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

This document contains only the initial part of the full paper: A.V. Yakovlev, A.M. Koelmans, Petri nets and Digital Hardware Design Lectures on Petri Nets II: Applications. Advances in Petri Nets, Lecture Notes in Computer Science, vol. 1492, Springer-Verlag, 1998, pp. 154-236.

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 [1, 81]. 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 [65]. 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 [33, 51, 42, 43]. Similarly, the theory and practice of digital system design has always recognised Petri nets as a powerful and easy-to-comprehend modelling tool [38, 56]. 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.

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

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 speed-independent circuits presented by D.E. Muller and W.C. Bartky [50, 46] 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 [24, 68], apparently the first "condition-event" approach to circuit specification, was formally related to Muller's state-based circuit classification [46]. 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 [32] 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 [48, 59, 22] and related formalisms, such as parallel flow graphs [17], 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 [8]. 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 [30, 31] 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 [80]. In parallel, new techniques for designing asynchronous control structures, very much in the style of Petri net based methods, had emerged [6]. Structural methods were studied and enhanced with additional modelling constructs and circuit components in the USSR-based work on aperiodic automata [2, 71], led by V.V. Varshavsky. In the UK, work on an asynchronous computer [49] and a design language called MUDL at Manchester University stimulated the use of Petri net models [34]. An interesting method for modelling and analysis of switching circuits with Petri nets was proposed in Germany [25]. Timed Petri nets were developed and applied [61] 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 [63], 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 [29, 54], 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 [69]. 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 [62, 37] and the USA [13, 11, 12], 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 [62]. The design-oriented links between Petri nets and self-time circuits were demonstrated in one of the most comprehensive monographs on asynchronous design [71].

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 [66]. Signal Transition Graphs and their close relative, Change Diagrams [36], 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 [57], symbolic traversal [58], and partial order (unfoldings) [45, 44]. 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 [28, 52]. 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 [21, 20, 18, 55, 15].

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* [63] 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 [3, 9], 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

[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.

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

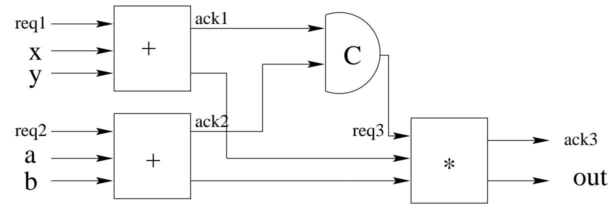
Physical level. Although we talk about "parallel programs" of the lowest possible level, this level is still a logical abstraction. Systems built from logic gates are themselves models of the real hardware, which behaves according to the laws of physics! Strictly speaking, in order to fully investigate the dynamic behaviour of switching processes in hardware, one should refer to the physical models of the circuits [63]. This can be done by means of systems of differential equations that describe a circuit as a dynamical system [5, 26]. 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. [7, 78]). 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 [39, 10]).

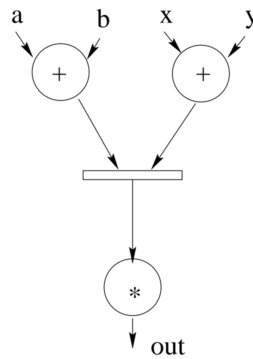
Logical level. At the logical level, the behaviour of an asynchronous circuit can be characterised by sequences of up and down transitions on the inputs and outputs of its components. The order between these transitions is not prescribed by any global scheduler or clock, and is determined by the *local* 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.

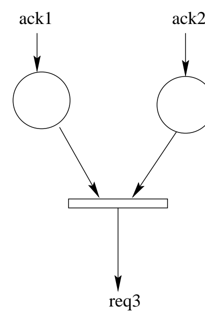
In addition, there is a *control* element called C, the *Muller C-element*, that controls the operation of the data path. In order to allow control of the data path, the adders and the multiplier have an extra input called 'req' (for *request*) and an extra output called 'ack' (for *acknowledgement*). 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.



(a)



(b)



(c)

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

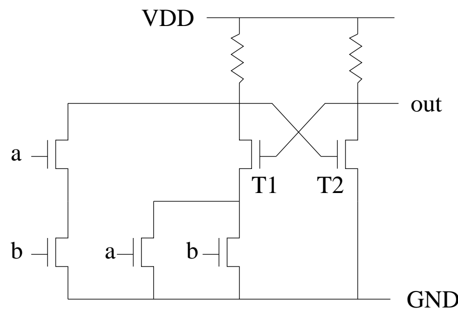


Fig. 2. Circuit implementation of the C-element

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 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 [78, 73].

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 [76, 40, 77], a tree arbiter [23, 79] or a modulo-n counter [19, 72], 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 [60, 35, 41, 53], 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 [71, 27]. 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 [78]. 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 [55], they provide a semantically rigorous bridge between other description languages and existing asynchronous circuit synthesis tools [15].

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.

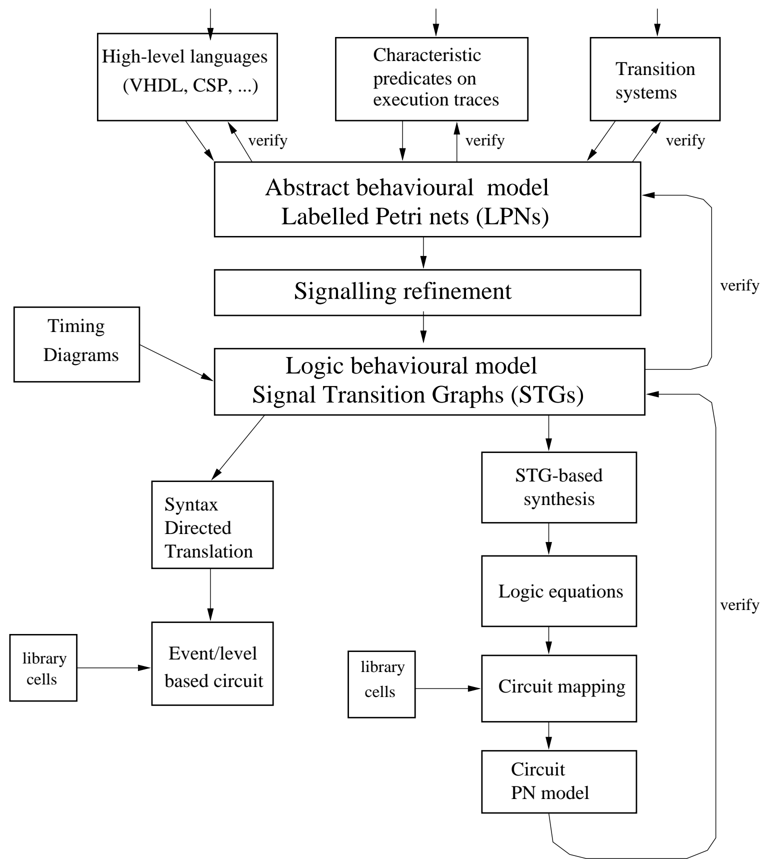


Fig. 3. Design flow of Petri net design system.

– 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.

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 [64].

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 hardware 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 con-*

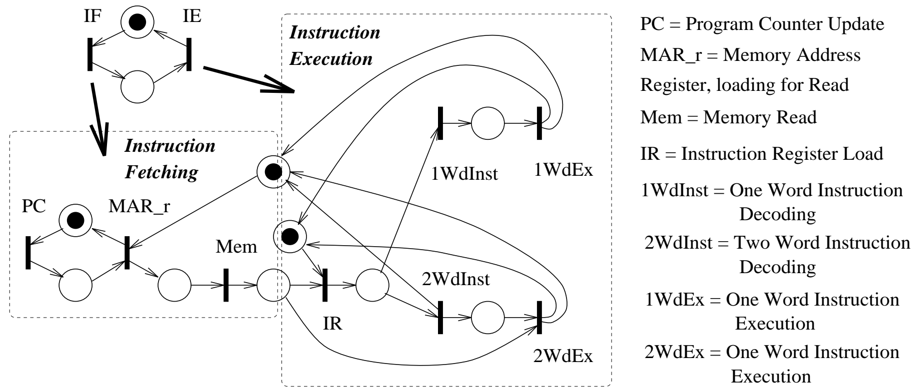


Fig. 4. Petri net example: a high level behavioural model of an asynchronous processor.

ditions to be implemented by the adapter circuit. The dashed arrows designate causal relations implemented by the environment, through the above-mentioned handshakes.

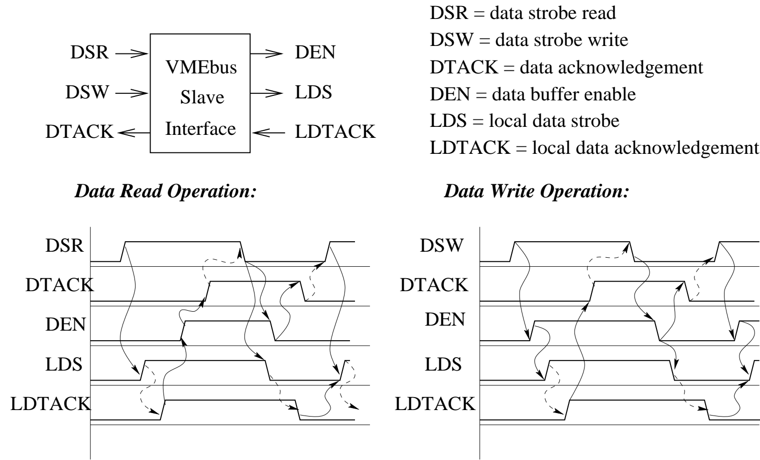


Fig. 5. VME bus adapter example: timing diagrams.

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.

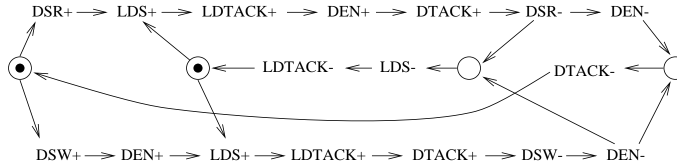


Fig. 6. VME bus adapter example: Signal Transition Graph.

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 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 $csc0$, inserted by Petrify for the purpose of appropriate state encoding (see Section **??**), is shown in Figure 7.

The process of constructing Signal Transition Graphs for this kind of hardware and synthesis of their logic implementations, is described in more detail in [76].

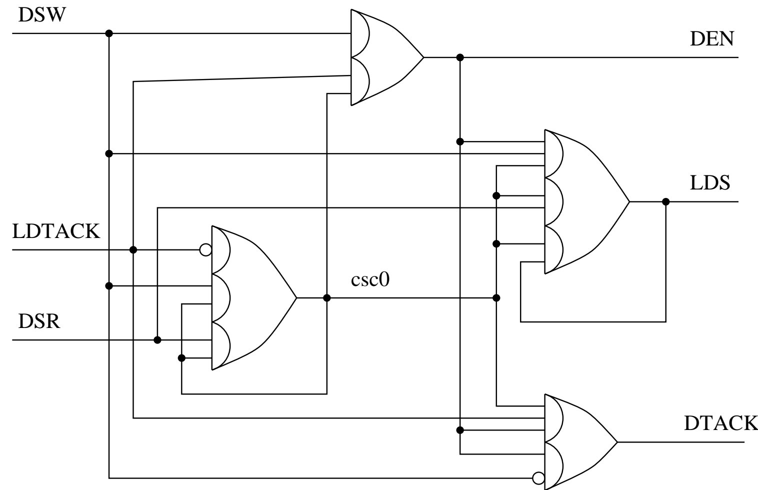
3 Overview of design transformations

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

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



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.

circuit. This means that these actions can be refined into so-called *active* handshakes [70]. 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.

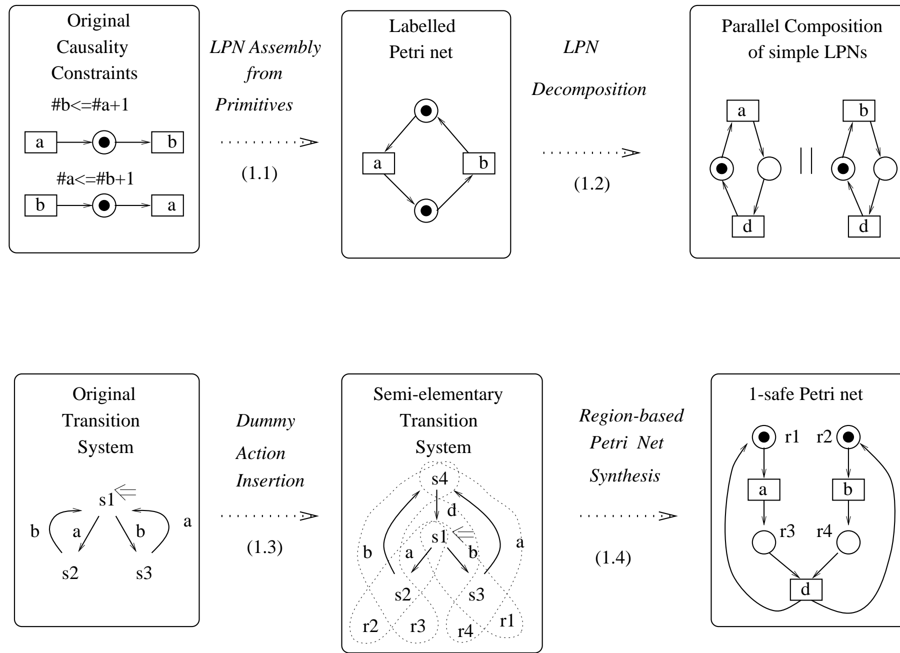


Fig. 8. Model transformations for abstract design.

Compositional approach (Section ??). 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* 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 syn-

chronisation 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* [47] 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 ??). 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 **??**. 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"[47]) 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 [55], 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.

[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.

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.

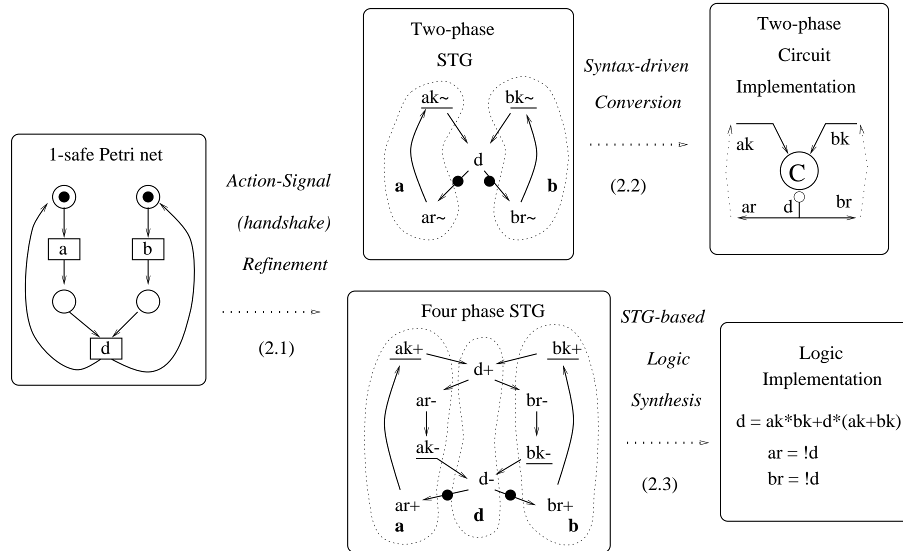


Fig. 9. Model transformations for logic design.

Action refinement (Section ??). 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 [74, 75]. 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 ??). 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 [67] or [59]. The class of 1-safe *simple* [51] Petri nets is sufficient to perform such a conversion [59]. 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]) [67], and (ii) for resolving conflicts with *output signal non-persistency*, by inserting *semaphore actions* which are implemented with arbitration elements [16]. 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 ??). 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 [16] 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 [65, 14], which themselves access the logic minimisation package Espresso [4]. 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.

References

1. Semiconductor Industry Association. National Technology Roadmap for Semiconductors. Available on Web (URL: http://www.sematech.org/public/roadmap),

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

[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.

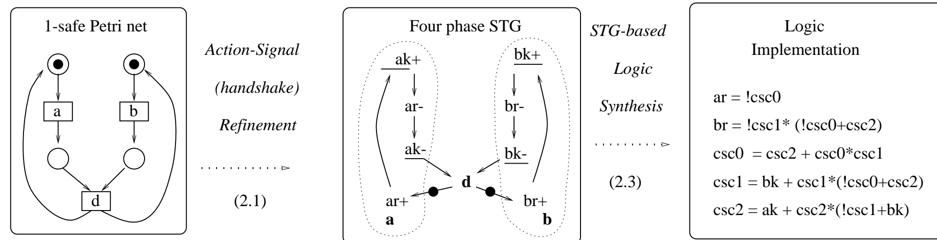


Fig. 10. Alternative four-phase STG refinement.

1994.

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. M. Ben Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall International, London, 1990.
4. R. Brayton et al. *Logic Minimisation Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Hingham, MA, 1984.
5. R.W. Brockett. Analog and digital computing. In *Lecture Notes in Computer Science, Vol. 653*, pages 279–289. Springer-Verlag.
6. J. Bruno and S. Altman. A theory of asynchronous control networks. *IEEE Transactions on Computers*, 20(6):629 – 638, June 1971.
7. 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.
8. 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.
9. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
10. 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.
11. T.-A. Chu. On the models for designing VLSI asynchronous digital systems. *Integration: the VLSI journal*, 4:99–113, 1986.
12. T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.
13. 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.
14. 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.
15. 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.

16. 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.
17. 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.
18. J. Desel and W. Reisig. The synthesis problem of Petri nets. Technical Report TUM-I9231, Technische Universität München, September 1992.
19. 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.
20. A. Ehrenfeucht and G. Rozenberg. A characterization of set representable labeled partial 2-structures through decompositions. *Acta Informatica*, 28:83–94, 1990.
21. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures. Part I, II. *Acta Informatica*, 27:315–368, 1990.
22. F.C. Furtek. Modular implementation of petri nets. Master's thesis, MIT, September 1971.
23. 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.
24. 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.
25. 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.
26. 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.
27. S. Hauck. Asynchronous Design Methodologies. *Proceedings of the IEEE*, 83(1), 1995.
28. H. Hulgaard and S.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.
29. R. Janicki and M. Koutny. On equivalent execution semantics of concurrent systems. In *Lecture Notes in Computer Science, Vol. 266*. Springer-Verlag, 1987.
30. J.R. Jump. Asynchronous control arrays. *IEEE Transactions on Computers*, TC-23(10):1020–1029, October 1974.
31. J.R. Jump and P.S. Thiagarajan. On the interconnection of asynchronous control structures. *Journal of ACM*, (4):596–612, October 1975.
32. R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, May 1969.
33. R.M. Keller. A fundamental theorem of asynchronous parallel computation. *Lecture Notes in Computer Science*, 24:103–112, 1975.
34. D.J. Kinniment. Regular programmable control structures. In *Proceedings VLSI-81 (Ed. by J.P. Gray), Edinburgh*, August 1981.
35. D.J. Kinniment. Evaluation of asynchronous adders. *IEEE Transactions on VLSI Systems*, 4(2), March 1996.
36. 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.

37. 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.
38. D. Lewin. *Design of Logic Systems*. Van Nostrand Reinhold (UK), 1985.
39. L.R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, C-30(2):107–115, February 1981.
40. A.J. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.
41. A.J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.
42. A. Mazurkiewicz. Concurrency, modularity and synchronization. In *Lecture Notes in Computer Science, Vol. 379*. Springer-Verlag, 1989.
43. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
44. 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.
45. K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. *Formal Methods in System Design*, 1995. (to appear).
46. R. E. Miller. *Switching theory*, volume 2, chapter 10, pages 192–244. Wiley and Sons, 1965.
47. R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, NJ, 1989.
48. D. Misunas. Petri Nets and speed-independent design. *Communications of the ACM*, pages 474–481, August 1973.
49. D. Morris and R.N. Ibbett. *The MU5 Computer System*. Macmillan Computer Science Series, 1979.
50. D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.
51. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.
52. C. Myers and T. H-Y. Meng. Synthesis of timed asynchronous circuits. In *Proceedings of the International Conference on Computer Design*, October 1992.
53. 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.
54. M. Nielsen, G.Plotkin, and G.Winskel. Petri nets, event structures and domains. Part 1. *Theoretical Computer Science*, 13:85–108, 1981.
55. M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96:3–33, 1992.
56. 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.
57. E. Pastor. *Structural Methods for the Synthesis of Asynchronous Circuits from Signal Transition Graphs*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona, 1996.
58. 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.

59. S. S. Patil and J. B. Dennis. The description and realization of digital systems. In *Proceedings of the IEEE COMPCON*, pages 223–226, 1972.
60. N.C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, University of Manchester, 1994.
61. C. Ramchandani. Analysis of asynchronous concurrent systems by Petri nets. Technical Report MAC-TR-120, MIT, Project MAC, February 1974.
62. 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.
63. C. L. Seitz. Chapter 7. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*. Addison Wesley, 1981.
64. 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.
65. 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.
66. 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.
67. I. E. Sutherland. Micropipelines. *Communications of the ACM*, June 1989. Turing Award Lecture.
68. 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.
69. M. Tiusanen. Some unsolved problems in modelling self-timed circuits using Petri nets. *Bulletin of EATCS*, 36:152–160, October 1988.
70. 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.
71. 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.
72. 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.
73. 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.
74. 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.
75. 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.
76. 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.

77. 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.
78. 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.
79. A. V. Yakovlev, A. I Petrov, and L. Lavagno. A low latency arbitration circuit. *IEEE Transactions on VLSI Systems*, pages 372–377, September 1994.
80. M. Yoeli. Petri nets and asynchronous control networks. Technical Report Research Report CS-73-07, University of Waterloo, Department of Computer Science, April 1973.
81. A. Yu. The future of microprocessors. *IEEE Micro*, 16(6):46–53, December 1996.

This article was processed using the LaTeX macro package with LLNCS style