transition may begin and end in the same state.

A2. For every $e \in E$ there are s, s' such that $(s, e, s') \in T$, i.e. every event must have some occurrence.

A3. For every $s \in S - \{s_{in}\}$ there are $(s_i, e_i, s_{i+1}) \in T$, for $i = 0, 1, \ldots, n$, such that $s_0 = s_{in}$ and $s_{n+1} = s$, i.e. every state is reachable from the initial state.

Two TS's $TS = (S_1, E_1, T_1, s_{in1})$ and $TS = (S_2, E_2, T_2, s_{in2})$ are *isomorphic* iff there exist a pair (h_S, h_E) of total bijective mappings: $h_S : S_1 \to S_2, h_E : E_1 \to E_2$ such that $(s, e, s') \in T_1 \iff (h_S(s), h_E(e), h_S(s')) \in T_2$.

We say that s' is *reachable* from s by a (possibly empty) sequence $\sigma \in E^*$ of events $e_i, i = 0, 1, \ldots, n$ if there is a (possibly empty) sequence of transitions $(s_i, e_i, s_{i+1}) \in T, s_0 = s$ and $s_{n+1} = s'$. This is denoted by $s \xrightarrow{\sigma} s'$. The sequence σ is then called *feasible* in state s. A special case of such a sequence is a *feasible event* in state s, i.e. $s \xrightarrow{e} s'$ iff $(s, e, s') \in T$.

With the notion of feasible sequences we can now easily define the interleaving semantics generated by a $TS = (S, E, T, s_{in})$ which is the set of feasible sequences, called *traces* in s_{in}; let us call it the *trace set* of TS and denote it by $\Sigma(TS)$. Note that $\Sigma(TS)$ is a prefix-closed set of sequences.

The *projection* of a sequence of events $\sigma \in E^*$ feasible in $s \in S$ on an event alphabet E' is an event sequence $\sigma' \in (E \cap E')^*$ obtained from σ by deleting all symbols which are not in E'. Let E' be an alphabet of *observable* events. We then say that a sequence of events σ' is *observably feasible* in $s \in S$ iff σ' is the projection of a sequence $\sigma \in E^*$ on E' and σ is feasible in s. This is denoted by $s \xrightarrow{\hat{\sigma'}} s'$. Again, a special case is $s \xrightarrow{\hat{e}} s'$, which means that there exists a sequence of events $\sigma \in E^*$ feasible in s and exactly one of these events is labeled with e.

[7] Note the difference between the usage of term "transition" in Transition Systems and Petri Nets.

Behavioural equivalence. Consider two TS's $TS_1 = (S_1, E_1, T_1, s_{in1})$ and $TS_2 = (S_2, E_2, T_2, s_{in2})$. A *weak bisimulation* with respect to an event alphabet E is a binary relation $\approx_E \subseteq S_1 \times S_2$ such that $(s_1, s_2) \in \approx_E$ implies, for all $e \in E$, that:

- (i) $s_1 \xrightarrow{\hat{e}} s_1'$ implies that there exist $s_2, s_2' \in S_2$, such that $s_2 \xrightarrow{\hat{e}} s_2'$ and $(s_1', s_2') \in \approx_E$, and
- (ii) $s_2 \xrightarrow{\hat{e}} s_2'$ implies that there exist $s_1, s_1' \in S_1$, such that $s_1 \xrightarrow{\hat{e}} s_1'$ and $(s_1', s_2') \in \approx_E$.

The above TS's TS_1 and TS_2 are called *observationally equivalent* (or *weakly bisimilar*) with respect to their initial states and an event alphabet E iff there is a weak bisimulation $\approx_E$ and $(s_{in1}, s_{in2}) \in \approx_E$.

This definition of bisimulation is quite similar to the original one from [77], and will be identical to it if we allow for sets $E1 \backslash E$ and $E2 \backslash E$ to consist of a single event, called the *silent action*. Such a restriction would justify transformation (1.3). For transformation (2.1), the above definition is slightly more convenient.

An example of a TS transformation which preserves observational equivalence with respect to the original set of actions $\{a, b\}$ was shown in Figure 8, step (1.3). The initial states of the TS's are both labelled with $s1$.

Labelled Petri Nets

Labelled Petri nets. The aim of abstract synthesis is to generate a labelled Petri net. We assume that the reader is familiar with the basic terminology of Petri nets [86]. We present a brief outline of the most relevant definitions.

A *Petri net* (PN) is quadruple $N = (P, E, F, m_0)$, where P is a finite set of *places*, E is finite set of *transitions* (or events), $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*, and $m_0 : P \to \{0, 1, \ldots\}$ is the *initial marking*. A PN is represented as a directed graph consisting of two types of vertices, circles for places and bars (or boxes) for transitions, and arcs, leading from circles to bars and from bars to circles, to show the flow relation. The initial marking is usually depicted by means of black dots put into places according to the number prescribed by the function m_0. Note that we allow a shorthand notation to be used for Petri nets. Two transitions can be connected by an arc directly — this arc would stand for a place with exactly one input and one output arc in a standard form. The "overloaded" arc thus becomes a carrier of tokens.

An event is *enabled* in a marking m if all its input places are marked under m. An enabled event may fire, producing a new marking (this marking is said to be *directly reachable* from the previous one) with one token less in each input place and an extra token in each output place of the transition. The set of all markings *reachable* from the initial marking is called the Reachability Set. The graph whose vertices are the markings and whose arcs correspond to the direct reachability relation is called the Reachability Graph (RG) of the net. The RG of a PN $N = (P, E, F, m_0)$ is a TS $RG(N) = (S_N, E, T_N, m_0)$, in which the set of states S_N is formed by all markings reachable from m_0, the set of events coincides with the set of events of N, the set of transitions T_N is formed by the

transitions between markings (m, e, m') whenever e can fire under $m \in S_N$, and the initial state is identified with the initial marking.

A labelled PN (LPN) is a PN in which every event $e \in E$ is labelled with a symbol from an alphabet A, thus giving rise to a *labelling function* $\lambda : E \rightarrow A$. Hence a labelled PN is a triplet $LN = (N, A, \lambda)$. In the case of *unique labelling*, i.e., if λ is bijective, each event in the net can be uniquely identified by its label. In such a case we can use the label as the event's name. In this paper, we mostly work with uniquely labelled events (with the possible exception of dummy events, unless we need to distinguish them). In addition to the TS $RG(N)$, the reachability graph of the underlying net N, a labelled PN LN produces another TS $RG(LN) = (S_N, A, T_N, m_0)$, which is graphically the same TS as $RG(N)$, but whose transitions are labelled with names from alphabet A. We call such a reachability graph the labelled reachability graph of a labelled PN LN. It is obvious that if λ is a *total unique labelling*, then $RG(N)$ and $RG(LN)$ are isomorphic.

Basic properties of Petri nets. A PN is called *k-bounded* if no more than k tokens can appear in a place. If such a finite k exists, the net is simply called *bounded*. A 1-bounded PN is also called *1-safe*, or simply *safe*. A PN is called *pure* if no (place, transition) pair is connected by mutually opposite arcs (bi-directional arcs are often used to represent such *self-loops* in PNs). A PN is called *(strongly) live* with respect to an event e if from any reachable marking m_1 it is possible to reach a marking m_2 under which e is enabled. A PN is called *live* if it is live with respect to all events. A PN is called *persistent* with respect to an event e if for any reachable marking m_1 under which this event is enabled we cannot reach another marking m_2 by firing another event e' and e is not enabled under m_2. A PN is called *persistent* if it is persistent with respect to all events.

The property of 1-safeness is crucial for the synthesis of a PN from a TS (a theory for synthesis of bounded PNs now also exists [3]) and for the conversion of a PN to a two-phase circuit. The property of liveness is not particularly critical for conversions but it helps to keep track of the effectiveness of all events and signals in the circuit, i.e., that they are not redundant in the modelled operational modes. Finally, persistency, especially persistency with respect to events modelling output signals of the circuit (see Section 5.1), is important because non-persistent events must be implemented by special arbitration elements (which contain analogue devices) to avoid hazards.

Observational equivalence and structural refinements. Two PNs N_1 and N_2 are called *observationally equivalent* with respect to a set of events E iff their RGs $RG(N_1)$ and $RG(N_2)$ are observationally equivalent with respect to E and their initial markings. Similarly, two LPNs LN_1 and LN_2 are *observationally equivalent* if their labelled RGs $RG(LN_1)$ and $RG(LN_2)$ are observationally equivalent with respect to alphabet A and their initial markings.

Examples of model transformations at the Petri net level which preserve observational equivalence with respect to the set of original events, are shown in

Figure 11. These transformations are purely syntactic, that is, they are insensitive to the semantics of the modelled system. We omit here any formal proofs of equivalence, as the reader can easily find such proofs in the literature (e.g. [86]). Note that the insertion of a dummy action $d1$ between a pair of original events a and b shown in Figure 11 (c) can sometimes be done without splitting the place p between a and b. This can be important when we want to preserve the fact that some condition, associated with such a place p, must be preserved true between the firings of a and b (see e.g. [130]).

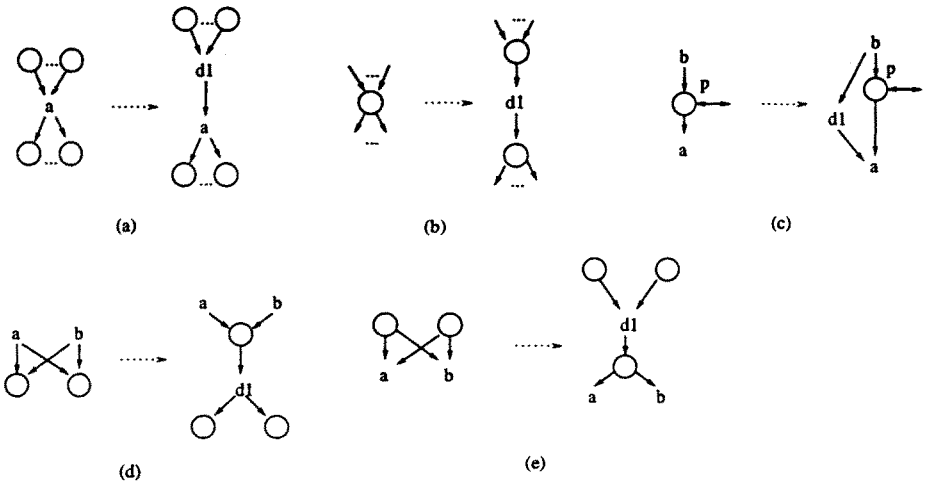


Fig. 11. Petri net structural refinements preserving observational equivalence.

PNs and LPNs inherit the trace-based interleaving semantics of their corresponding RGs seen as TSs.

Composition of labelled Petri Nets. There exist many different forms of parallel composition of Petri nets, unlabelled or labelled ones. They are sometimes called parallel composition with synchronisation on transitions [72]. In this paper we define just one specific type of LPN composition, analogous to [101], which appears to be useful for asynchronous hardware design.

For an LPN $\langle P, E, F, m_0, A, \lambda \rangle$, for any $a \in A$, $E(a) = \{e \mid e \in E : \lambda(t) = a\}$ and $F(a) = \{f \mid f \in F : (f = (e,p) \vee f = (p,e)) \wedge \lambda(e) = a\}$. They are naturally generalised to sets: $E(A) = \bigcup_{a \in A} E(a)$ and $F(A) = \bigcup_{a \in A} F(a)$. Also, $E^0 = \{e \mid e \in E : \lambda(e) = undefined\}$ and $F^0 = \{(p,e),(e,p) | \ (p,e),(e,p) \in F : \lambda(e) = undefined\}$.

Let two abstract behaviours be defined by two LPNs:
$N1 = \langle P1, E1, F1, m1_0, A1, \lambda 1 \rangle$ and $N2 = \langle P2, E2, F2, m2_0, A2, \lambda 1 \rangle$.

The *parallel composition* of these nets, denoted as $N1 \parallel N2$, is an LPN $\langle P, E, F, m_0, A, \lambda \rangle$ obtained in the following way:

1. $P = P1 \cup P2$.
2. $E = E1(A1 \setminus A2) \cup E2(A2 \setminus A1) \cup E12 \cup E1^{\circ} \cup E2^{\circ}$, where $E12 = E1(A1 \cap A2) \times E2(A1 \cap A2)$.
3. $F = F1(A1 \setminus A2) \cup F2(A2 \setminus A1) \cup F12 \cup F1^{\circ} \cup F2^{\circ}$, where $F12 = \{(e,p),(p,e) \mid e = (e1,e2) \in E12, p \in P : (p,e1) \in F1 \vee (e1,p) \in F1 \vee (p,e2) \in F2 \vee (e2,p) \in F2\}$.
4. $m_0 = m1_0 \cup m2_0$.
5. $A = A1 \cup A2$.
6. for each e:

$$\lambda(e) = \begin{cases} \lambda1(e): & e \in E1(A1 \setminus A2) \\ \lambda2(e): & e \in E2(A2 \setminus A1) \\ \lambda1(e1): & e = (e1,e2) \in E12 \\ undefined: & otherwise \end{cases}$$

Note that the most difficult part in this definition is the case of merging the transitions in $N1$ and $N2$ which have non-surjective (multiple) labelling with the same label. This requires to produce a cartesian product of such transitions, to allow for all possible means of their synchronisation. Figure 12 shows how the single transition labelled a in the first LPN needs to be "split" into two in the composition net, since it should be able to synchronise with any of the two transitions named a in the second net.

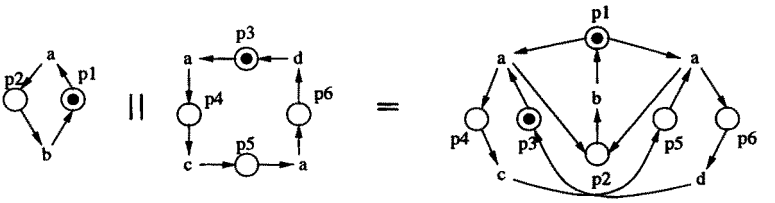


Fig. 12. Composition of labelled Petri nets.

Since we completely identify the labels with the same name in the two LPNs, it is natural to assume that the $\|$ operator is symmetric, i.e. $N1 \| N2 = N2 \| N1$, whereby in the above construction we should have $E12 = E21$ and $F12 = F21$. It is easy to prove that the $\|$ operator is associative, i.e. $(N1 \| N2) \| N3 = N1 \| (N2 \| N3)$.

Some interesting properties of the trace languages of LPN compositions can be found in [101]. Combined with the notions of synchronisation of trace structures, taken for example from [124], one can derive important links between characteristic properties of synchronised trace structures and the corresponding compositions of LPNs [133]. For example, the conjunction of characteristic predicates describing some causality conditions between events is equivalent to the parallel composition of the corresponding LPNs (cf. transformation (1.1) in Figure 8).

We can use these results to semantically justify the "forceful" imposition of idempotence onto the $\|$ operator. Thus, despite the formal definition of $\|$, we assume that $N \| N = N$. This pragmatic measure helps us to avoid unnecessary splitting of transitions if two identical nets are composed using the above construction.

Using LPNs to construct process descriptions from simpler ones offers an important advantage: the use of LPN algebra, based on the work of Mazurkiewicz [72] for unlabelled Petri nets. Mazurkiewicz algebra consists of Petri nets produced by parallel composition of Petri nets, with as zero element the empty Petri net $N0 = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ and as generator set the set of all one place nets. A Petri net $N(p, E1, E2, k) = \langle \{p\}, E1 \cup E2, (E1 \times \{p\}) \cup (E2 \times \{p\}), \{(p, k)\} \rangle$ is called a *one place net*. It contains only one place p and two sets of transitions. Its initial marking has k tokens in p ($k \geq 0$).

We extend this approach by defining the empty LPN $N0 = \langle \emptyset, \emptyset, \emptyset, 0, \emptyset, \emptyset \rangle$ and one place LPNs $N(p, E1, E2, k, A, \lambda)$ in the obvious way. The algebra of LPNs can thus be constructed using our parallel composition $\|$ operator.

We are now ready to discuss the two main approaches to abstract design.

4.3 Compositional approach to abstract design

The design of a control circuit begins with the definition of its external behaviour in abstract terms. We need to decompose this behaviour into an interconnection of simpler sub-modules, that collectively implement the same function. The behaviour of such a structural composition must be defined much more formally than can be done with hardware description languages such as VHDL or Verilog, because we do not want to rely on *simulation* to assess its correctness. Thus, we define the circuit as a *discriminator*[8] D, which can be viewed as a black box with a set of "pins" labelled by symbolic names of ports or events (e.g., read, write, strobe, acknowledge, ...) that can occur on the border between the circuit and its environment. At this level we neglect issues such as encoding of operations with signal levels or transitions on wires. Using such pins we can structurally interconnect D with other discriminators, and thus build discriminators of a higher abstraction level. An interconnection of such discriminators can communicate by performing *shared actions* on a hypothetical underlying medium.

How to construct LPN specifications for discriminators?

FIFO buffer example. A k-place buffer, denoted as $BUF_k(a, b)$, can be used to model a FIFO storage of finite capacity. Here, event a has the following meaning: 'a data item enters the FIFO through the input port', and event b means that 'a data item is retrieved from the FIFO through the output port'. Its behavior is defined by the following characteristic predicate: *for every trace t defined on*

[8] The use of this term reflects the natural ability of a control component to "discriminate" between signal transitions on its terminals during the operation, according to some prescribed strategy.

events a and b (i.e., for every arbitrary finite sequence of symbols a and b): $0 \leq (t\#a - t\#b) \leq k$, in which $t\#a$ (or simply $\#a$ when t is clear from the context) denotes the number of occurrences of symbol a in trace t. Note that, since $BUF_k(a,b)$ does not define the order in which items are retrieved from the FIFO, it would be more appropriate to use the more general term "buffer" here. In fact, this first part of the specification just ensures that we don't require too many resources from the underlying data path. The latter can be described as follows. Let $d(p_i)$ denote the i-th data value in the ordered sequence passing through the port p. The buffer realizes the FIFO discipline if $d(a_0) = d(b_0)$ and for all $i > 0 : d(a_i) = d(b_i)$ implies $d(a_{i+1}) = d(b_{i+1})$.

The predicate form of specification is convenient for logical reasoning, but may not always be intuitive to the designer. The use of a graphical capture mechanism, such as a labelled Petri net, often helps to define the internal causal relationship between the events on the discriminator boundary. This is equivalent to replacing an abstract requirement that states "every p is followed by a q" with a behavioural specification where every occurrence of p *causes*, through some physical connection, an occurrence of q. The set of traces of a given discriminator can hence be viewed as *generated* by a labeled Petri net (LPN), rather than *defined* by a predicate.

Following the above-mentioned Mazurkiewicz approach [72], LPN specifications of discriminators can be created by parallel composition of elementary fragments, describing the basic paradigms of sequential behaviour (causality, selection, ..., see Figure 13) which are common to most high-level specification formalisms (e.g., guarded commands or CSP [70]).

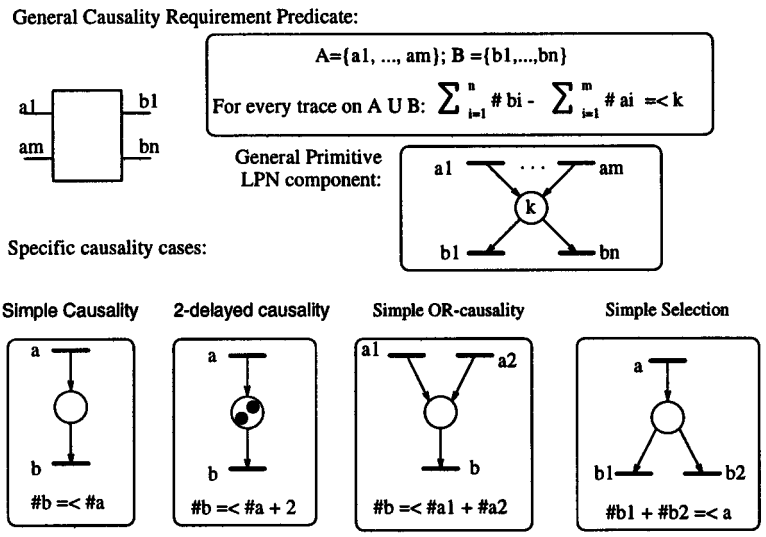


Fig. 13. Primitive LPN components.

The LPN construction process for our k-place buffer uses the fact that its characteristic predicate defined above can be composed from two primitive causality requirements joined by conjunction: $(\#b - \#a \leq 0)$ and $(\#a - \#b \leq k)$. If we apply parallel composition to the two one place LPNs corresponding to these two causality constraints joined by conjunction, we will have the LPN model for the k-place buffer, as shown in Figure 14.

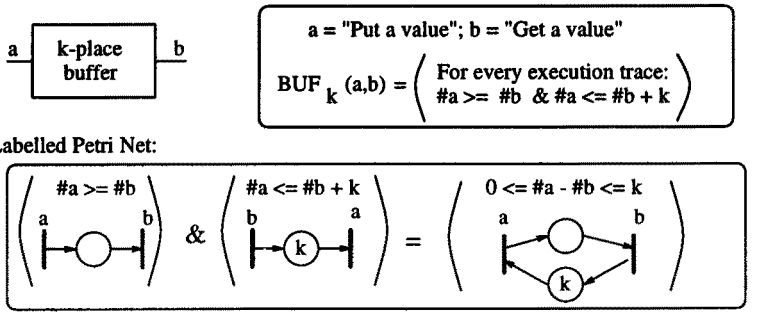


Fig. 14. Constructing an LPN for a k-place buffer.

The table shown in Figure 15 contains some typical discriminators along with their symbolic names and LPNs.

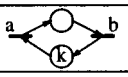
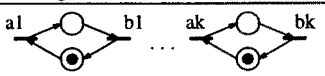
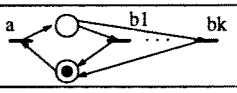
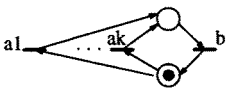
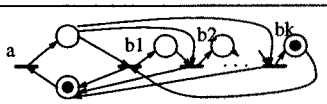
name	symbol	labelled Petri net
k-place buffer	BUF_k(a,b)	
multi-channel k-place buffer	MBUF_k(A,B)	
k-channel selector	SEL_k(a,B)	
k-channel multiplexer	MUX_k(A,b)	
ordered k-channel selector	OSEL_k(a,B)	

Fig. 15. Typical discriminators and their LPNs.

The question now arises: what if an LPN for a given discriminator is too complex to be directly synthesised into logic?

How to decompose LPN specifications for discriminators? Abstract synthesis is essentially a process of decomposing complex discriminators into simpler ones. The interconnection of elementary discriminators corresponds to the parallel composition of their LPNs. The complexity of a discriminator can be measured in terms of the number of states its LPN model generates. The reason for using such a metric, despite the fact that the LPN of a decomposed discriminator can be descriptively very simple, is that the logic synthesis stage essentially draws upon the State Graph representation. Hence this parameter is a rather crude estimate of both the final area/delay cost of the implementation, and of the complexity the logic synthesis process.

FIFO buffer decomposition. We can decompose the behavior of a FIFO buffer in (at least) two possible ways, as shown Figures 16(a) and (b). The first one is a pipeline of buffers of lower capacity, connected *in series*. Each sub-buffer must have its own storage, and therefore every data item must travel across all sub-buffers before leaving the module.

The second decomposition is a *parallel* interconnection of k buffers of capacity 1, which together correspond to a multi-channel buffer of capacity k. Two additional submodules, ordered selector and multiplexer, are used for ordering the input and output flow to and from these elementary buffers. Both organisations satisfy the FIFO discipline.

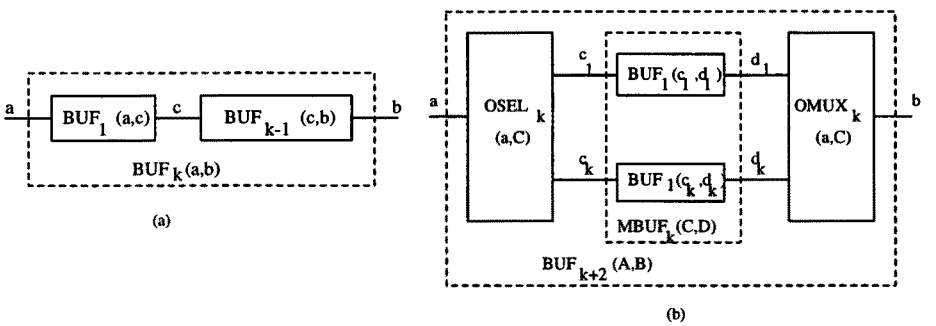


Fig. 16. Buffer decomposition: series (pipeline) and parallel.

Both these solutions have some good properties because of their regularity, and the generic structure of the buffer control circuit. The second solution has an advantage over the first in terms of speed. Although they both have the same throughput, it is easy to see that the propagation delay for one data item is proportional to the buffer length in the first case.

Although superior in speed, the second solution has a disadvantage in terms of silicon area, which can be crucial if we consider buffers with large capacity. The area of a first-cut implementation of the control circuitry for both these solutions is linear in k, but the second one has a large multiplicative factor. We

can improve the size of the control by making its largest component logarithmic in k, by using a counter as part of the buffer control circuit. The LPN models for the solution with a counter (a modulo-k Up/Down counter), which also uses a dual-port memory and an arbiter, can be found in [134].

An LPN description of a discriminator corresponding to a modulo-k Up/Down counter is shown in Figure 17(a). The LPN model of the environment for such a counter is shown in Figure 17(b). Note that the LPN for a modulo-k counter uses $(k-1)$-bounded places as well as $(k-1)$-weighted arcs, corresponding to a flow relation in which $k-1$ tokens are involved in the enabling and firing conditions. In other words, if a $(k-1)$-weighted arc connects a place with a transition, the latter is enabled only when the former has at least $k-1$ tokens, and when the transition is fired it removes $k-1$ tokens from such a place. Thus, the model of a modulo-k counter reflects the fact that such a counter counts up (action *inc*) from the zero state for $k-1$ UP commands arriving in excess of DOWN commands, and then resets back to zero when the counter is FULL. It is easy to to see that in the latter case the counter produces an event (action *inc'*) which can be used to generate a carry signal to the next stage if the counter was part of a cascaded counting circuit. Similar behaviour is produced for the DOWN commands (with actions *dec* and *dec'*, respectively).

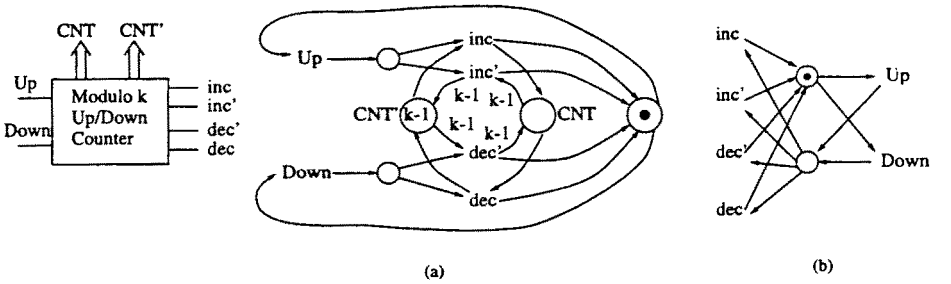


Fig. 17. (a) Labelled Petri net of a modulo-k Up/Down counter; (b) LPN model of the environment.

Hierarchical decomposition of a modulo-k counter into modulo-2 and modulo-$k/2$ counters is shown in Figure 18. It depicts corresponding structural images of the discriminators for such counters and, additionally, a pair of two-input multuplexers. The interconnections between the labelled pins correspond to the synchronised transitions in their LPNs.

Verification of LPN *compositions.* The process of decomposing the initial LPN specification into a set of LPNs for simpler discriminators does not guarantee correctness by construction. Such a compositional design must be verified [133]. There exist a number of techniques and algorithms for the verification of conformance between a specification and an implementation, most of them defined

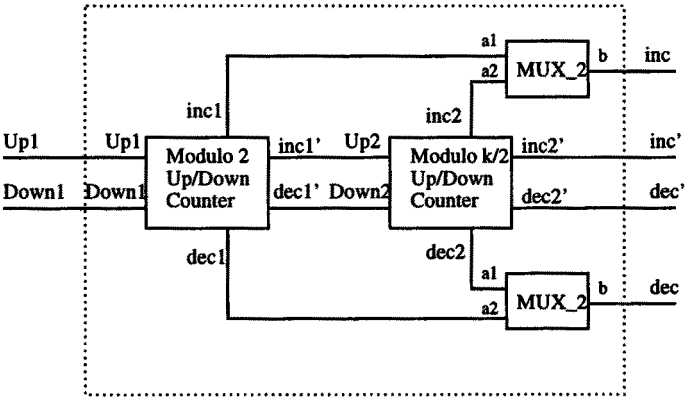


Fig. 18. Hierarchical decomposition of a modulo-k counter.

within the general framework of a labelled transition system, and hence applicable to Petri nets. These techniques are not particularly specific to hardware design. Some have been applied to Petri net models of communication protocols or asynchonous circuits, and they work for different equivalence notions such as trace equivalence [28, 74] and bisimulation [33]. It is also worth mentioning the applicability of bisimulation concepts to Petri nets [91].

4.4 Synthesis of Labelled Petri nets from Transition Systems

Our approach to abstract design is based on the synthesis of an LPN model of a control circuit from its original description in the form of a Transition System. The theory of regions [31, 30, 27, 84, 5] provides ideas for this kind of synthesis. The implementation and applications of this theory have been presented in [23]. Petrify (see Section 6.3) performs such synthesis automatically. Here we outline important aspects of this approach, and illustrate it with a simple yet realistic application.

Counterflow pipeline control example. Let us consider the problem of synthesising an asynchronous control circuit for a stage of the Sproull Counterflow pipeline processor (CFPP) [116]. This problem was proposed as an exercise in applying formal techniques to asynchronous circuit design [81].

For a complete description of the CFPP architecture we refer to [116]. Here, we would like to ignore the details of instruction execution, and concentrate on the issue of the behavioural specification of control in a basic stage of the CFPP.

The overall organisation of control in a CFPP is as follows. There are two mutually synchronised pipelines, one for instructions and one for results, where the results are used by instructions and may be produced or updated by them. These pipelines allow instructions and results to propagate in opposite directions. Each operates as an ordinary pipeline with data items passing between any pair of adjacent stages if one of the stages is empty and the preceding stages holds

a data item. The role of data items is played by instructions, in the instruction pipe, and by results, in the result pipe.

Figure 19 shows the state diagram of a pipeline stage control proposed by Charles Molnar. The states have the following meaning:

E: **Empty.** Neither instruction nor result is present.
I: **Instruction.** Only an instruction is present.
R: **Result.** Only a result is present.
F: **Full.** Both instruction and result have arrived.
C: **Complete.** The CFPP execution rules [116] have been enforced, and both instruction and result are free to move on [9].

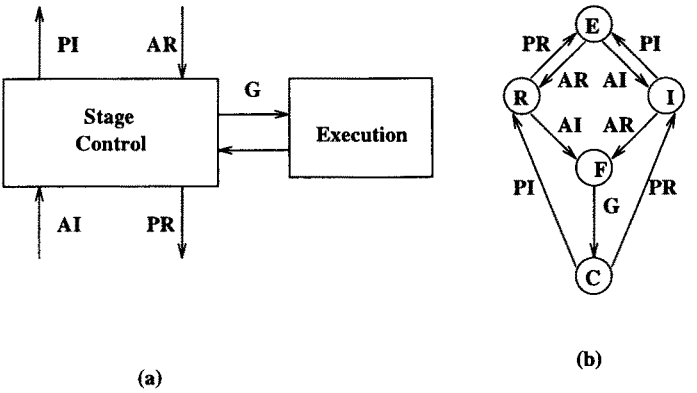


Fig. 19. Counterflow pipeline stage control: (a) structural view, (b) state diagram.

The transitions that involve motion of instructions and results are labelled *AI* (accept instruction from below), *PI* (pass instruction upward), *AR* (accept result from above), *PR* (pass result downward), and *G* (perform *garnering*, which means either executing the instruction if its operand matches the result or releasing both instruction and result).

We note that in the state graph two states are present in which dynamic arbitration may take place. The first one is state *I*, where *either* an instruction may be passed before the result arrives in the stage *or* a result may arrive before an instruction is allowed to leave the stage. Similarly, in state *R*, *either* a result may be passed before an instruction arrives in the stage *or* an instruction may arrive before a result is allowed to leave the stage.

The major challenge in designing an asynchronous circuit for Molnar's transition system is to transform the latter into a formal model amenable to sub-

[9] As was noted in [116], in practice this state might be divided further to allow the result to advance while the instruction is being executed. We, however, abstract away from such distinctions in this paper.

sequent circuit synthesis. We would therefore like to obtain a labelled Petri net that would be observationally equivalent to Molnar's TS.

From Transition Systems to Petri Nets The basic intuitive idea behind the construction of a Petri net whose behaviour is equivalent [10] to the original TS is a correspondence between regions and places in the synthesised net. This allows a 1-1 correspondence between states of a region and markings of the Petri net in which the place corresponding to the region has a token.

More specifically, a region is a subset of states with which *all* transitions labelled with the same event e have exactly the same "entry/exit" relationship. We say that a subset of states r is *entered* by event e if for every transition labelled with e the source state does not belong to r while the destination state is in r. Similarly, r is *exited* by e if for every e-labelled transition the source state is in R while the destination is outside r. In the remaining cases, e is said to be *non-crossing*, either *internal* or *external*, event for a region. Thus, to become a region, a subset r must satisfy exactly one of the three cases for every event e: (i) r is entered by e, (ii) r is exited by e, and (iii) r is not crossed by e.

A region r is a *pre-region* (*post-region*, *co-region*) of an event e if r is exited by (entered by, internal for) e.

Figure 20 illustrates a pair of regions, $r1 = \{E, R\}$ and $r2 = \{I, F, C\}$, in the TS of the CFPP stage control. Note that $r1$ is a pre-region for event AI and a post-region for PI whereas $r2$ is a pre-region for PI, post-region for AI and a co-region for G. Both regions are not crossed by AR and PR. Finally, G is an external event for $r1$ and internal for $r2$.

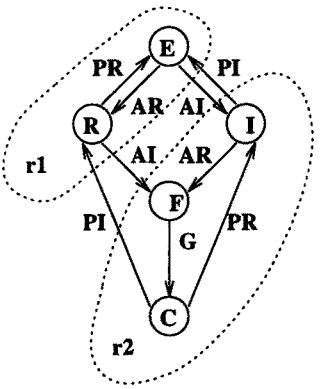


Fig. 20. Illustration of regions.

It is known from [90] that in order to generate an elementary net whose

[10] We use a strong notion of equivalence, isomorphism, between the given transition system and the transition system which is obtained from the reachability graph of the Petri net.

reachability graph is isomorphic to a given TS, the TS must be *elementary*. Elementary nets are effectively a subclass of 1-safe Petri nets [11]. The "gap" between 1-safe nets and elementary nets is filled by *non-pure* nets, which allow a self-loop relation between places and transitions. Such nets and their TS's, called *semi-elementary*, have been studied in[99]. In this design exercise, we need to be able to deal with non-pure nets.

The *semi-elementarity conditions*, additional to conditions A1 – A3, discussed in Section 4.2, are as follows:

A4. The state separation property, which requires that for any two different states there must exist a region which contains one of the states and does not include the other.

A5. The forward closure property, which requires that, for every state s and every event e, if the sets of pre-regions and co-regions of e are included in the set of regions such that each of them contains s, then e must be enabled in s (i.e., there must be a transition from s labelled with e).

Following [99], for any semi-elementary TS TS there exists a 1-safe PN N such that: (1) each event in N is *uniquely* labelled with an event of TS; (2) the $RG(N)$ is isomorphic to TS.

The basic procedure to produce a PN from a semi-elementary TS is as follows:

1. For each event e an event labelled with e is built in the PN;
2. For each region r a place named r is generated;
3. Place r contains a token in the initial marking iff the corresponding region r contains the initial state of the TS;
4. The flow relation is built according to the relationship between pre-/co-regions and events, and between events and post-/co-regions.

A PN synthesised by this procedure is called a *saturated* net, since all regions are mapped into corresponding places. A saturated net may have a lot of redundancy, i.e. some of its places may be removed without disturbing the isomorphism of reachability graphs. As shown in [5], it is sufficient to consider only regions which are not sub-regions of other regions (such regions are called *minimal*). The net constructed from all minimal regions is called a *minimal saturated* net. The method described in [23] and implemented in the Petrify tool [22] performs additional optimisation, and produces an irredundant net with minimal regions (the idea is somewhat similar to an irredundant cover of prime implicants in logic minimisation [8]). The semi-elementarity condition sometimes requires the use of non-minimal regions as co-regions. Since enumeration of non-minimal regions is computationally hard, Petrify applies some heuristics to optimise its search for co-region candidates. Furthermore, Petrify uses a modified version of the semi-elementarity condition, called "Excitation Closure", based on excitation

[11] We say "effectively" because formally elementary nets are defined in a slightly different way (see, e.g., [90]), but any elementary net can be converted into a behaviourally equivalent 1-safe net, by possibly adding complementary places [23].

regions [23], which allows a more efficient checking procedure using a symbolic state representation framework.

A set of states is a *generalised excitation region* for event e, denoted by $GER(e)$, if it is a maximal set of states such that in every element of this set event e is enabled. Excitation Closure requires that for every event e the intersection of pre-regions and co-regions of e is equal to $GER(e)$.

In the TS of Figure 20, the semi-elementarity property does not hold for several events. For example, $GER(PI) = \{I, C\}$, but the only pre-region of PI is region $r2 = \{I, F, C\}$; $GER(G) = \{F\}$ but the set of pre-regions of G is empty. This TS is therefore not semi-elementary.

Synthesis of a Petri net model for CFPP stage control. Let us now revisit the original TS model of the CFPP stage control and transform it to a TS that would generate a PN using the above technique. The main obstacle to satisfying the semi-elementarity (Excitation Closure) condition comes with event G, for which we do not have appropriate pre-regions and post-regions. We need to insert auxiliary (dummy) events into the original TS in such a way that the resulting TS is semi-elementary and bisimilar to the original TS. This would correspond to stage (1.1) in our classification of Section 3.1.

Intuitively, and this is one of the heuristics of the dummy insertion method, we need to establish proper "diamond" structures in the TS, reflecting the potential concurrency between pairs (AI, AR) and (PI, PR).

The solution is shown in Figure 21(b), where states I and R are shared between the diamonds and the only dummy event is labelled with d. This dummy plays the same role for the (PI, PR) diamond as G for the (AI, AR) pair.

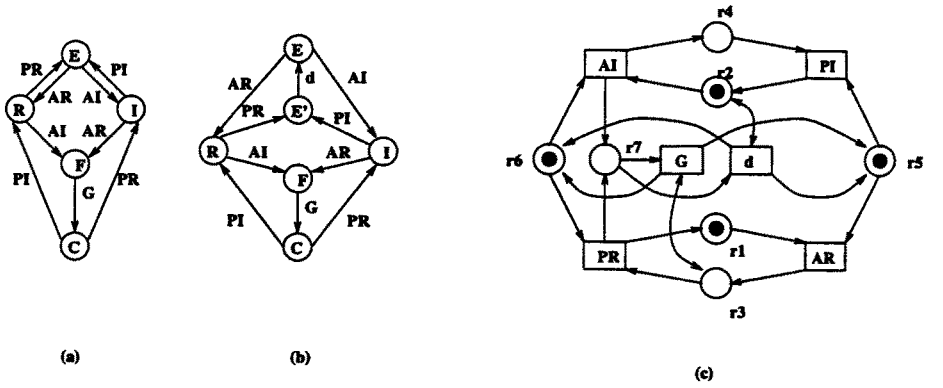


Fig. 21. Converting Molnar's model to a Petri net: (a) original TS, (b) transformed TS, (c) 1-safe LPN.

The TS is not elementary in its basic form [90] (without use of co-regions) but it is semi-elementary since it satisfies the Excitation Closure condition for

pre-regions and co-regions. We can therefore proceed to stage (1.4) in our transformation process and apply the procedure described above. This produces the net shown in Figure 21 (c). The regions that generate the places of this net are as follows: $r1 = \{E, I, E'\}, r2 = \{E, R, E'\}, r3 = \{R, F, C\}, r4 = \{I, F, C\}, r5 = \{E, I, C\}, r6 = \{E, R, C\}$ and $r7 = \{I, E', F\}$. Note that (r_1, r_3), (r_2, r_4) and $(r6, r7)$ are pairs of complementary regions (for any such pair, the TS is always either in the first or second region). The reader may check the pre-regions and co-regions for all events by tracing them back from the arcs. Due to the presence of co-regions to events, and hence self-loop arcs, the resulting net is non-pure. The complete description of the circuit design process from this Petri net can be found in [130].

5 Logic Design

Let us now examine the second design stage, Logic Design, and the role of Petri net models in producing logic implementations of the behavioural specifications obtained from the Abstract Design, presented in the previous section. The key formalism at the logic design stage is the Signal Transition Graph.

5.1 Models for synthesis of circuits at the logic level

Signal Transition Graph. A Signal Transition Graph (STG) is a special case of a Labelled Petri Net model, and is used to describe *signalling expansion* of LPNs produced by abstract design. Sometimes, however, an STG description can be built directly, say, from a timing diagram, if the design process starts immediately at the binary signal level (which is often the case). The major advantage of using STGs at the logic design stage is that this model proves to be very efficient in defining causality and parallelism at the binary signal level. If STG models are build from the original informal descriptions, such as timing diagrams for bus protocols, it is possible to use the same techniques as those described in the previous section, i.e. the compositional approach or synthesis from Transition Systems. For example, it was shown in [135] that an STG for a bus interface adapter (one of our initial examples in Section 2.6), can be built as a parallel composition of STGs built for separate "partial" behavioural views (called "snippets" by R. Sproull) of the designed circuits.

STGs, either derived from LPNs or built from the original informal descriptions, can be analysed by the same methods and tools as LPNs. This avoids the problem of looking for an intermediate formal notation to prove the semantic relationship between the abstract and logic design models. There are several techniques and tools for synthesis of asynchronous circuits from STGs (see Section 6).

It should be stressed that STGs represent a narrower class of processes than those that can be defined by LPNs. But the narrowness only concerns their alphabet of transition labels. We do not impose any restriction upon the structure of the underlying Petri nets, so that the causality paradigms achievable at the

abstract level can be preserved at the signal level. Furthermore, since it is possible to insert auxiliary signal transitions, we can transform the design by changing its STG.

Formally, a *Signal Transition Graph* is a triplet $G = (N, Y, \lambda)$, where $N = (P, E, F, m_0)$ is a PN, Y is a nonempty set of binary signals, and $\lambda : E \rightarrow Y \times \{+, -, \sim\}$, where $y+(y-)$ stands for the rising (respectively, falling) edge of signal y (e.g., in the four-phase signalling), while $y\sim$ means either rising or falling edge of y (e.g., in the two-phase signalling). In other words, $y\sim$ means a transition of signal y regardless of its current state. Thus an STG is a PN whose events are labelled with the names of binary signal transitions. Note that the labelling function λ for STGs can generally be partial, that is, some transitions in the underlying PN may not be labelled by signal transitions. These are again *dummy* or *silent* transitions. They are often used to simplify the representation of the behaviour described by an STG. They help to avoid cluttering when transitions are labelled with the same signal name when merging alternative branches on a place.

Because it is a special case of an LPN, an STG generates an RG whose events are labelled with signal transitions. Two STGs G_1 and G_2 are *observationally equivalent* if their labelled RGs $RG(G_1)$ and $RG(G_2)$ are observationally equivalent with respect to a signal set Y and their initial markings. For example, for an RG RG_1 produced by an STG G_1 with dummy events, one can obtain an observationally equivalent TS TS without dummy events. Then the TS can be converted into another STG G_2 using a synthesis technique similar to the one described in Section 4.4, though possibly with split events [23]. Such a G_2 will be observationally equivalent to G_1. Similarly, if an originally defined STG G_1 is augmented with new signal transitions or dummy events, then the structural transformations at the Petri net level shown in Figure 11 always preserve observational equivalence.

State Graph: consistent state coding and output-persistence. With respect to its underlying PN, any STG inherits all properties of PNs, such as boundedness, liveness, persistency etc. There are, however, additional properties introduced by the signal or circuit interpretation of STG events and states. These properties are crucial for the synthesis of the logic implementation from an STG specification. They are defined through the notion of a State Graph.

Any STG $G = (N, Y, \lambda)$, where $N = (P, E, F, m_0)$ is an underlying PN, can be defined with an explicit initial binary state, a vector $v_0 : Y \rightarrow \{0, 1\}^n, n = |Y|$, associated with the initial marking m_0. A feasible sequence σ of such a G is called *valid* iff for every signal $y \in Y$: (i) the next possible edge of signal y after $y+(y-)$ can only be $y-(y+)$, and (ii) the first change of signal y is consistent with the initial state v_0, i.e.: if the value of y is 0(1) in v_0, then only $y+(y-)$ can first appear in σ. An STG is *valid* iff every firing sequence generated from m_0 is valid.

It has been shown [104, 19] that any valid STG has a *consistent binary state encoding* for all of its reachable markings in set S_N. In fact, the actual validity check can be performed through analysis of consistency of the state encoding v :

$S_N \rightarrow \{0,1\}^n$ of the reachability graph. Namely, such an encoding is *consistent* iff for each edge $m' \rightarrow m''$ in the reachability graph, labelled with signal y:

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

Note that the above notion of STG validity and state encoding consistency is crucial only for STGs in which there is at least one signal (e.g., y) with a four-phase signalling (i.e. the corresponding events on y are not of the toggle type $y\sim$). For STGs with toggle events, an STG is valid if for any reachable marking m no two or more transitions labelled with the same signal name y are concurrently enabled in 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* (SG) of the STG.

An efficient method for checking STG validity using the symbolic traversal of the RG has been proposed in [59]. Other techniques, that avoid state reachability analysis, use a partial order approach [62, 108, 64, 61].

Thus, the property of validity is a necessary and sufficient condition for a bounded STG to produce a consistently encoded finite state graph. This is important for the logic design of a circuit based on logic synthesis. It is, however, only a partial condition for logic implementability of the behavioural specification.

Another important property for logic synthesis is that of persistence of signal transitions in the specification. This property is defined in two versions, one at the STG level and the other at the SG level.

An STG is called *output-persistent* if its underlying PN is persistent with respect to transitions labelled with non-input signals. This property is ensured by means of an STG transformation called the mutual exclusion event insertion. This is explained below.

An SG is called *output-persistent* if for each non-input signal y and for any reachable marking $m \in S_G$ such that m enables a transition t, labelled with $y*$ (where $y* \in Y \times \{+,-,\sim\}$), any other marking m' reachable from m by firing any other transition t' also enables some transition labelled with $y*$.

The difference between STG output-persistence and SG output-persistence lies in the possibility of an STG to be non-output-persistent and yet produce an SG that is output-persistent, as shown in Figure 22. Here we have STG $G1$ and $G2$ with a and b being non-input signals. These STGs are observationally equivalent, and they produce isomorphic SGs. In fact we have depicted only one of the SGs, showing only the binary vectors and hiding the place markings. This SG is ouput-persistent. Likewise STG $G2$, whereas $G1$ is not output-persistent because transitions incident to place $p1$ disable each other. The property of output-persistence has been studied at length in connection with the logic implementability of STGs in [137, 59].

At this point we should distinguish between the two major threads of circuit

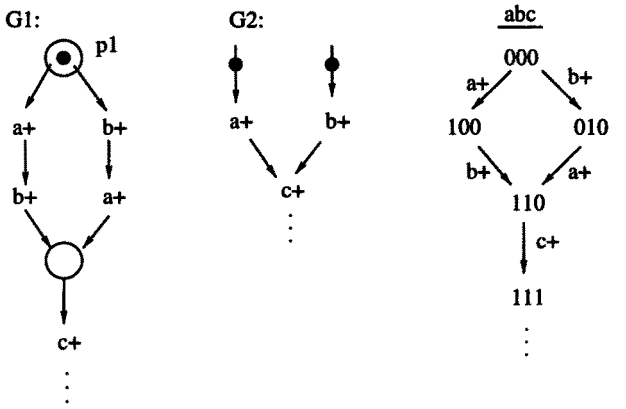


Fig. 22. Difference between notions of output-persistence at the Signal Transition Graph level and at the State Graph level.

design from Signal Transition Graphs, according to the transformations (2.2) and (2.3) initially outlined in Section 3.1 .

Before we examine these threads in more detail, let us first detail the notion of a signalling expansion of an LPN specification into an STG (transformation (2.1) in Figure 9).

5.2 Signalling expansion of labelled Petri nets

We have already mentioned certain specification refinements which can be applied purely at the syntactic level, either for abstract LPNs or for STGs. Two other types of refinements with "semantic flavour" are those of *handshake expansion*, and *semaphore insertion* or *mutual exclusion insertion*. They are discussed below.

Handshake expansion. The two main handshake refinements are shown in Figure 23. These are *passive* and *active* handshakes [123]. Each can be applied in one of two signalling protocols: (i) *two-phase* (also called Non-Return-to-Zero (NRZ)) protocol, and (ii) *four-phase* (Return-to-Zero (RZ)) protocol. Consider for instance the two-phase case. An abstract action a on a port with a request-acknowledgement handshake (ar, ak) is typically refined with two events $ar\sim$ and $ak\sim$. For a passive (active) handshake the request $ar\sim$ is an input (output) event, and the acknowledgement $ak\sim$ is an output (input) event. Input events are underlined in the figure. We assume that whenever the environment is willing to execute action a on a passive (active) handshake the circuit must be ready, and vice versa. This is reflected in the synchronisations shown in the figure. Note that, if we semantically identify the critical event $ak\sim$ with the original action a (indeed, the acknowledgement $ak\sim$ actually determines that action a has been completed), then it should be clear that for a 1-safe Petri net, the net obtained

after such refinements is observationally equivalent due to the rules of the PN transformations shown in Figure 11. Similarly, a four-phase protocol can be used to refine an abstract event a for either a passive or an active handshake.

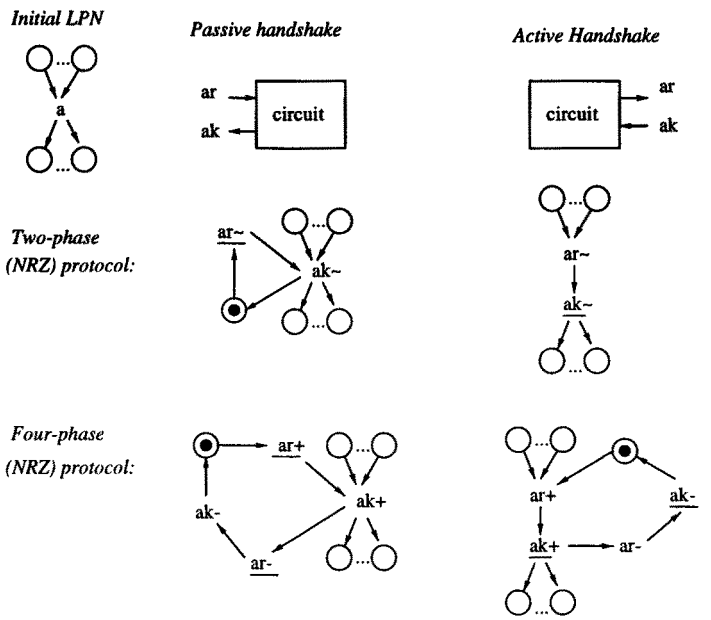


Fig. 23. Passive and active handshake refinement of an abstract action.

A simple example of the use of these signalling refinements is shown in Figure 24, where action a ("receive a data item from the environment") is refined as a passive handshake while action b ("send a data item to the environment") as an active handshake. Note that a handshake expansion is essentially design-dependent, and there is no strict rule on how precisely two or more handshake events must be synchronised on the non-critical events. For example, in our one-place buffer example, we could "reshuffle" the non-critical transitions of both handshakes in a number of ways. The question which particular "reshuffling" is better depends on a tradeoff between the speed of the circuit, for example, by achieving maximum concurrency between the most time-consuming events, and its complexity (more concurrent specifications usually require additional state signals to resolve state encoding conflicts). An example of a more constrained four-phase synchronisation between the a and b links in the one-place buffer is shown in Figure 25. Here, the handshakes are also synchronised in the release phase, which is made "symmetric" to the asserting phase, which leads to simpler logic (just one C-element for the ak signal, the br signal is a "delayed" copy of ak) than the one that would have been produced for the more concurrent option, which requires adding a state signal to resolve the so-called Complete State Coding problem (explained in Section 5.4).

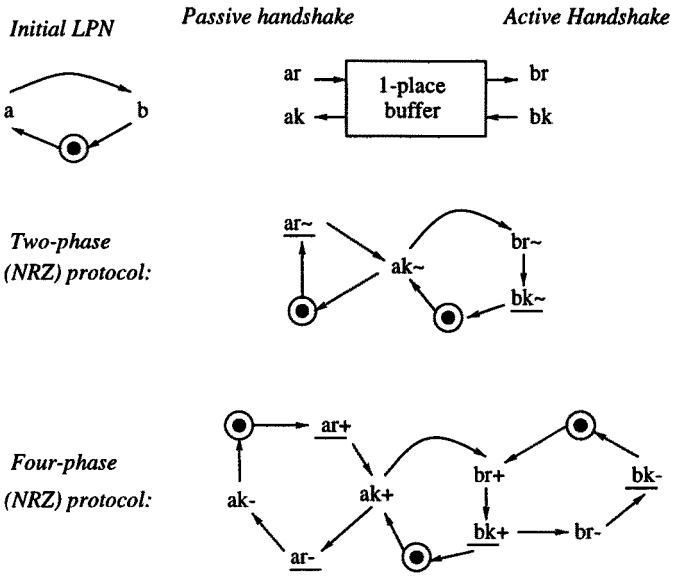


Fig. 24. Passive and active handshake refinement of a one-place buffer LPN.

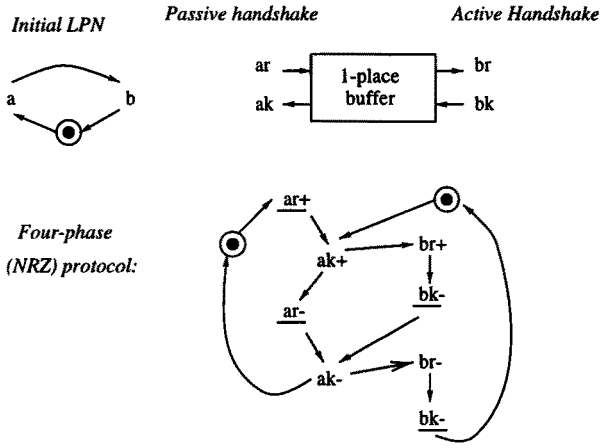


Fig. 25. Illustration of reshuffling.

Mutual exclusion insertion. Asynchronous circuit design sometimes requires the construction of control circuits with internal conflict resolution. Conflicts may

exhibit themselves either as part of the algorithm or circuit specification, or as a result of events being disabled by other events. Examples of the former situation are designs of asynchronous arbiters [138]. The latter case is less intentional — the specification of a circuit may involve operation of a signal in such a way that it is controlled and tested by independent, potentially concurrent processes; e.g. interrupt handling logic, or devices where the environment may not always fulfill the prescribed timing constraints [25].

In both situations the corresponding LPN or STG specification is not persistent — a net transition enabled in one marking may be disabled by the firing of another transition. In fact, it may be non-persistent with respect to transitions which are labelled with signals produced by the environment. Such a case is not critical for the design since it is concerned with external nondeterministic behaviour. However, if a particular non-persistent transition is associated with a non-input signal, then the latter cannot be implemented without a hazard unless its transition is regarded as a *critical section* in the specification, and suitably protected in the circuit. A method for protecting signals whose initial specification is non-persistent has been described in detail in [25] and is based on:

- the insertion of special auxiliary *mutex* transitions which are similar to semaphore actions in concurrent programs at the Petri net level in such a way that nonpersistence with respect to the original noninput signal is eliminated;
- synthesising all the original non-input signals in the usual way and considering these auxiliary mutex signals as inputs;
- implementing the auxiliary signals on special mutex or arbiter circuit components which are analogue circuits; and
- connecting the outputs of the mutex elements with the logical part of the circuit.

An example of mutual exclusion action insertion is shown in Figure 26. This refinement allows for the use of a particular type of arbiter, a so-called RGD arbiter [118] with a single "Done" signal [116]. Again, this refinement is well backed up by the PN transformations of Figure 11, and preserves observational equivalence with respect to actions a and b (and, of course, the rest of the surrounding net's actions). It can be observed that the original nonpersistence between events a and b, which are considered to be critical sections, has been shifted to a "programmed" conflict between the grant signals of the RGD arbiter, G_a and G_b. Examples of mutual exclusion event insertion can be found in [130, 14, 67].

Let us now proceed to discussing methods for circuit implementation of logic-level Petri net specifications.

5.3 Syntax-directed circuit synthesis

This approach starts with a labelled Petri net specification, refined at the level of signal transitions, e.g. by a Signal Transition Graph, and constructs a circuit

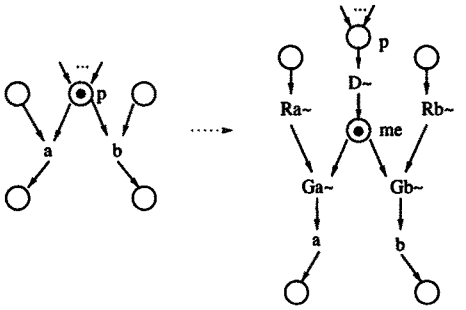


Fig. 26. Refinement with explicit arbiter signals.

which effectively "simulates" the net, by replacing its components with circuit elements. The net *must be 1-safe* in order to make sure that the circuit produced is meaningful, since we consider here only the binary meaning of the token flow[12]. Within this approach two alternative threads can be identified. They are as follows.

"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. Examples of this style are described in [43, 127, 128, 129]. Several types of flip-flops reflecting the AND-causal logic of transitions are shown in Figure 27. 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 27(a), is manifested in the circuit by setting the corresponding SR-flip-flop (whose outputs are labelled $p1$ 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 $p1 = 0, p1' = 1$.

The arrival of a token in place $p3$ in Figure 27(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 27(c) back to the state $p1 = 0, p1' = 1$, which corresponds 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 $(p3, p3')$

[12] It is of course possible to generalise the notion of direct simulation, and thus relax the 1-safeness to boundedness, for example by "embedding" a certain element of datapath into the main control flow design (e.g., compiling k-bounded places into Up-Down counters rather than flip-flips). For the sake of simplicity we do not consider such extensions here.

have been set to states in which $p2 = 1, p2' = 0$ and $p3 = 1, p3' = 0$. The sum-of-product logic of the gate that implements $p1'$ 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. An analysis of these semantical aspects may be found in [120].

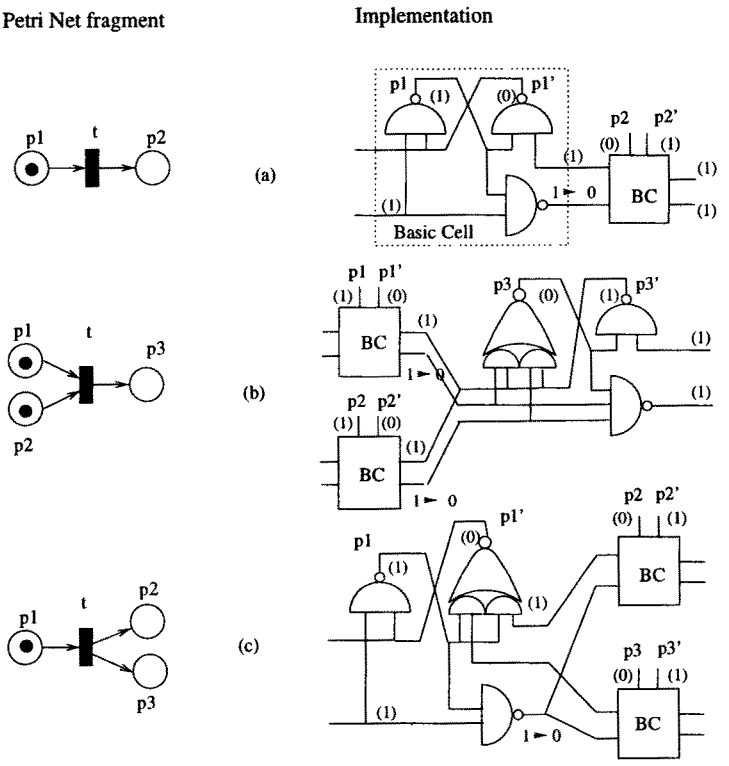


Fig. 27. Syntax-directed translation from Petri nets based on "place-latch" relationship.

For a practical example of the use of this translation style the reader is referred to [136]. 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 main idea of this translation is captured in Figure 28. Here the transitions in the initial Petri net model (part (a)) are interpreted as operations to be activated by the designed control circuit, and places are associated with latch-based cells. The operations may involve actions on a datapath, e.g. copying a value from a port into a register, or simply control of binary latches, e.g. setting a latch to logical 1. In the implementation, these blocks are then inserted between the request and acknowledge signals in the resulting control logic. For example, Operation $Op1$ is started by signal $a1$ switching from 1 to 0 and acknowledged by signal $b1$ also going from 1 to 0. The phase of control in which $a1$ and $b1$ are reset to logical 1 follows the activation phase immediately, that is, prior to the activation of the next operation. This can be seen from the ordering of signal transitions in the STG in Figure 28(b).

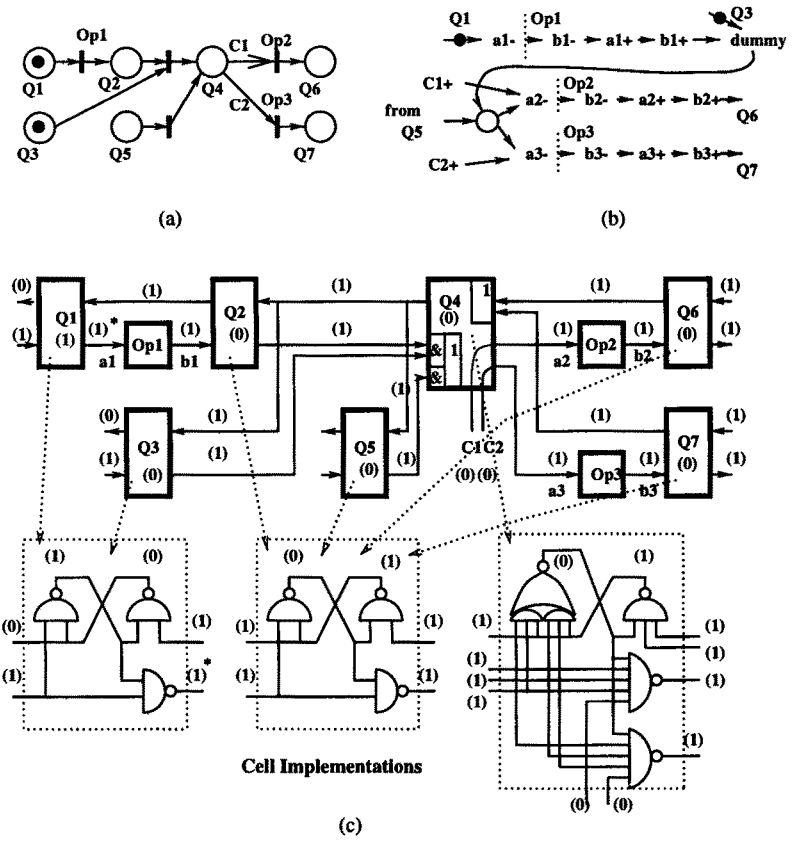


Fig. 28. "Circuit compilation" of a Petri net specification: fragment of net specification (a); Signal Transition Graph description of the operation synchronisation (b); control circuit implementation based on the idea of "place-latch" relationship.

This method proved to be highly effective for the design of a circuit to control around 50 different request-acknowledgement handshakes, described in [136]. Other techniques, e.g. those based on Finite State Machines or Signal Transition Graphs, would have faced with the computationally hard problems of state assignment and hazard-free implementation. These techniques could 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.

Event-based, two-phase circuit compilation. Another way to directly "simulate" the control flow in Petri nets is based on the correspondence between the firing of a transition in a net and the switching of a logical level of a circuit signal. The firing of a transition has purely an event-based meaning, and it does not imply the direction of a transition between logical levels. This technique is therefore more appropriate for specifications defined as STGs with "toggle" transitions ($y\sim$). For such models, there is no difference between the rising and falling edges of the control signals. This translation strategy originates from [97]. Figure 29 shows Patil's "mapping" of primitive fragments of Petri nets into event-based circuits.

The two most important elements are as follows. A C-element implements AND-causality between a set of predecessor events and a given event; it can also be viewed as the so-called JOIN-function between subprocesses. An XOR logic gate implements (exclusive) OR-causality between a group of mutually exclusive predecessor actions and a given action; this functionality is often called MERGE. In order to implement net fragments with a structural conflict, i.e. where a single predecessor place can be shared between several transitions, one has to use a Switch element. This element has an internal arbitration component. Therefore specifications with conflicts on output signals (cf. non-output-persistent STGs) can be implemented.

Patil's mapping is applicable to a structural subclass of nets known as *Simple Nets* (recall that nets are also assumed to be 1-safe). 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 30 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 30. We should also bear in mind that this method, which converts net transitions into control signals, is effectively applied in its original form only to STGs with injective labelling. In order to be able to work with multiple labelling we need some form of multiplexing, which requires use of a special auxiliary component, e.g. a Call-element (see below).

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

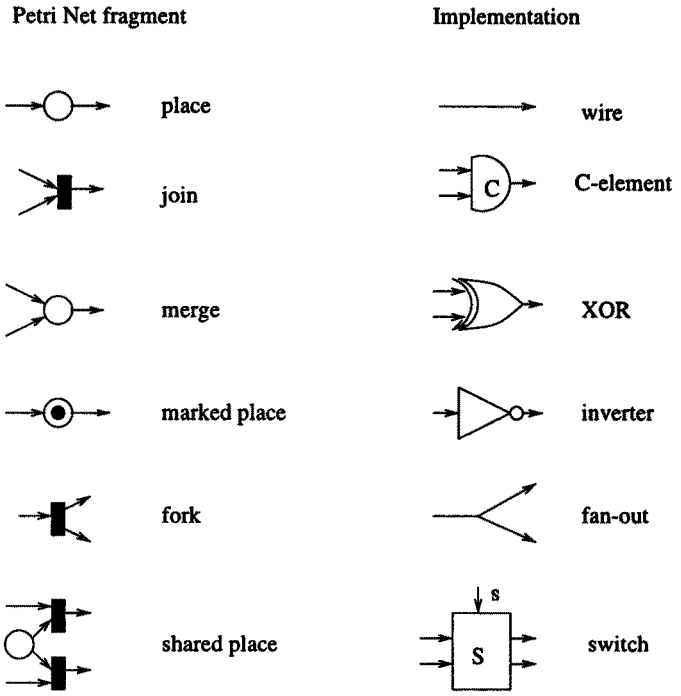


Fig. 29. Syntax-directed translation based on the "transition-signal event" relationship.

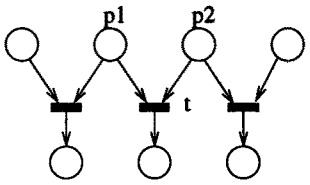


Fig. 30. A fragment of a net which is not Simple Net.

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 31,(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 $p2$

for the second time (this means that input y becomes 0 again), it may either again assist in enabling $t1$, 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 $z2$.

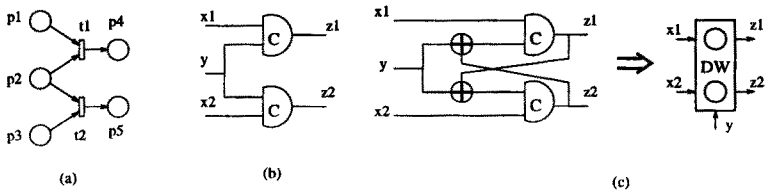


Fig. 31. Problems with C-elements; need for Decision-Wait element.

The above problems can be verified 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 we can use C-gates (i.e. $t1$ must fire an even number of times before $t2$ is enabled, and vice versa). 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, i.e. purely on an event basis. Figure 31(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 [47]. Another useful application of a DW element is illustrated in Figure 32, 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.

It should be pointed out that Furtek [34] has presented a set of circuit constructs associated with four main types of Petri net arcs, which effectively proves that *any safe net* can be directly implemented into a circuit with equivalent behaviour – the circuit will use Patil's set of modules augmented with an element similar to the DW shown in Figure 32. However, these constructs are

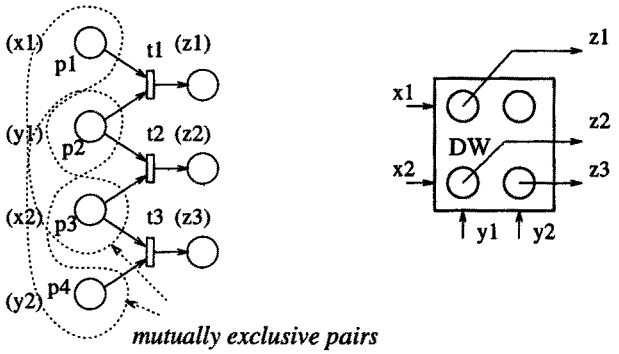


Fig. 32. Use of Decision-Wait elements for non-simple nets.

prohibitively complex and are mostly of theoretical interest.

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 semantic correspondence between the net and the circuit. Some of these transformations may optimise the overall design by recognising complex fragments of Petri nets and compiling them directly into larger circuit elements, such as Select, Call and Toggle. Their LPN models and possible implementations are shown in Figures 33, 34 and 35, respectively.

Select is a module which allows us to implement parts of LPN specifications which model the interfaces between event-based control signals and a datapath or level-based logic. Such nets use *self-loop* arcs (also called *read* arcs, or *positive contextual* arcs [82]). A self-loop arc between a place and a transition syntactically means that the place is both input and output for the transition. This arc has the following operational meaning. Whenever such a place is marked with a token and the transition is enabled the firing of this transition does not change the state of the place. Note that the implementation of Select uses a component called *Transparent Latch* (marked L in the figure). It is described by the following equation: $out' = dc + out\ (d + \bar{c})$, where d is a 'data' input and c is the 'clock' input.

Call is a module which allows us to 'multiplex' multiple occurences of the same action in the LPN model. It is similar to a procedure call in a program. Each separate request to execute a unique action a is multiplexed onto the single request. When the action is complete, the corresponding acknowledgement is generated. The two-way version of Call has two handshake interfaces $(R1, D1)$ and $(R2, D2)$ for the signals directly translated from LPN transitions labelled with action a and a single handshake R, D for the unit associated with a. We should recall that, in the case of multiple labels, one has to verify first that no two or more transitions bearing the same label are enabled concurrently. Then

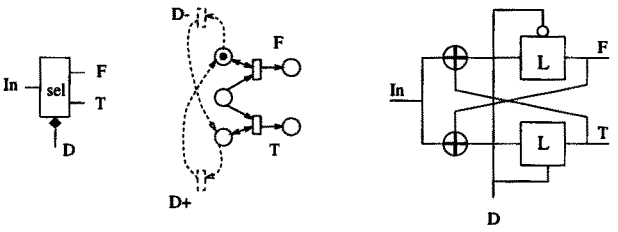


Fig. 33. Select module: Petri net fragment and circuit.

there are two possibilities. The first one would be to use a multiplexing construct like Call. An alternative would be to produce such a multiplexer at the LPN level, by transforming the net into one with injective labelling [41].

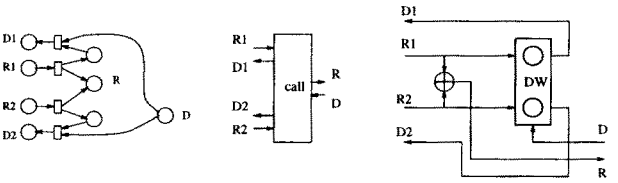


Fig. 34. Implementing multiple labelling: Call module.

Finally, a *Toggle* is another useful macro-component, which allows simplification of the logic if the initial specification contains transitions with different occurence numbers in the operational cycle of the process, e.g. in models of counting circuits. Toggle's possible implementation also uses transparent latches.

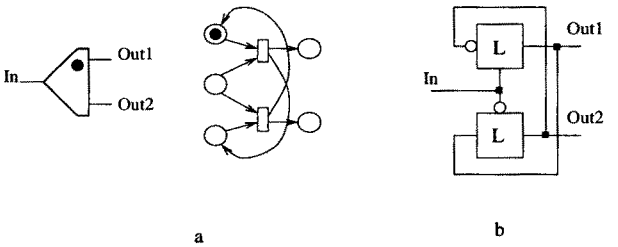
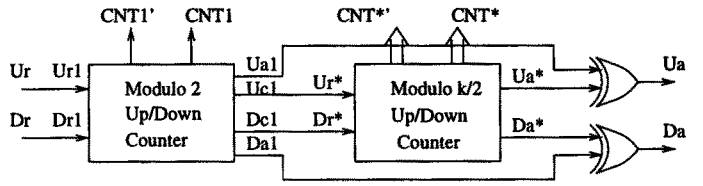


Fig. 35. Toggle module: Petri net fragment and circuit.

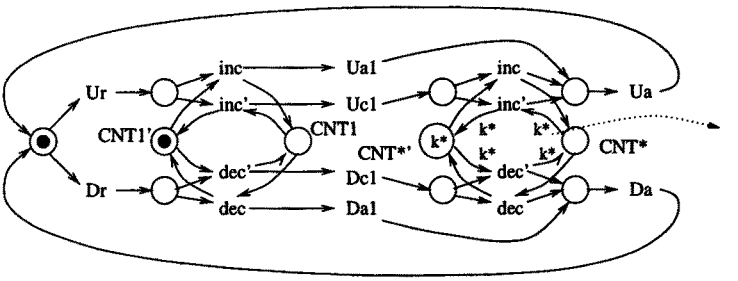
Direct synthesis example: modulo-k Up/Down counter. As an example, let us consider the synthesis of a modulo-k Up/Down counter that uses the event-based signalling discipline. We assume that the environment always guarantees

mutual exclusion between sending requests for Up and Down operations. We refer to the abstract design of such a counter shown in Figure 17 and Figure 18. Let k be a power of 2 for the sake of simplicity, as it allows a recursive decomposition of a modulo-k counter into a modulo-2 and modulo-$k/2$ ones, shown in Figure 18.

Let us first refine abstract actions U and D into handshake pairs of signals: Ur and Dr for requests of Up and Down counting respectively, and Ua and Da for acknowledgements. We also refine pairs of events $(inc1, inc'1)$ and $(dec1, dec'1)$ to produce intermediate pairs of acknowledge signals $(Ua1, Uc1)$ and $(Da1, Dc1)$.



(a)



(b)

Fig. 36. 'Recursive' decomposition of a modulo-k Up/Down counter at the logic design level (note that $k* = k/2 - 1$).

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 is incremented while in state 1. Signals $Da1$ and $Dc1$ have similar functionality, respectively, for the decrement operations.

The model of the first stage of the circuit is shown in Figure 37(a).

In order to implement this net by a circuit in Patil's style, we first transform it to an observationally equivalent net, shown in Figure 37(b). In the new net, each signal event has a unique transition associated with it, and we can perform a syntax-directed translation of it into a circuit, using the component models described in the previous section. Note that the 2-by-2 DW synchronises an event occurring on exactly one of the two inputs $B+$ and $B-$ with either event on Ur

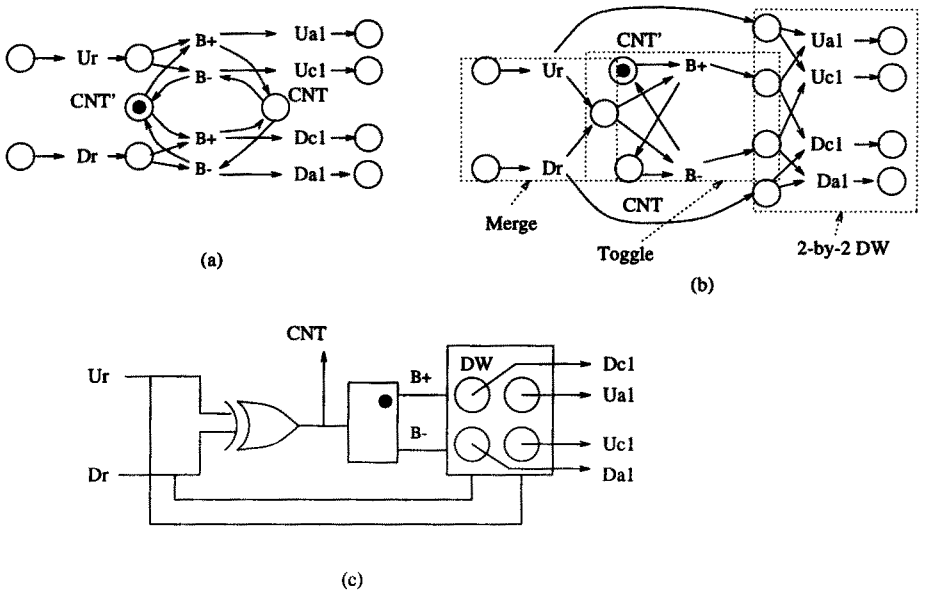


Fig. 37. Circuit implementation for one bit Up/Down counter stage.

(Up-counting) or on Dr (Down-counting). The lower level implementations of the 2-by-2 DW can be found in [47].

It is easy to see 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 recursively refine the modulo-$k/2$ counter in a similar way, until k becomes equal to 1, to finally obtain the circuit implementation. This design is speed-independent since the change of signal CNT in each bit stage is always acknowledged by the corresponding completion signals. The signals $Ua1, Uc1, Dc1$ and $Da1$ always change last in each stage.

Another example of direct translation synthesis is shown in Figure 38. Part (a) shows a two-phase signalling model for a "lazy ring" arbitration adapter [70, 67], whose implementation is depicted in part (b). Note that places labelled $t = 0$ and $t = 1$ stand for the status of the arbitration token (hence use of name t), either being held outside or inside the adapter, respectively. For brevity, we do not show the intermediate STG refinement which has led to this circuit. Note, however, that this is an example of a non-ouput-persistent specification, which requires mutual exclusion expansion. The result of such an expansion is an RGD arbiter which 'protects' conflicts between the two dummy transitions corresponding to non-input signals. The reader may refer to [67] for details of designs of ring arbiters using Petri nets.

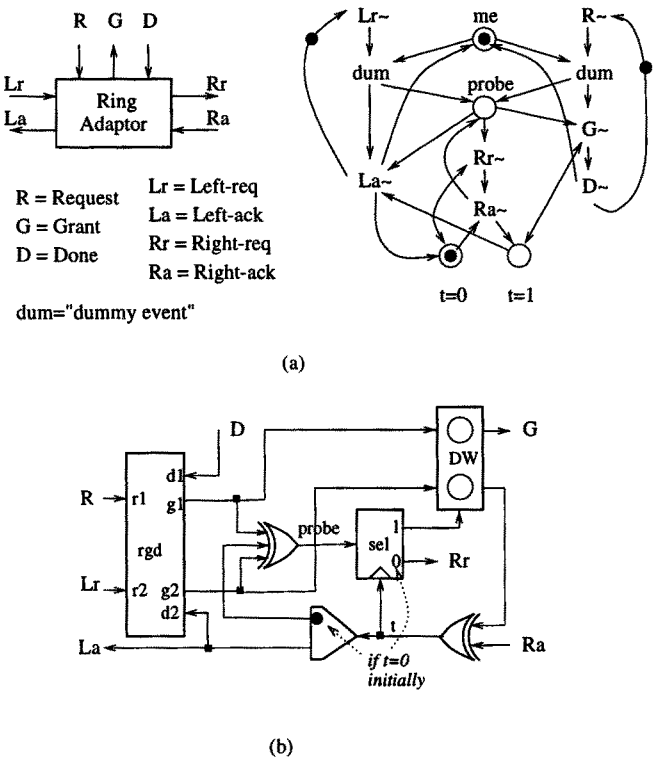


Fig. 38. "Lazy ring" arbitration adapter example: Petri net model with NRZ signalling (a), result of direct translation into circuit (b).

5.4 Synthesis based on logic minimisation

Unlike the direct translation approach, this approach implements STG specification by means of deriving its logic through a process similar to classical logic synthesis. For each non-input signal this process finds a Boolean cover over the states in which the signal is supposed to be equal to logical 1. This Boolean cover is then mapped into a physical interconnection of logic gates, providing appropriate constraints which would guarantee freedom from hazards in the resulting circuit.

The overall process, though quite straightforward in principle, can be highly non-trivial when the designer wants to ensure that the eventual implementation is totally insensitive to the delays of every single gate in the circuit.

Let us first examine the most general conditions that an STG specification should satisfy in order to be convertable to Boolean logic, that is, without any particular restriction on the type of Boolean covers or equations it may generate.

First of all, as has already been discussed in Section 5.1, an STG specification must satisfy the following properties:

- (i) boundedness of the underlying PN, to guarantee that the final circuit has a finite size;
- (ii) STG validity, or consistency of its the binary encoding of its Reachability Graph, to allow a meaningful interpretation in terms of binary states and to be able to generate a State Graph;
- (iii) output-persistency of its State Graph, to guarantee a hazard-free behaviour of the discrete logical gates derived for each non-input signal, so that such behaviour is observationally equivalent to the original specification.

These properties are all necessary for performing logic synthesis from an STG or its SG. Thus, the designer needs to correct the STG if it does not deliver any of them. The reason why it is the designer who is responsible for modifying the STG is that such corrections always affect observational equivalence between the original model and the final behaviour produced by the implementation. In fact, output-persistence appears to be a property which can often be fixed formally, by insertion of mutual exclusion events into the net. Unfortunately, none of the existing software tools are capable of doing this as yet.

Complete State Coding. In addition to the properties discussed above, there is also a property necessary for logic synthesis which can be provided by means of formal algorithms. This property is called *Complete State Coding* (CSC). A bounded and valid STG G has the CSC property if for all reachable markings with the same binary coding vector, the corresponding sets of enabled non-input transitions are equal. More specifically, if there exist a pair of markings m_1 and m_2 with the same code and in one of them a signal y is enabled while in the other y is stable, we say that there is a *state coding conflict* between m_1 and m_2 with respect to y. An STG which has state coding conflicts is said to have a CSC problem.

A simple example illustrating a CSC problem is shown in Figure 39. In part (a) an STG is shown that specifies an autonomous circuit (a pulse generator without inputs) with two outputs x and y. The RG of the underlying Petri net is shown in part (b), and its binary encoded version, the SG, is shown in part (c). It should be obvious that the SG has two states in coding conflict, both labelled with 00, but with a different enabling of the output signals. In order to resolve this conflict, an additional state signal z has been inserted. The transitions of z do not change the original ordering between output signals x and y, i.e. the new STG, shown in part (d), is observationally equivalent to the original one. Part (e) depicts the SG of the new STG and demonstrates that the conflict between the states originally in conflict has been resolved - originally encoded 00, they are now distinguished by the different values of the new signal z.

Another example of a CSC problem is shown in Figure 40. It is an SG obtained for the STG presented in Section 2.6 in Figure 6. Here we have two pairs of conflicting states, encoded with 100011 and 010111, respectively. Both conflicts have been resolved by adding one state signal, which results in the circuit shown in Figure 7.

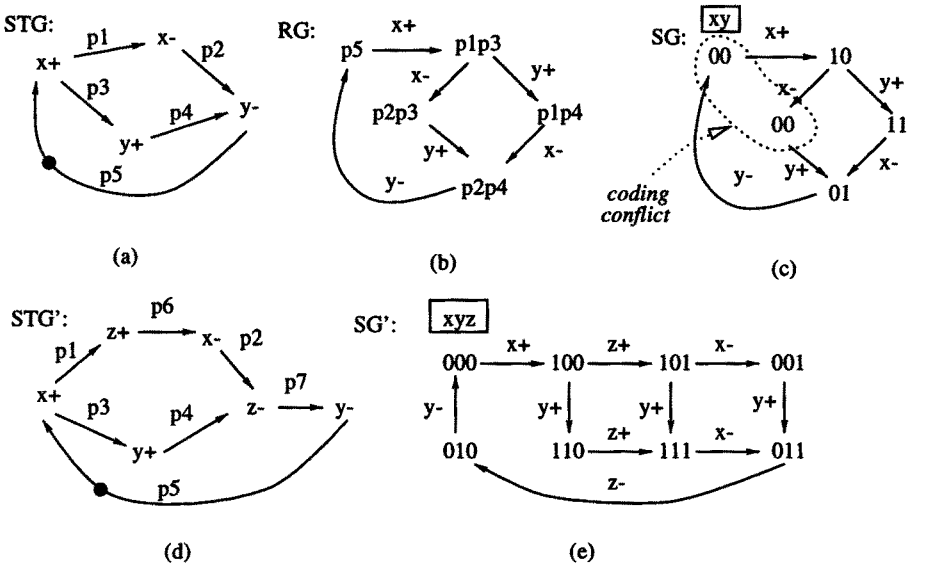


Fig. 39. Simple example to illustrate the Complete State Coding property: (a) initial STG with two noninput signals x and y, (b) its Reachability Graph, (c) State Graph with state coding conflicts, (d) modified STG with extra state signal z and (e) its State Graph.

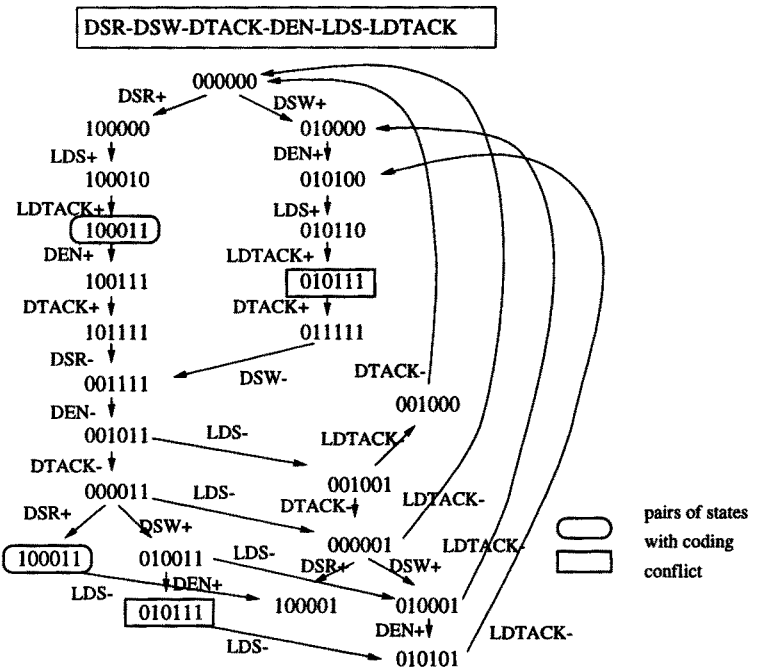


Fig. 40. State graph for VMEbus adapter example (STG shown in Figure 6).

The subject of solving the CSC problem has been actively investigated during the last decade, with initial research being aimed at checking the CSC property for STGs [19, 104, 135], and identifying coding conflicts [125] for subsequent manual resolution. The most recent work has been focused on finding fast and effective algorithms for solving CSC by means of automatic insertion of new signals [65, 55, 141, 21]. These investigations have given rise to a number of theoretically challenging, mutually related subproblems:

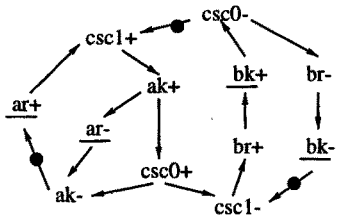
- the problem of identifying state coding and other types of conflicts amongst the reachable states; some of these conflicts may be based on subvectors of state codes, and involve specific subsets of states (cf. work on Monotonic Cover conditions [60]).
- the problem of efficient partitioning of the state space into subsets to be encoded by new state signals;
- the problem of new signal event insertion, whilst preserving the observational equivalence, output-persistency and input-output interface.

A key role in solving the partitioning and event insertion problems is played by the notion of regions (see Section 4.4) in the state graph. The techniques proposed in [21] exploit the fact that the finest granularity level at which new signals can be inserted is provided by regions, intersections of regions and unions of intersections. This approach solves many problematic CSC cases which none of the previously known methods have been able to deal with in an efficient way. Other tools, like assassin, had to apply some event re-ordering to the original STG specification in order to insert new events. Another useful feature of the region-based approach [60] is that the method, despite working at the state graph level, generates a new STG from the modified SG. The ability of the designer to review the CSC-compliant STG is important, as it is a good reference point for manual optimisations, reverse engineering of the circuit, or simply for the purpose of visual representation of the circuit behaviour in a more compact form than the state graph, with its interleaving of concurrency.

In fact, the four-phase STG shown in Figure 24 is one of the "difficult" examples for formal tools. The reader may, as an exercise, like to construct its SG and observe a large number of state coding conflicts. Another version of signalling refinement, shown in Figure 25, does not have the CSC problem. It is more constrained with respect to the level of parallelism between events in the a and b handshakes. The version in Figure 24, which allows for more loosely coupled synchronisation between these handshake protocols, may be prefered by the designer. One of the possible CSC solutions is shown in Figure 41. It requires the addition of two signals to resolve all the conflicts in the original model.

Logic synthesis and logic decomposition

Basic ideas about logic implementation. An STG specification which satisfies the basic implementability conditions (bounded, consistent, output-persistent), and



Logic implementation:

ak = csc0' csc1 + ar ak

br = csc0 csc1'

csc0 = ak csc1 + (bk' + csc1) csc0

csc1 = ak' ar csc0' + (bk + csc0') csc1

Fig. 41. STG with Complete State Coding (original STG shown in Figure 24).

whose SG has the Complete State Coding property can be used for logic synthesis. Truth tables can be obtained from the SG state codes for each non-input signal. The process of obtaining a truth table depends on a particular implementation architecture chosen for each signal. For example, we can extract truth tables for functions associated with the states where a signal y_i is *enabled*, or it is *stable*, or its *implied value* is equal to 1 etc. The implied value for signal y_i in state m with binary code $v(m)$ is defined as the complement of its binary value $v_i(m)$ if y_i is excited in m, $v_i(m)$ otherwise. For example, in state 000 in Figure 39 (e), the implied value of x is 1 (x is enabled), and the implied value of y (stable) is 0. Thus, the functions defined by these truth tables are essentially associated with certain subsets of states, unified by a specific feature, such as the implied values.

The process of deriving such functions is central to the implementation stage – the obtained Boolean covers are directly associated with logic elements in the circuit. A Boolean function *covers* a state $m_j \in S_N$ if the function evaluates to TRUE when the variables have values equal to the signals in the binary code of m_j. A function *covering* a set of states is called a *cover function* or simply *cover*. Each term of the cover is called a *cube* as it may cover several states in the state space.

Deriving logic for typical implementation architectures. The following architectures are commonly used for the synthesis of speed-independent circuits:

1. each non-input signal is associated with an *atomic complex* gate; this architecture is a basic type in which every signal is implemented by a single complex-gate that may include internal feedback.
2. each signal is associated with a memory element, or latch (e.g. SR-flip-flip or C-element), with two excitation functions (e.g. S for set and R for reset) controlling the latch; these excitation functions are implemented as atomic complex gates;
3. similar to the above except that the excitation functions are decomposed into simpler gates (combinational decomposition) or latches (sequential decomposition); the gates and latches are from a restricted subset called the *library cells*.

We do not consider here all these architectures, and the logic synthesis aspects related to them. These can be found in [58, 112].

The simplest from the point of view of the synthesis process is the first type, based on a complex Boolean cover for each signal. Such a cover can be either combinational or sequential. In the latter case, the signal function includes the signal itself. Thus each atomic complex gate corresponding to this cover contains both a combinational part and a sequential part implemented as an internal feedback, where delays are assumed to be negligible. To derive such covers, two mutually complementary subsets of the reachable states are distinguished in the SG for every non-input signal y, *on*-set(y) and off-set(y), which include all states in which the value of signal y is implied to be 1 and 0, respectively. The remaining binary combinations, which form the unreachable state set, are regarded as a *dc*-set(y) (*don't care set*). The implementation of y is derived by finding a Boolean cover for the *on*-set(y). The *dc*-set(y), if it is nonempty, can be used for minimisation (by applying standard minimisation tools such as Espresso [8]). An illustration of this process is shown in Figure 42, continued from Figure 39. Here, part (c) shows a Karnaugh map for the *on*-set(z) with *dc*-set(z) being empty, due to the use of all eight binary combinations to encode the eight reachable markings of the STG. The minimal size cover for *on*-set(z) is $x + y'z$. Following this process for all three non-input signals (recall that the designed circuit is an autonomous pulse generator), we obtain three complex gates, whose functions are as follows: $x = z'(y' + x), y = x + z$ and $z = x + y'z$. These gates are in fact not so complex and can be easily implemented directly in the form of an NMOSC or CMOS transistor network (cf. Figure 2 for C-element). They can also be easily embedded into standard library cells: x into a Reset-dominant SR-latch, with set function $S(x) = z'y'$ (thus only y' can be connected to input S of the latch), reset function $R(x) = z$; y into an OR gate; z into a Set-dominant SR-latch, with $S(z) = x$ and $R(z) = yx'$ (only y can be connected to input S of the latch).

The reader may have noticed the use of inverted inputs (literals with primes, e.g. x') in these equations. This issue is neglected here in the sense that we do not consider inverters as separate gates with a delay at the output. This is not always a completely fair assumption (it often relies on certain technological or timing constraints) and an interested reader may refer to [127] for synthesis techniques avoiding use of such implicit inverters.

Correctness issues in logic decomposition. The complex gate implementation is speed-independent by construction, i.e. any bounded delay attached to any of the three outputs x, y and z will not change the behaviour of logic that is observationally equivalent to the original specification. Indeed, since during the synthesis we have not created any new gates, there will be no deviation from the original specification.

Other types of implementation require decomposition of the complex gates, and as a result may be prone to hazards. For example, even the second type, based on separate implementation of excitation functions, may lead to potential behavioural deviations. This type of implementation requires the derivation of *hazard-free* (sometimes also called *hard* or *monotonic* [60]) covers for excitation functions, for particular latches. An example of such cover for a set function

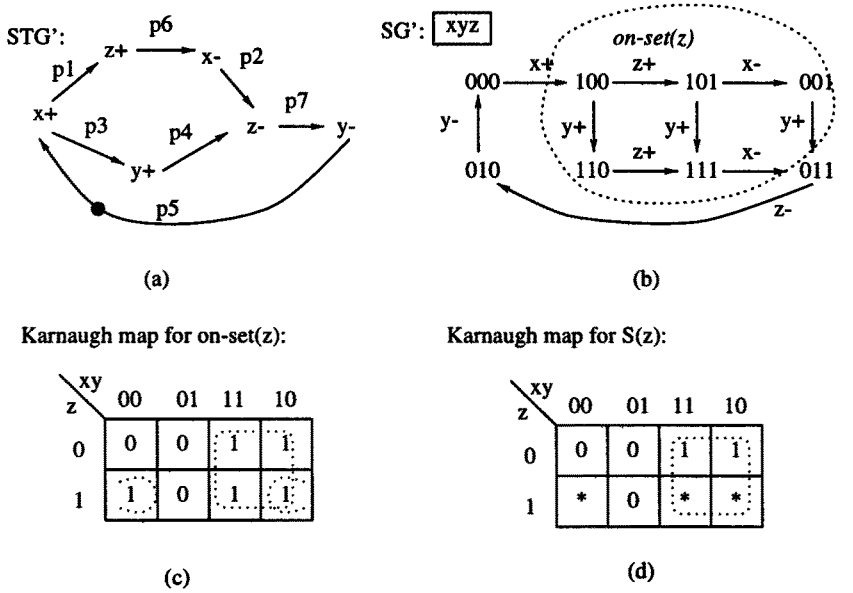


Fig. 42. Synthesis by means of finding Boolean covers: (a) Signal Transition Graph, (b) its State Graph, Karnaugh maps for the implementation by (c) complex gates and (d) via excitation functions.

$S(z) = x$ of signal z is shown in 42(d). It is obtained as follows. The set cover for a signal *must cover* all states in which the signal has value 0 and is enabled (the excitation region), and it *may cover* (these states contribute to the *dc*-set for this cover) any state in which the signal is stable and has value 1. Informally, the monotonicity condition requires that this cover never evaluates to TRUE outside the *on*-set of the function and never changes its value from TRUE to FALSE more than once in all feasible sequences passing through the *on*-set. The cover $S(z) = x$ is hazard-free, because it is just a single literal and is guaranteed to be correct by construction (see the above discussion about complex gates).

Discussing these aspects in more detail would require introducing many new notions, and we therefore refer to work on logic synthesis and decomposition of speed-independent circuits [13, 58]. We mention one important aspect which has a close link with Petri net techniques. This aspect concerns the representation of the semantics of the STG specification appropriately for logic synthesis.

Approximate cover approach. Two major approaches for deriving Boolean covers for signals in STGs have been investigated in the literature. The traditional one, which has been discussed above, is based on exact covers. An *exact* cover for a set of states S can be obtained directly from the set of their binary codes $v(m_j)$. However, it requires an explicit enumeration of all the states. Generating exact covers is very costly due to the exponential number of states that highly concurrent STGs may contain — this is known as the state explosion problem. To

overcome this, *approximate* covers can be generated using structural information from the STG, which avoids state explosion [95, 94]. For example, the method discussed in [95, 94] works with free-choice nets and therefore applies powerful algorithms based on State Machine decomposition of free-choice nets [40]. Another approach [112] is based on the use of Petri net unfoldings [75], which offer a compact representation of the semantics of highly concurrent STGs. These methods use the notion of Boolean cover approximations for STG places and transitions. A place p is approximated by a Boolean cover $C(p)$, defined on the support of binary signals Y, which consists of the literals of those signals which are not concurrent to p. The concurrency relation between places and transitions is a key notion in these techniques. It can be easily calculated from the STG unfolding, since the latter is an acyclic graph.

Each such initial place cover is a simple *cube*; approximate covers for transitions are found as intersections of the covers of their predecessor places. When these initial covers of the elements of the Petri net structure are found, the synthesis process goes to the stage in which the approximate covers for the necessary state sets are calculated. For example, for a complex gate implementation (type 1), this amounts to finding an approximation of the on-set(y) and off-set(y) for each non-input y, and checking if they are non-intersecting. If they are, the synthesis process is finished; otherwise the approximate covers must be *refined*. The algorithms for finding approximate covers for specific sets of states, and their refinement using the unfolding information, are not trivial [112]. They are based on the notion of cuts (sets of place instances), slices (sets of cuts) and their specific types (e.g. min excitation cut, max stable cut, excitation slice etc.), as well as the relations of precedence, conflict and concurrency between the elements of the unfolding.

Figure 43 depicts the unfolding of the STG from Figure 42. The approximate place covers are shown next to their places (i.e. arcs in the shorthand notation). As an example, the Set and Reset excitation covers are shown for signal z. For this example, they coincide with those obtained by the exact method (one should of course exclude the z literal itself as a support if using those functions as inputs to an SR-latch to implement signal z), so the refinement is not needed, which is not true in general case.

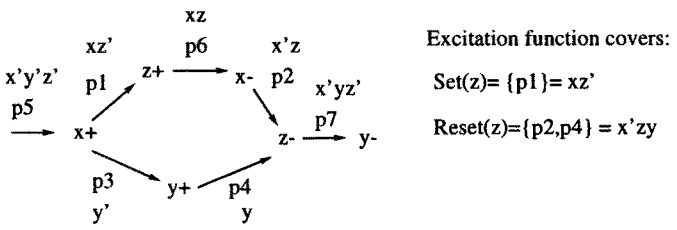


Fig. 43. Finding approximate excitation covers from an STG unfolding.

We conclude this section by referring the reader back to Figure 7, which shows the logic implementation of the VMEbus interface adapter, obtained using Petrify. This is a complex gate solution. Current research is focused on obtaining efficient algorithms and cost functions for the hazard-free decomposition of this logic into specific libraries, such a decomposition is called *technology mapping*. Initial work has been presented in [58, 24], which develops the theory and software implementation for technology mapping of speed-independent circuits. This technique combines a number of algorithms such as Boolean relations for functional decomposition of complex Boolean covers, and speed-independent preserving sets, based on regions, for correct insertion of new signals. Future work should also tackle aspects of *timing-driven* synthesis of control circuits from STGs, based on the information about delays of library elements, to increase the effectiveness and performance of the solutions without sacrificing their hazard-freedom properties.

5.5 Verification of asynchronous control circuits with Petri nets

This tutorial is primarily about use of Petri nets for the synthesis of asynchronous controllers. However, in previous sections we used the term 'design' more often than term 'synthesis'. Design obviously assumes both synthesis and verification of design solutions. Circuits synthesised by means of formal methods are guaranteed to be formally correct. But in most practical cases designers construct circuits either completely manually (because the tools for automated synthesis are not mature enough) or introduce changes into the synthesis results by using their experience and intuition. This seems to be inevitable, since it is going to be a long time before tools would be able to take into account all possible heuristics concerned with re-ordering of events in STGs or with the role of delay information in logic decomposition. Furthermore, designs would often be produced in parts and then assembled together, with glue logic included manually. Such cases would obviously require the application of formal verification techniques.

Petri nets have a good track record as a formalism for capturing complex discrete event systems, and for verification with respect to a set of correctness conditions. The correctness conditions are classified into the following major groups: (i) *safety* requirements, (ii) *liveness* requirements, and (iii) *conformance* between implementation and specification. Often, however, group (iii) is "dissolved" between the first two groups. Safety conditions say that "something bad will never happen in the circuit", and liveness conditions say that "something good will eventually happen".

Asynchronous designs are a typical example of discrete event systems. Their typical safety requirement is that a circuit must be *free from hazards*. Another safety requirement is that the circuit, in its interaction with the environment that never stops providing stimuli to the circuit's inputs, *never reaches a deadlock* state.

The liveness properties are less of an issue for asynchronous designs, at least at the discrete logic level. Indeed, these qualities are usually guaranteed by the

finite nature of delays in the logic gates, and by the absence of nondeterministic choice in asynchronous designs if they are free from hazards. In the latter case we certainly should exclude some circuits such as arbiters, which produce nondeterministic behaviour due to inherent metastability. But their analysis for boundedness of delays in resolving metastability can be done with "probabilistic pragmatism", i.e. by proving that an arbiter responds in a given time *almost surely* [80]. To fully model and verify such properties for arbitration and other internally analogue devices, one needs the modelling power of dynamical systems, which falls outside the scope of this discussion.

Let us briefly outline the major ideas about circuit verification for hazard-freedom at the discrete event level. A hazard in an asynchronous circuit is a spontaneous deviation of the circuit's behaviour from its prescribed specification. This deviation is usually "local" to a specific signal or a gate's output, where the hazard exhibits itself as a short pulse or *glitch*. The glitch is caused by a change of the input conditions for the gate at a time when the gate is enabled to switch its output according to the specification. Such conditions may arise as follows. The gate is the result of a logic decomposition and its signal y' is not included in the list of the original signals y. The gate's function is therefore a cover which evaluates to TRUE only in some states belonging to, say, *on*-set(y), and resets to FALSE in the other parts of the *on*-set(y). It is easy to imagine that this gate, under particular delay conditions (maybe it is too slow to complete its reset to 0 when the new set condition arrives), may have a glitch at its output. Logic decomposition does not therefore guarantee speed-independence. Another example of potentially hazardous behaviour could be a manual logic design which takes into account some timing constraints between delays in the circuit and in the environment that are not satisfied in reality. We need a means to model the circuit and check if its composition may produce such an undesirable effect.

Three basic questions must be answered in the context of the use of Petri nets as a modelling language:

(1) How to model logic circuits and their environment with Petri nets?

(2) Which properties of Petri nets could serve as an analogue of those of hazardous or hazard-free behaviour in the circuit?

(3) What type of the Petri net semantics and what kind of analysis algorithms are to be used for checking the properties identified by the answer to question (2)?

Question (1) can be answered in at least two ways:

(1.1) Represent each circuit gate as a finite state Petri net component, connect those Petri nets together according to some rules of Petri net composition, extract the model of the environment from the circuit specification, and connect it to the overall Petri net. The whole system can then be analysed as a composition of Petri nets.

(1.2) Represent each gate as a Boolean equation, regard each equation as a discrete model which has precise enabling and stability conditions, and build a Petri net model of the environment from the circuit specification. The entire system will therefore consist of two parts which can have finite state behaviour

and can be traversed simultaneously for analysis, satisfying the conditions of compatibility on the signals in the interface between the circuit and its environment.

Let us consider briefly the (1.1) approach (more details can be found in [139, 109]). This approach can be subdivided into two modelling techniques, depending on the signalling type of the logic circuits. These are: level-based elements and event-based elements. These two types directly correspond to the types of signalling expansion of labelled Petri nets – the four phase and two phase ones (see Section 5.2). Level-based elements are ordinary logic gates, such as ANDs, NANDs, ORs, NORs, AND-OR-NOTs. Sequential elements are described by a complex gate equation of the form $Y = S + R'y$, where S is a set subfunction and R is a reset subfunction. Event-based elements are less conventional, they include micropipeline control elements, Join (C-element), Merge (XOR-gate), Toggle, Select, Call, RGD-arbiter (see Section 5.3) [118]. Their Petri net models can be built along the lines of Patil's approach to the direct translation of Petri nets to circuit, discussed in Section 5.3). For example, Join is modelled by a transition synchronising two predecessor transitions; Merge is modelled by a place with two input transitions assumed to be fired on a mutually exclusive basis.

Modelling level-based logic with Petri nets. The idea of representing a logic circuit built from level-based components (i.e. ordinary logic gates, as opposed to event-based micropipeline elements [118, 139]) by so-called *Circuit Petri nets* was described originally in [37] and refined in [127]. A Circuit Petri net 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 $y+$ 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 logic. The actual input "guards" for these transitions are formed by using *self-loop* (i.e. read-only) 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 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.

Figure 44 shows two simple examples of such models, an inverter and an OR-gate.

An example of a model of a level-based circuit is shown in Figure 45(a). This circuit is closed, i.e. it is autonomous and has no interconnections with its environment. The Circuit Petri net model is shown in Figure 45(b).

The (1.2) approach has been described in [102].

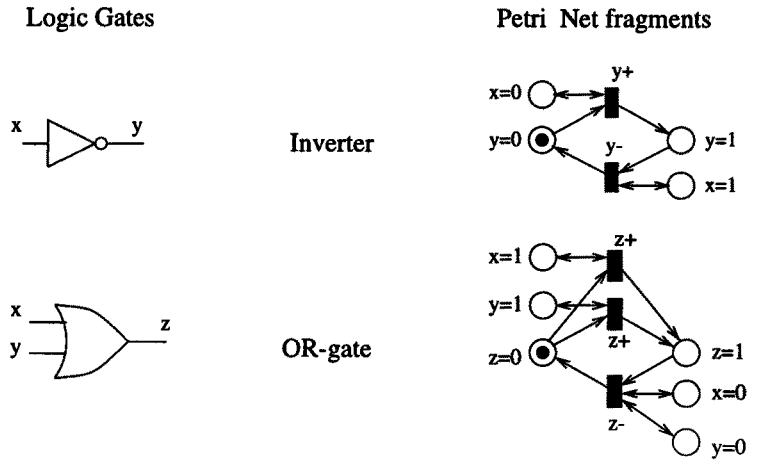


Fig. 44. Modelling ordinary logic gates (level-based elements) with Petri nets.

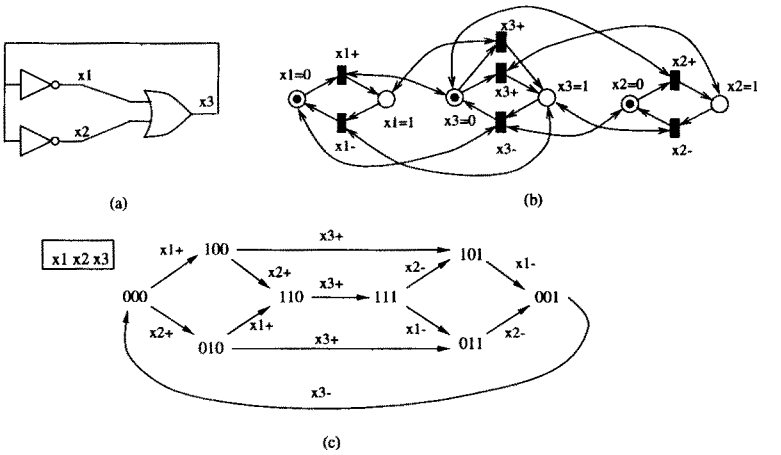


Fig. 45. Example of a level-based circuit model.

Analysis of hazard-freedom via net persistency check. Question (2) can be answered as follows.

The notion of a gate being enabled to make a transition of its output signal from, say, logical 0 to logical 1 is analogous to the notion of a Petri net transition being enabled by the set of its input places. Similarly, if the gate's enabling is removed before it has managed to switch, this can be modelled by removing some of the tokens from the input places of the corresponding transition in the net model. Therefore, potentially hazardous circuit behaviour can be mimicked by the nonpersistence of the corresponding net model. This sort of relationship between asynchronous circuits and labelled transition systems was noted by R.M.

Keller [51]. Thus, methods for checking Petri net nonpersistency can be used for hazard analysis.

Let us return to the example of the Petri net model of a level-based circuit shown in Figure 45(b). The behaviour of the circuit can be analysed using the state graph shown in Figure 45(c). 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.

Note that for event-based circuits one can use the property of non-1-safeness to interpret hazardous behaviour [139]. For example, consider the model of a Merge (XOR) element with two inputs, shown in Figure 46. If one of the inputs changes its state before the output has been able to respond to the change of its other input, then this manifests potentially hazardous behaviour of the Merge. In the Petri net model, this would correspond to the arrival of two tokens into place p – thus causing the net to be non-1-safe.

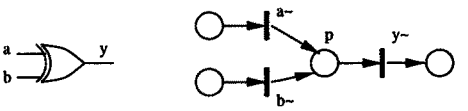


Fig. 46. Model of an XOR gate.

Finally, let us briefly answer question (3); again, more details can be found elsewhere [139].

Avoiding state explosion in hazard-freedom analysis. Several methods have been suggested to overcome the state space explosion of the straightforward reachability state set analysis. Among those are Petri net *symbolic traversal* [96], *stubborn set* methods [122] and Petri net *unfoldings* [75]. Symbolic traversal uses *implicit representation* of the reachability set in the form of Binary Decision Diagrams [11] (BDDs) which are canonical representations of boolean functions in graphical form. Symbolic traversal has been shown to be efficient for analysis of "state-based" properties such as freedom from deadlock. However, this method does not immediately provide the relations between transitions.

Stubborn set methods use the fact that interleavings of concurrent transitions lead to the same marking. These methods *partially represent* the reachability set. Although efficient in finding deadlocks, they do not produce a complete

representation of the reachable state space, and checking for properties other than freedom of deadlocks usually involves exploring other states.

Petri net unfolding represents the full reachability graph using *partial orders* that preserve the relations between transition occurrences (a transition occurrence is a *unique* event associated with a single act of firing of the transition). Since all reachable markings are represented in the Petri net unfolding, the concurrency relation for two transitions can be easily obtained. It has been shown in [109] that the use of unfoldings of Circuit Petri nets should be treated with special care. Indeed, the presence of a large number of self-loop (or positive contextual) arcs presents a problem for the unfoldings of ordinary Petri nets, that is, those where the self-loop arcs are represented explicitly as two single headed arcs. Since in Circuit Petri nets the meaning of self-loop arcs is exactly like that of contextual or read-only arcs, treating them as a superposition of two ordinary arcs would be wrong. The size of the unfolding of such nets may grow exponentially with the number of concurrently enabled transitions which are incident via self-loop arcs to a a place. A special method of unfolding nets with positive contextual arcs has been developed and applied to the verification of logic circuits [109], with significant performance gain.

The effectiveness and efficiency of the above-mentioned analysis methods depends on the structural and behavioural type of the net. It is therefore advantageous to combine them. Petri net unfolding produces temporal relations between places and transitions. Such relations can then be used to optimise the order of variables in the BDD representing the reachability set. Experiments showed improvements from this synergy of methods [111].

Unlike ordinary (untimed) Petri nets, where every transition firing has no specific firing time or delay, a circuit transition is usually associated with an action that takes a finite amount of time. This amount is typically a physical delay associated with a signal change. If two transitions are fired concurrently, the overall time is the maximum of the firing times of the transitions, as opposed to their sum as in the case of sequential operation. A design in which a certain major module is decoupled from the rest of the circuit would be considered more time-efficient, thus suggesting a pipelined operation in the system. Such an operation is easily captured by a Petri net description.

Circuit models based on *Time(d) Petri Nets* are amenable to timing analysis and verification – a circuit whose behaviour may be incorrect in the delay-independent sense, may be perfectly acceptable under appropriate delay conditions (imposed by the environment and/or by the implementation parameters). Recent results in this area can be found in [103, 44, 110].

Petri nets and VHDL in asynchronous design. The combined use of Petri nets and conventional high-level hardware description languages such as VHDL is important at different levels of asynchronous design. Indeed, VHDL is a language with an increasingly powerful set of supporting commercial and academic tools for simulation and even synthesis of logical circuits. We imagine the role of the link between Petri nets and VHDL to be increasing and at least consist in the following.

The designer develops an asynchronous circuit from a formal Petri net model, such as an STG. Firstly, he visualises the behaviour of the specification in the classical and customary form of waveforms or timing diagrams. Secondly, the circuit is either designed by hand or synthesised by an asynchronous synthesis tool. If it is then modified by hand, it may need to be examined in a conventional way. The easiest way here would be to use existing software tools for simulation and visualisation. These two important tasks can be realised by means of creating a conversion tool from Petri net models, including STGs, to VHDL. Such tools have been described in [126, 117]. While the first reference focuses on the role of VHDL for simulation and testing of the already designed circuit, the second one demonstrates the involvement of VHDL representation forms at different intermediate synthesis stages, such as CSC resolution and logic decomposition.

The reverse link, from VHDL to Petri nets and their subsequent translation into asynchronous or synchronous logic, has been described in [32, 78].

6 Overview of Petri net based tools

Many software tools have been written to perform operations on Petri nets. It would be impossible to discuss all of them here. In general, the following functions are performed by such tools:

- net editing, usually through a graphical front-end;
- net simulation, often with graphical feedback, and report generation;
- generation of reachability graphs;
- verification of net properties, such as liveness;
- performance and timing constraint satisfaction analysis;
- synthesis of circuits, either synchronous or asynchronous;
- verification of behavioural properties of asynchronous circuits, such as hazard-freedom.

There is no single tool to perform all these functions, and the Petri net and/or circuit designer will usually have a set of tools as his disposal. Different tasks are performed with different levels of efficiency by different tools, so a designer may have clear preferences. Sometimes different tools produce different results on the same net, which may indicate the presence of bugs in the software!

Below we briefly discuss the software tools favoured by the authors. These tools are a mixture of general purpose Petri net tools and highly specialised tools for the synthesis of asynchronous circuits.

6.1 SIS

SIS [114] is a very popular software system for the synthesis of both synchronous and asynchronous sequential circuits. It was written at the University of California, Berkeley. It accepts input in the form of a state transition table (e.g. in

Berkeley BLIF format), a signal transition graph (in ASTG format), or a logic-level description of a circuit. From this it produces an optimized net-list in the target technology. The software contains a large number of different synthesis algorithms, which allow the user a great deal of choice at each stage of the design process.

SIS is freely available from http://www-cad.eecs.berkeley.edu, and is widely used throughout academia as a back-end synthesis system.

6.2 FORCAGE

FORCAGE is a comprehensive suite of software tools (for MS-DOS machines) developed originally by Trassa Co-op in St. Petersburg, Russia (later maintained at Technical University of Denmark and at the University of Aizu, Japan). This package is based on the theory of Change Diagrams, described in detail in the monograph by M. Kishinevsky, A. Kondratyev, A. Taubin and V. Varshavsky [55] (the book is normally supplied with a diskette). Change Diagrams are a close "relative" of Petri nets, and their relationship has been investigated in [132].

The package provides the following tools:

- TRANAL, to verify asynchronous circuits for speed-independence and semi-modularity (analogous to Petri net persistency [51], and hence characterising hazard-freedom). The circuit and its environment must be defined by its set of logical equations. The tool performs analysis by means of reachability set traversal.
- TRASPEC, to verify asynchronous circuits for speed-independence and distributivity (semi-modularity restricted with AND-causality [132] only) by means of building a Change Diagram that describes the behaviour of the circuit. Unlike TRANAL, this tool uses a polynomial algorithm (based on partial orders) that does not restore all states of the circuit.
- TRASYN, to check the logic implementability of a Change Diagram specification and to synthesise a speed-independent circuit implementing this specification.

FORCAGE is available through anonymous FTP from:
ftp://ftp.id.dtu.dk/pub/forcage.

Additionally, FORCAGE is compatible with a program for analysis of performance of speed-independent circuits. The program (which runs under Unix) computes the cycle time and critical cycles for a class of STGs based on marked graphs. This program is available from ftp://ftp.id.dtu.dk/pub/timesim/timesim.tar.Z.

6.3 Petrify

Petrify [22] is a tool which helps the designer in performing the following two tasks:

- manipulation of concurrent specifications
- synthesis and optimisation of asynchronous circuits

Petrify accepts the following behavioural descriptions: Petri nets, Signal Transition Graphs and Transition Systems (TSs). A TS is a state graph with the arcs labelled with abstract names of events.

Given a Petri net, an STG or a TS, Petrify generates another Petri net or STG which is simpler than the original description, and then produces an optimised net-list of an asynchronous controller in the target library while preserving the specified input-output behaviour. Thus, given a specification, Petrify provides the designer with a net-list of an asynchronous circuit and a Petri net-like description of the circuit behaviour in terms of events and ordering relations between events. The latter ability of back-annotating to the specification level helps the designer to control the design process. The final net-list is guaranteed to be speed-independent, that is hazard-free under any distribution of gate delays and multiple input changes satisfying the initial specification.

The tool is being developed as a collaborative work of research teams from Polytechnic University of Catalunya (Spain), Polytechnic of Turin (Italy), University of Aizu (Japan) and University of Newcastle upon Tyne. It is available from http://www.ac.upc.es/~vlsi/petrify/petrify.html.

6.4 ASSASSIN

ASSASSIN [140] is a powerful system for the sysnthesis of asynchronous control circuits from various input formats. It was developed at the IMEC institute in Leuven, Belgium. The following input formats are supported: STGs (using a language which is an extension of the SIS ASTG format), FSM descriptions (in particular, for the description of burst-mode circuits [92]), and CSP-based models. The approach allows the synthesis of heterogeneous specifications. The system has a particular emphasis on industrial applications.

ASSASSIN output consists of a low level description of a hazard-free implementation of a circuit in terms of logic equations and asynchronous memory elements. It is freely available.

6.5 Versify

Versify is a command-line driven tool that performs the following functions:

- verification of a speed-independent circuit against an STG specification;
- verification of STGs;
- verification of Petri nets.

The tool requires both the specification of a circuit and the specification of its environment. The tool performs reachability analysis, and checks for unexpected behaviour of the circuit as it reacts to changes in the environment. It was developed at the Polytechnic University of Catalunya (Spain). It is available from http://www.ac.upc.es/~vlsi/versify.

6.6 PUNT

PUNT [106] was written at the University of Newcastle upon Tyne. It uses a Petri net unfolding technique as the basis for analysis. PUNT implements both (Labelled) Petri net unfolding and the STG-unfolding segment techniques. This allows its use for analysis of asynchronous circuits as well as their specifications in the form of Labelled Petri nets and STGs. The tool allows analysis of the Petri nets/STG for safety, persistency, correctness, boundedness. In addition, the designer may derive concurrency relations between the transitions of the original Petri net/STG. This helps to determine the design's performance bottlenecks. The relations between transitions help the designer to arrive to a design with higher degree of concurrency and hence higher throughput.

PUNT supports the following input formats: PNS and ASTG. The former is used in the PNS Petri net simulator, which allows graphical input of the Petri net. The latter is used in a variety of current asynchronous analysis and synthesis tools. PUNT is a command line driven tool where the properties to be checked are specified as options at the command line. There is also provision for a menu which comes up after the unfolding/segment has been constructed. This allows interrogation of the segment in order to verify a particular property. If the property is violated, then at least one trace leading to the offending marking is produced. From this trace the designer can determine which corrections are required. In addition, PUNT is able to produce a variety of output formats for further processing of the original net in such tools as PROD and the unfolding/segment for use in graphical layout tools such as VCG and DOT.

6.7 PROD

PROD [39] is a general net analysis tool written at Helsinki University of Technology. It has no graphical frontend. Its main function is efficient reachability analysis. The main problem with reachability analysis is state explosion: the state space can become so large that it would be impossible to inspect all the states of the system.

PROD generates reduced reachability graphs using the following methods:

- stubborn sets, using several different algorithms;
- the so called CFFD preserving stubborn sets method;
- the sleep set method;
- the use of symmetries.

PROD is widely used in academia and industry. The tool and associated documentation is available from http://topos.hut.fi/~petrinet/prod.html.

6.8 UltraSan

UltraSan [121] is a highly sophisticated tool set developed at the University of Illinois at Urbana-Champaign. The tools allow model based performance, dependability and performability evaluation of systems, accessed through a graphical frontend. The user constructs a stochastic activity network for his system,

possibly in a hierarchical fashion. Reward variables are used to model dependability aspect. Simulation and analytic solvers are available to manipulate the models. Comprehensive report facilities are available.

The tool set is widely used in industry and academia. It is available from http://www.crhc.uiuc.edu/UltraSan.

6.9 PNIT

PNIT [56] is a high level framework developed at the University of Newcastle upon Tyne. It is written entirely in Tcl/Tk, and allows the following operations:

- editing of hierarchical Petri nets using a sophisticated graphical front-end;
- easy manipulation of the Petri net using the tools described above, by simply clicking on an appropriate menu item. PNIT will handle the input and output files, as well as the execution of the software;
- the use of a standard interchange format for Petri nets called PNIF to allow exchange of designs between different tools.

The overall idea is to give a designer easy access to a range of available design tools that are already available, and to make it easy to add new design tools to the system. In order to make interchange of designs between different tools possible, an interchange format, called PNIF, has been developed [57] which allows textual description of Coloured Petri Nets (CPNs) [46]. PNIF has a LISP-like syntax, and is heavily influenced by EDIF, the Electronic Design Interchange Format (version 2.0.0.). The basic types that are supported are boolean, (subranges of) integer, floating point, and enumerated types. PNIF supports a number of keywords to specify arithmetical and boolean expressions using these basic types. Programming language constructs are available through if and the while statements. A new colour (i.e. data type) for use in token expressions can be defined. It is also possible to declare variables, constants, arrays, and functions. Statements are available to assign values to variables. PNIF thus supports a full range of programming constructs. In order to allow the user to keep the complexity of the net within reasonable bounds, PNIF has provisions for hierarchical descriptions of nets, a feature frequently not widely implemented in Petri Net tools. Hierarchy is essential for reducing the complexity of net descriptions to manageable proportions. The net is modelled as a collection of small nets, which may instance each other.

7 Designing synchronous controllers from Petri nets

Petri nets have been used relatively little in the design of synchronous circuits. The reasons for this are mostly historical: the art of designing synchronous circuits was already firmly established when the theory of Petri nets was first developed. Computer aided design software was developed (e.g. schematic capture systems) to design larger circuits. These systems have now developed into very

big and very successfull commercial systems. The use of hardware description languages such as VHDL and Verilog as an input format to these systems is now widespread.

However, as hardware systems become ever more complex, modelling and verification of highly concurrent and parallell systems has become ever more difficult using the standard design methods. An FSM description is most useful whenever a single state is active at any time. Whenever there are multiple states active simultaneously in multiple concurrent processes, the FSM approach loses it attractions. The use of Petri nets is more suitable in these circumstances. State of the art work is being done at the University of Bristol [7]. Similar investigations have been reported, e.g. in [1, 68].

The following semantic interpretation is used to synthesize synchronous circuits from Petri nets. To every transition, a *predicate* is attached, which is a function of the input signals. The outputs of the circuits can be associated with either the places or the transitions. If they are associated with the places, they are called Moore outputs. If they are associated with the transitions, they are called Mealy outputs. Enabled transitions do not fire immediately, but instead do so when the next clock tick occurs (so all enabled transitions fire simultaneously). A transition is enabled when the input places are marked and the predicate is true.

It is normally required that the Petri net used to model a synchronous logic block is safe, that in any reachable marking the enabled transitions are not in conflict, and that there are no self-loops in the net (i.e. the net must be pure). These conditions guarantee that the digital circuit will be free from anomalous behaviour that might jeopardise its correct behaviour.

Once verified, the net can be subjected to reduction in order improve the efficiency of the generated hardware. Finally, an appropriate software tool can generate an appropriate output format that would be used by a synthesis tool to generate the circuit layout. An appropriate output format would be VHDL (particularly, a VHDL dataflow description), since there are currently many CAD tools that can synthesize VHDL. The CAD software would attempt to minimise the hardware using standard techniques such as Karnaugh Maps. The result would be a highly efficient circuit implementation.

It remains to be seen whether the use of Petri nets will become widespread in the synchronous design community, given the large investment in other techniques, and the inherent reluctance of the designers to change design methods that are perceived as having worked well for a large number of years.

8 Conclusions

In this tutorial we have tried to present our understanding of a "bridge" between two related areas of research, Petri nets and asynchronous hardware design. This relationship has been built historically and is believed to be highly cross-fertilising for the future benefits of both sides of the "bridge". We described our view on the process of asynchronous control circuit design using Petri nets.

The help of Petri nets in modelling the discrete aspects of asynchronous system behaviour seems indispensable, and allows synthesis and verification of circuits at different levels of abstraction. We also showed that Petri nets can be used for designing parallel synchronous controllers.

Our opinion is that today's work in this area should be focused (at least) on the following aspects:

- Improvements of high-level modelling techniques. There is a need for a greater synergy in methods for designing control and datapath circuits; this could be achieved, e.g., by searching for new ways of synthesis of asynchronous circuits from coloured or predicate-transition nets, which are already supported by powerful analysis tools [46].
- Investigations of new links between the discrete nature of Petri nets and the analogue or "hybrid" character of asynchronous circuit behaviour. Such investigations should tackle problems of better capturing circuit properties like causality and conflicts between electronic signals, metastability and hazards [132, 137]. Finding a flexible combination of Petri nets and dynamical system models (systems of differential equations and their phase spaces) [9] is of critical importance.
- Studies into hierarchical and reactive system modelling. There is a clear need for unified modelling tools for designing embedded and real-time systems, consisting of hardware and software components. Such systems need an adequate capture of the aspects of synchrony and asynchrony, parallelism and interrupts etc. Some attempts to find a model of that kind have been reported in [54].
- Further automation of the existing approaches to verification and synthesis. This work should seek greater efficiency in model checking for asynchronous design and synthesis of circuits from Petri nets. There is a great potential in structural methods and methods based on partial orders, e.g., developing better techniques for evaluation of boolean covers in Signal Transition Graphs [112]. A big challenge is the problem of timing analysis, where use of canonical techniques based on timed reachability graphs is absolutely impractical [110].

Finally, it is our sincere hope that the designers of the next generation of digital VLSI systems will use Petri nets as an underlying formalism for their design tools.

9 Acknowledgements

The authors would like to thank their friends and colleagues Jordi Cortadella, Mike Kishinevsky, David Kinniment, Alex Kondratyev, Maciej Koutny, Marta Pietkiewicz-Koutny, Luciano Lavagno, Lee Lloyd and Alex Semenov for stimulating discussions and long-standing collaboration. We are also grateful to all members of the two sister communities, represented by their mailing lists

PetriNets@daimi.aau.dk and asynchronous-private@pharos.cs.columbia.edu, for keeping us up-to-date on both fronts of knowledge.

This work was supported by EPSRC under grants GR/J52327, K70175, L28098.

References

1. A. Amroun and M. Bolton. Synthesis of controllers from Petri Net descriptions and application of ELLA. In L. Claesen, editor, *Proc. IMEC-IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, pages 57–74.
2. A.G. Astanovsky, V. I. Varshavsky, V. B. Marakhovsky, V. A. Peschansky, L. Y. Rosenblum, N.A. Starodubcev, R.L.Finkelshtein, and B. S. Tzirlin. *Aperiodic Automata.* Nauka, 1976. in Russian.
3. E. Badouel, L. Bernardinello, and Ph. Darondeau. Polynomial algorithms for the synthesis of bounded nets. Technical Report 2316, INRIA, RENNES Cedex, France, 1994.
4. M. Ben Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall International, London, 1990.
5. L. Bernardinello and F. De Cindio. A survey of basic net models and modular net classes. In *Lecture Notes in Computer Science*, volume 609, pages 304–351, 1992.
6. G. Berthelot. Transformations and decompositions of nets. In *Lecture Notes in Computer Science, Vol. 254*, pages 359–376. Springer-Verlag, 1987.
7. K. Bilinski and E.L. Dagless. High level synthesis of synchronous parallel controllers. In *Lecture Notes in Computer Science, Vol. 1091: Proceedings of the 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka*, pages 93 – 112. Springer Verlag, June 1996.
8. R. Brayton et al. *Logic Minimisation Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Hingham, MA, 1984.
9. R.W. Brockett. Analog and digital computing. In *Lecture Notes in Computer Science, Vol. 653*, pages 279–289. Springer-Verlag.
10. J. Bruno and S. Altman. A theory of asynchronous control networks. *IEEE Transactions on Computers*, 20(6):629 – 638, June 1971.
11. R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
12. J. A. Brzozowski and C-J. Seger. Advances in asynchronous circuit theory – part I: Gate and unbounded inertial delay models. *Bulletin of the European Association of Theoretical Computer Science*, October 1990.
13. S. Burns. General conditions for the decomposition of state holding elements. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Aizu, Japan*, March 1996.
14. C. Carrion and A. Yakovlev. Design and Evaluation of Two Asynchronous Token Ring Adapters. Technical Report no. 562, Department of Computing Science, University of Newcastle upon Tyne, October 1996.
15. J.C. Cavarroc, M. Blanchard, and J.Gillon. An approach to the modular design of industrial switching systems. In *Proceedings of the Int. Symp. on Discrete Systems, Riga*, volume 3, pages 93–102, 1974.
16. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

17. T.J. Chaney and C.E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.
18. T.-A. Chu. On the models for designing VLSI asynchronous digital systems. *Integration: the VLSI journal*, 4:99–113, 1986.
19. T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.
20. T.-A. Chu, C.Leung, and T.Wanuga. A design methodology for concurrent VLSI systems. In *Proceedings of the International Conference on Computer Design*, October 1985.
21. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete state encoding based on the theory of regions. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Aizu, Japan*, March 1996.
22. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *Proc. of the 11th Conf. Design of Integrated Circuits and Systems*, pages 205–210, Barcelona, Spain, November 1996.
23. J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri nets from state-based models. In *Proc. of ICCAD'95*, pages 164–171, November 1995.
24. J. Cortadella, A. Kondratyev, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Technology mapping of speed-independent circuits based on combinational decomposition and resynthesis. In *Proceedings of the European Design and Test Conference (ED&TC)*, pages 98–105, March 1997.
25. J. Cortadella, L. Lavagno, P. Vanbekbergen, and A. Yakovlev. Designing asynchronous circuits from behavioural specifications with internal conflicts. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah*, pages 106–115, November 1994.
26. J.B. Dennis. First version of a data flow procedural language. In *Lecture Notes in Computer Science, Vol.19*, pages 362–376. Springer-Verlag, 1974.
27. J. Desel and W. Reisig. The synthesis problem of Petri nets. Technical Report TUM-I9231, Technische Universität München, September 1992.
28. D.L. Dill, S.M. Nowick, and R.F. Sproull. Automatic verification of speed-independent circuits with Petri net specifications. In *Proceedings of Int. Conf. on Computer Design (ICCD'89), Cambridge, MA*, October 1989.
29. Jo C. Ebergen and Ad M. G. Peeters. Design and analysis of delay-insensitive modulo-N counters. *Formal Methods in System Design*, 3(3), December 1993.
30. A. Ehrenfeucht and G. Rozenberg. A characterization of set representable labeled partial 2-structures through decompositions. *Acta Informatica*, 28:83–94, 1990.
31. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures. Part I, II. *Acta Informatica*, 27:315–368, 1990.
32. P. Eles, K. Kuchcinski, Z. Peng, and M. Minea. Synthesis of VHDL concurrent processes. In *Proceedings of Euro-DAC'94*, pages 540–545.
33. J.C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3), May 1990.
34. F.C. Furtek. Modular implementation of petri nets. Master's thesis, MIT, September 1971.
35. H.J. Genrich and R.M. Shapiro. Formal verification of an arbiter cascade. In *Proceedings of 13th Int. Conferenece on Application and Theory of Petri Nets*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1992.

36. D.B. Gilles. A flow chart notation for the description of the speed-independent control. In *Proceedings of the Second AIEE Symposium on Switching Circuit Theory and Logical Design, Detroit, Michigan*, volume S-134, October 1961.
37. J. Grabowski. On the analysis of switching circuits be means of Petri nets. In *Elektronische Informations-verarbeitung und Kybernetik*, volume 14, pages 611–617. 1978.
38. M.R. Greenstreet and P.Cahoon. How fast will the flip flop? In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 77–86, November 1994.
39. 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.
40. M. Hack. Analysis of production schemata by Petri Nets. Technical Report TR 94, Project MAC, MIT, 1972.
41. M. H. T. Hack. Petri net languages. Technical Report TR-159, MIT, Laboratory of Computer Science, 1976.
42. S. Hauck. Asynchronous Design Methodologies. *Proceedings of the IEEE*, 83(1), 1995.
43. L. A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.
44. Henrik Hulgaard and Steven M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11, November 1994.
45. R. Janicki and M. Koutny. On equivalent execution semantics of concurrent systems. In *Lecture Notes in Computer Science, Vol. 266*. Springer-Verlag, 1987.
46. K. Jensen. *Coloured Petri Nets*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.
47. 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.
48. J.R. Jump. Asynchronous control arrays. *IEEE Transactions on Computers*, TC-23(10):1020–1029, October 1974.
49. J.R. Jump and P.S. Thiagarajan. On the interconnection of asynchronous control structures. *Journal of ACM*, (4):596–612, October 1975.
50. R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, May 1969.
51. R.M. Keller. A fundamental theorem of asynchronous parallel computation. *Lecture Notes in Computer Science*, 24:103–112, 1975.
52. D.J. Kinniment. Regular programmable control structures. In *Proceedings VLSI-81 (Ed. by J.P. Gray), Edinburgh*, August 1981.
53. D.J. Kinniment. Evaluation of asynchronous adders. *IEEE Transactions on VLSI Systems*, 4(2), March 1996.
54. M. Kishinevsky, J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev. Coupling asynchrony and interrupts: place chart nets and their synthesis. In *18th International Conference on Application and Theory of Petri Nets*, Toulouse, France, June 1997.
55. 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.

56. A.M. Koelmans, L. Lloyd, A. Semenov and A. Yakovlev. PNIT: a framework for the design of (a)synchronous circuits using Petri nets. In *Proc. ESPRIT ACiD-WG Workshop on Asynchronous Circuit Design, Groningen*, September 1996 (TR CSN9602, Computer Science Notes Series, University of Groningen).
57. A.M. Koelmans, D.J. Kinniment, Y. Xu and A. Yakovlev, PNIF: An Interchange format for system specification with coloured Petri nets, Technical Report Series No. 538, University of Newcastle upon Tyne, Computing Science, November 1995.
58. A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Technology mapping for speed-independent circuits: decomposition and resynthesis. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 240 – 253, April 1997.
59. A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev. Checking Signal Transition Graph implementability by symbolic BDD traversal. In *European Design and Test Conference, Paris*, pages 325–332. IEEE Comp. Society Press, N.Y., March 1995.
60. A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proceedings of the Design Automation Conference*, pages 56–62, June 1994.
61. A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten. A structural approach for the analysis of Petri nets by reduced unfoldings. In *Lecture Notes in Computer Science, Vol. 1091: Proceedings of the 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka*, pages 346 – 365. Springer Verlag, June 1996.
62. A. Kondratyev and A. Taubin. Verification of speed-independent circuits by STG unfoldings. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 64–75, November 1994.
63. A. Y. Kondratyev, L. Y. Rosenblum, and A. V. Yakovlev. Signal graphs: a model for designing concurrent logic. In *Proceedings of the 1988 International Conference on Parallel Processing*. The Pennsylvania State University Press, 1988.
64. A. Kovalyev and J. Esparza. A polynomial algorithm to compute the concurrency relation of free-choice signal transition graphs. In *Proceedings of Int. Workshop on Discrete Event Systems (WODES'96), Edinburgh*, pages 1–6. IEE, August 1996.
65. L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits*. Kluwer Academic Publishers, 1993.
66. D. Lewin. *Design of Logic Systems*. Van Nostrand Reinhold (UK), 1985.
67. K.S. Low and A. Yakovlev. Token Ring Arbiters: an exercise in asynchronous logic design with Petri nets. Technical Report Technical Report Series, no. 537, Department of Computing Science, University of Newcastle upon Tyne, November 1995.
68. R.J. Machado, J.M. Fernandes, and A.J. Proenca. Specification of industrial digital controllers with object-oriented Petri nets. In *IEEE Int. Symp. on Industrial Electronics (ISIE'97), Guimarães, Portugal*, July 1997.
69. L.R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, C-30(2):107–115, February 1981.
70. A.J. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.
71. A.J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.
72. A. Mazurkiewicz. Concurrency, modularity and synchronization. In *Lecture Notes in Computer Science, Vol. 379*. Springer-Verlag, 1989.
73. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

74. K. McMillan. Trace theoretic verification of asynchronous circuits using unfoldings. In *Computer Aided Verification, Proc. 7th Int. Conference*, volume 939 of *Lecture Notes in Computer Science*, pages 180–195, Liège, Belgium, July 1995. Springer-Verlag.
75. K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. *Formal Methods in System Design*, 1995. (to appear).
76. R. E. Miller. *Switching theory*, volume 2, chapter 10, pages 192–244. Wiley and Sons, 1965.
77. R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, NJ, 1989.
78. J. Mirkowski, K. Bilinski, and E.L. Dagless. Petri net modelling of VHDL simulation cycle for high level synthesis purposes. In *Proceedings VHDL Forum in Europe Spring'96 Working Conference, Dresden, Germany*, pages 35–46. Shaker Verlag, May 1996.
79. D. Misunas. Petri Nets and speed-independent design. *Communications of the ACM*, pages 474–481, August 1973.
80. I.M. Mitchell. Proving Newtonian Arbiters Correct, Almost Surely. Master's thesis, University of British Columbia, Canada, October 1996.
81. C.E. Molnar and H.M. Schols. The design problem scpp-a. Technical Report Technical Report TR-95-49, Sun Microsystems Laboratories, Mountain View, CA, December 1995.
82. U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 36:545–596, 1995.
83. D. Morris and R.N. Ibbett. *The MU5 Computer System*. Macmillan Computer Science Series, 1979.
84. M. Mukund. Petri nets and step transition systems. *Int. Journal of Foundations of Computer Science*, 3(4):443–478, 1992.
85. D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.
86. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.
87. C. Myers and T. H-Y. Meng. Synthesis of timed asynchronous circuits. In *Proceedings of the International Conference on Computer Design*, October 1992.
88. Christian D. Nielsen and Alain J. Martin. Design of a delay-insensitive multiply-accumulate unit. In *Proc. Hawaii International Conf. System Sciences*, pages 379–388. IEEE Computer Society Press, 1993.
89. M. Nielsen, G.Plotkin, and G.Winskel. Petri nets, event structures and domains. Part 1. *Theoretical Computer Science*, 13:85–108, 1981.
90. M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96:3–33, 1992.
91. M. Nielsen and G. Winskel. Petri nets and bisimulation. *Theoretical Computer Science*, 153(1-2):211–244.
92. S. M. Nowick and D. L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.
93. J.V. Oldfield and R. C. Dorf. *Field-programmable gate arrays : reconfigurable logic for rapid prototyping and implementation of digital systems*. John Wiley and Sons, Inc., 1995.

94. E. Pastor. *Structural Methods for the Synthesis of Asynchronous Circuits from Signal Transition Graphs.* PhD thesis, Universitat Politècnica de Catalunya, Barcelona, 1996.
95. E. Pastor and J. Cortadella. Polynomial algorithms for the synthesis of hazard-free circuits from signal transition graphs. In *Proceedings of the International Conference on Computer-Aided Design*, November 1993.
96. 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*, Zaragoza, Spain, June 1994.
97. S. S. Patil and J. B. Dennis. The description and realization of digital systems. In *Proceedings of the IEEE COMPCON*, pages 223–226, 1972.
98. N.C. Paver. *The Design and Implementation of an Asynchronous Microprocessor.* PhD thesis, University of Manchester, 1994.
99. M. Pietkiewicz-Koutny and A. Yakovlev. Non-pure nets and their transition systems. Technical Report Technical Report Series, no. 528, Department of Computing Science, University of Newcastle upon Tyne, September 1995.
100. C. Ramchandani. Analysis of asynchronous concurrent systems by Petri nets. Technical Report MAC-TR-120, MIT, Project MAC, February 1974.
101. I. Reicher and M. Yoeli. Net-based modeling of communicating parallel processes with applications to VLSI design. Technical Report 532, Technion, Haifa, 1988.
102. O. Roig, J. Cortadella, and E. Pastor. Hierarchical verification of speed-independent circuits. In *Proceedings of Second Working Conference on Asynchronous Design Methodologies, London*, pages 128–137. IEEE Computer Society, N.Y., May 1995.
103. T. G. Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.
104. L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *International Workshop on Timed Petri Nets, Torino, Italy*, 1985.
105. C. L. Seitz. Chapter 7. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems.* Addison Wesley, 1981.
106. A. Semenov. *Verification and Synthesis of Asynchronous Control Circuits using Petri Net Unfolding.* PhD thesis, University of Newcastle upon Tyne, Department of Computing Science, July 1997.
107. A. Semenov, A.M. Koelmans, L. Lloyd, and A. Yakovlev. Designing an asynchronous processor using Petri nets. *IEEE Micro*, 17(2):54–64, March 1997.
108. A. Semenov and A. Yakovlev. Event-Based Framework for Verifying High-Level Models of Asynchronous Circuits. Technical Report Technical Report Series, no. 487, Department of Computing Science, University of Newcastle upon Tyne, May 1994.
109. A. Semenov and A. Yakovlev. Contextual Net Unfolding and Asynchronous System Verification. Technical Report Technical Report Series, no. 572, Department of Computing Science, University of Newcastle upon Tyne, December 1996.
110. A. Semenov and A. Yakovlev. Verification of asynchoronous circuits using time petri net unfolding. In *Proceedings of ACM/IEEE Design Automation Conference (DAC96), Las Vegas*, pages 59–63, June 1996.
111. 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.

112. A. Semenov, A. Yakovlev, E. Pastor, M.A. Peña, and L. Lavagno. Partial order based approach to synthesis of speed-independent circuits. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 254 – 265, April 1997.
113. Semiconductor Industry Association. National Technology Roadmap for Semiconductors. (http://www.sematech.org/public/roadmap), 1994.
114. 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.
115. R.M. Shapiro. Validation of a VLSI chip using hierarchical colored Petri nets. In *International Conference on Application and Theory of Petri Nets, Paris, France*, pages 224–243, June 1990.
116. R.F. Sproull, I. Sutherland, and C.E. Molnar. The Counterflow Pipeline Processor Architecture. *IEEE Design & Test of Computers*, pages 48 – 59, Fall 1994.
117. N. Starodoubtsev, A. Yakovlev, and S. Petrov. Use of VHDL-based environment for interactive synthesis of asynchronous circuits. In *Proceedings VHDL Forum in Europe Spring'96 Working Conference, Dresden, Germany*, pages 21–34. Shaker Verlag, May 1996.
118. I. E. Sutherland. Micropipelines. *Communications of the ACM*, June 1989. Turing Award Lecture.
119. R.E. Swartwout. One method for designing speed-independent logic for a control. In *Proceedings of the Second AIEE Symposium on Switching Circuit Theory and Logical Design, Detroit, Michigan*, volume S-134, October 1961.
120. M. Tiusanen. Some unsolved problems in modelling self-timed circuits using Petri nets. *Bulletin of EATCS*, 36:152–160, October 1988.
121. UltraSan User's Manual. Technical report, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1994.
122. A. Valmari. Stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322, 1991.
123. K. van Berkel, J. Kessels, M. Roncken, R. Saejis, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of European Design Automation Conference*, pages 384 – 389, 1991.
124. J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1985.
125. P. Vanbekbergen. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *Proceedings of the International Conference on Computer-Aided Design*, pages 184–187, November 1990.
126. P. Vanbekbergen, A. Wand, and K. Keutzer. A design and validation system for asynchronous circuits. In *Proceedings of Design Automation Conference (DAC96), Las Vegas*, June 1995.
127. 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.
128. V. Varshavsky and V. Marakhovsky. Asynchronous control device design by net model behavour simulation. In *Lecture Notes in Computer Science, Vol. 1091: Proceedings of the 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka*, pages 497 – 515. Springer Verlag, June 1996.

129. V. Varshavsky and V. Marakhovsky. Hardware support of discrete event coordination. In *Proceedings of Int. Workshop on Discrete Event Systems (WODES'96), Edinburgh*, pages 332–340. IEE, August 1996.
130. A. Yakovlev. Designing control logic for counterflow pipeline processor using petri nets. *Formal Methods in System Design*, 1995. Accepted for publication.
131. A. Yakovlev. Solving ACiD-WG design problems with Petri net based methods. In *Proc. ESPRIT ACiD-WG Workshop on Asynchronous Circuit Design, Groningen*, September 1996 (TR CSN9602, Computer Science Notes Series, University of Groningen).
132. A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with OR causality. *Formal Methods in System Design*, 9:189–233, 1996.
133. A. Yakovlev, A.M. Koelmans, and L. Lavagno. High level modeling and design of asynchronous interface logic. Technical Report Series 460, University of Newcastle upon Tyne, Computing Science, November 1993.
134. 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.
135. A. Yakovlev and A. Petrov. Petri nets and parallel bus controller design. In *International Conference on Application and Theory of Petri Nets, Paris, France*, pages 244–263, June 1990.
136. 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.
137. A. V. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. *Formal Methods in System Design*, 9:139–188, 1996.
138. A. V. Yakovlev, A. I Petrov, and L. Lavagno. A low latency arbitration circuit. *IEEE Transactions on VLSI Systems*, pages 372–377, September 1994.
139. A.V. Yakovlev, A. M. Koelmans, A. Semenov, and D.J. Kinniment. Modelling, analysis and synthesis of asynchronous control circuits using Petri nets. *Integration: the VLSI journal*, 21:143–170, 1996.
140. C. Ykman-Couvreur, B. Lin, and H. De Man. ASSASSIN: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.
141. Ch. Ykman-Couvreur and B. Lin. Optimized state assignment for asynchronous circuit synthesis. In *Second Working Conference on Asynchronous Design Methodologies*, pages 118–127, May 1995.
142. M. Yoeli. Petri nets and asynchronous control networks. Technical Report Research Report CS-73-07, University of Waterloo, Department of Computer Science, April 1973.
143. A. Yu. The future of microprocessors. *IEEE Micro*, 16(6):46–53, December 1996.