

Synthesis of Asynchronous Hardware from Petri Nets

Josep Carmona¹, Jordi Cortadella¹, Victor Khomenko², and Alex Yakovlev²

¹ Universitat Politècnica de Catalunya, Barcelona, Spain
jcarmona@ac.upc.es, jordicf@lsi.upc.es

² University of Newcastle, Newcastle upon Tyne NE1 7RU, UK
{Victor.Khomenko,Alex.Yakovlev}@ncl.ac.uk

Abstract. As semiconductor technology strides towards billions of transistors on a single die, problems concerned with deep sub-micron process features and design productivity call for new approaches in the area of behavioural models. This paper focuses on some of recent developments and new opportunities for Petri nets in designing asynchronous circuits such as synthesis of asynchronous control circuits from large Petri nets generated from front-end specifications in hardware description languages. These new methods avoid using full reachability state space for logic synthesis. They include direct mapping of Petri nets to circuits, structural methods with linear programming, and synthesis from unfolding prefixes using SAT solvers.

1 Introduction

1.1 Semiconductor Technology Progress

The International Technology Roadmap for Semiconductors (ITRS) [1] predicts the end of this decade will be marked by the appearance of a System-on-a-Chip (SoC) containing four billion 50-nm transistors that will run at 10GHz. With a steady growth of about 60% in the number of transistors per chip per year, following the famous Moore's law, the functionality of a chip doubles every 1.5 to 2 years. Such a SoC will inevitably consist of many separately timed communicating domains, regardless of whether they are internally clocked or not [1]. Built at the deep sub-micron level, where the effective impact of interconnects on performance, power and reliability will continue to increase, such systems present a formidable challenge for design and test methods and tools.

The key point raised in the ITRS is that *design cost* is the greatest threat to the continued phenomenal progress in microelectronics. The only way to overcome this threat is through improving the productivity and efficiency of the design process, particularly by means of design automation and component reuse. The cost of design and verification of processing engines has reached the point where thousands of man-years are spent to a single design, yet processors reach the market with hundreds of bugs [1].

1.2 Self-timed Systems and Design Tools

Getting rid of global clocking in SoCs offers potential added values, traditionally quoted in the literature [60]: greater operational robustness, power savings, electro-magnetic compatibility and self-checking. While the asynchronous design community continues its battle for the demonstration of these features to the semiconductor industry investors, the issue of design productivity may suddenly turn the die to the right side for asynchronous design. Why?

One of the important sub-problems of the productivity and reuse problem for globally clocked systems is that of timing closure. This issue arises when the overall SoC is assembled from existing parts, called Intellectual Property (IP) cores, where each part has been designed separately (perhaps even by a different manufacturer) for a certain clock period, assuming that the clock signal is delivered accurately, at the same time, to all parts of the system. Finding the common clocking mode for SoCs that are built from multiple IP cores is a very difficult problem to resolve.

Self-timed systems, or less radical, globally asynchronous locally synchronous (GALS) systems [11, 70], are increasingly seen by industry as a natural way of composing systems from predesigned components without the necessity to solve the timing closure problem in its full complexity. As a consequence, self-timed systems highlight a promising route to solving the productivity problem as companies begin to realise. But they also begin to realise that without investing into design and verification tools for asynchronous design the above promise will not materialise. For example, Philips, whose products are critical to the time-to-market demands, is now the world leader in the exploitation of asynchronous design principles [27]. Other microelectronics giants such as Intel, Sun, IBM and Infineon, follow the trend and gradually allow some of their new products involve asynchronous parts. A smaller ‘market niche’ company Theseus Logic has been successful in down-streaming the results of their recent investment in asynchronous design methods (Null-Convention Logic) [26].

1.3 Design Flow Problem

The major obstacle now is the absence of a flexible and efficient design flow, which must be compatible with commercial CAD tools, such as for example the Cadence toolkit. A large part of such a design flow would be typically concerned with mapping the logic circuit (or sometimes macro-cell) netlist onto silicon area using place and route tools. Although hugely important this part is outside our present scope of interest, as it is essentially the same as in the traditional design flow. What we are concerned with is the stage in which the behavioural specification of a circuit is converted into the logic netlist implementation.

The pragmatic approach to this stage suggests that the specification should appear in the form of a high-level Hardware Description Language (HDL). Examples of such languages are the widely known VHDL and VERILOG, as well as TANGRAM [2] or Balsa [22] that are more specific for asynchronous design. The latter are based on the concepts of processes, channels and variables, similar to Hoare’s CSP.

We can in principle be motivated by the success of behavioural synthesis achieved by synchronous design in the 90s. However, for synchronous design the task of translating an HDL specification to logic (see, e.g., [47]) is fairly different from what we may expect in the asynchronous case.

Its first part was concerned with the so-called architectural synthesis, whose goal was the construction of a register-transfer level (RTL) description. This required extracting a control and data flow graph (CDFG) from the HDL, and performing scheduling and allocation of data operations to functional data path units in order to produce an FSM for a controller or sequencer. The FSM was then constructed using standard synchronous FSM synthesis, which generated combinational logic and rows of latches.

Although some parts of architectural synthesis, such as CDFG extraction, scheduling and allocation, might stay unchanged for self-timed circuits, the development of the intermediate level, an RTL model of a sequencer, and its subsequent circuit implementation, would be quite different.

1.4 How Can Petri Net Help?

Two critical questions arise at this point. Firstly, what is the most adequate formal language for the intermediate (still behavioural) level description? Secondly, what should be the procedure for deriving logic implementation from such a description?

The present level of development of asynchronous design flow suggests the following options to answer those questions:

(1) Avoid (!) answering them altogether. Instead, follow a syntax-driven translation of the HDL directly into a netlist of hardware components, called handshake circuits. This sort of silicon-compilation approach was pursued at Philips with the TANGRAM flow [2]. Many computationally hard problems involving global optimisation of logic were also avoided. Some local ‘peephole’ optimisation was introduced at the level of handshake circuit description. Petri nets were used for that in the form of Signal Transition Graphs (STGs) and their composition, with subsequent synthesis using the PETRIFY tool [52, 18]. Similar sort of approach is currently followed by the designers of the Balsa flow, where the role of peephole optimisation tools is played by the FSM-based synthesis tool MINIMALIST [12]. The problem with this approach is that, while being very attractive from the productivity point of view, it suffers from the lack of global optimisation, especially for high-speed requirements, because direct mapping of the parsing tree into a circuit structure may produce very slow control circuits.

(2) Translate the HDL specification into a STG for controller part and then synthesise this it using PETRIFY. This approach was employed in [4], where the HDL was VERILOG. This option was attractive because the translation of the VERILOG constructs preserved the natural semantical execution order between operations (not the syntax structure!) and PETRIFY could apply logic optimisation at a fairly global level. If the logic synthesis stage was not constrained by the state space explosion inherent in PETRIFY, this would have been an ideal situation.

However, the state space explosion becomes a real spanner in the works, because the capability of PETRIFY to solve the logic synthesis problem is limited by the number of logic signals in the specification. STGs involving 40–50 binary variables can take hours of CPU time. The size of the model is critical not only for logic minimisation but, more importantly, for solving state assignment and logic decomposition problems. The state assignment problem often arises when the STG specification is extracted automatically from an HDL. This forces PETRIFY into solving Complete State Coding (CSC) using computationally intensive procedures involving calculation of regions in the reachability graph.

While the logic synthesis powers of PETRIFY should not be underestimated, one should be realistic *where* they can be applied efficiently. Thus the solution lies where the design productivity similar to that of (1) can be achieved together with the circuit optimality offered by (2). We believe that the way to such a solution is through finding more efficient ways of logic synthesis in the framework of the design flow shown in Fig. 1.

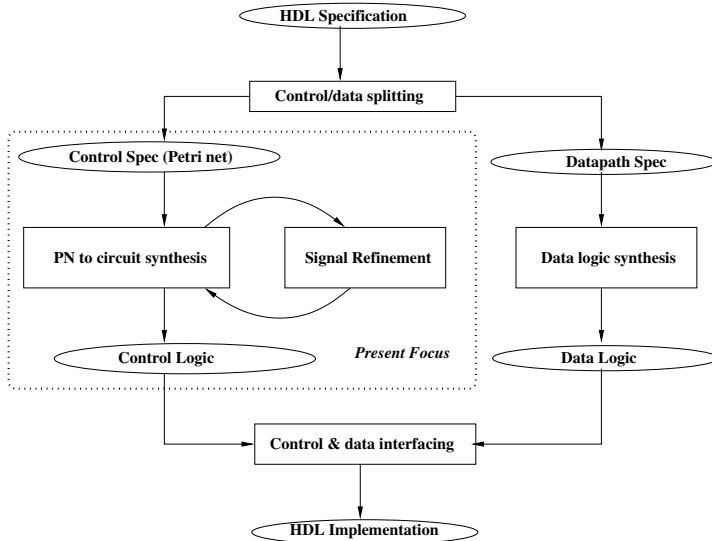


Fig. 1. Design Flow with Logic Synthesis from Petri nets.

The original HDL specification is syntactically and semantically analysed, giving rise to control and data path specifications. Data path can be synthesised using standard RTL-based (synchronous) design flow, applied to the main fragments of the data path, namely combinational logic and registers. There exist methods of converting such logic to self-timed implementations, e.g., [43]. This aspect of design is outside our scope here. The control specification is assumed to be extracted from the HDL in the form of a Petri net, which will thus act as the intermediate behavioural representation. Such an extraction is in gen-

eral non-trivial and relies on rigorous semantic relationship between control-flow constructs used in typical behavioural HDLs and their equivalents in Petri nets. For example, if one uses Balsa, such constructs basically include sequencing, parallelisation, two-way and multi-way selection, arbitration and (forever, while and for) loops, as well as macro and procedure calls. Those can be translated into Petri nets quite efficiently as done for example in PEP [3] for the translation of basic high-level programming language notation, $B(PN)^2$, into Petri nets.

1.5 Methods for Logic Synthesis from Petri Nets

The question of what kind of Petri nets is appropriate for subsequent logic synthesis of control depends on the method used for synthesis. Roughly, synthesis methods are split into two main categories. The first category comprises techniques of direct mapping of Petri net constructs to logic. In various forms it appeared in [51, 20, 32, 68, 74, 6, 58]. In the framework of 1-safe Petri nets and speed-independent circuits this problem was solved in [68], however only for autonomous (no inputs) specification where all operations were initiated by the control logic specified by a labelled Petri net. Another limitation was that the technique did not cover nets with arbitrary dynamic conflicts. Hollaar's one-hot encoding method [32] allowed explicit interfacing with the environment but required fundamental mode timing conditions, use of internal state variables as outputs and could not deal with conflicts and arbitration in the specifications. Patil's method [51] works for the whole class of 1-safe nets. However, it produces control circuits whose operation uses 2-phase (non-return-to-zero) signalling. This results in lower performance than what can be achieved for 4-phase circuits used in [68].

The second category considers the Signal Transition Graph refinement of the Petri net control specification. These methods usually perform an explicit logic synthesis, by deriving Boolean equations for the output signals of the controller using the notion of next state functions obtained from the STG [14, 18]. It should be noted that sometimes the STG specification for control can be obtained directly from the original specifications, e.g., if those are provided in the form of Timing Diagrams.

In this paper we will not concentrate on the problem of synthesis of Petri nets for logic synthesis of controllers and refer the reader to most recent literature, such as [4].

Our focus will be on the most recent advances in logic synthesis from Petri nets and Signal Transition Graphs. These methods try to avoid using the state space generated by the Petri net model directly. They follow two possible approaches. The first one, called a *structural* approach, performs graph-based transformations on the STG and deals with the approximated state space by means of linear algebraic representations. The second one, called an *unfolding-based* method, represents the state space in the form of true concurrency (or partial order) semantics provided by Petri net unfoldings.

The remaining structure of the paper is as follows. Section 2 introduces the problem of synthesis of control circuits from Petri net based specifications. It will

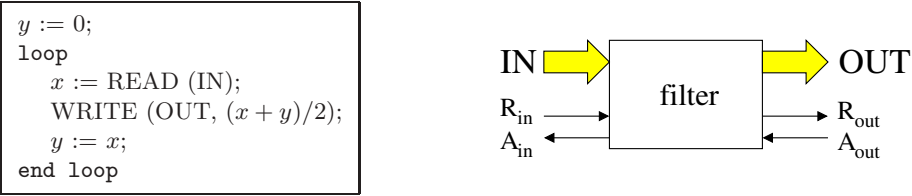


Fig. 2. High-level specification of a filter.

do it in an informal way by considering two characteristic examples of control logic to be designed by this sort of methodology. Section 3 provides an overview of the traditional state-based synthesis, which is currently implemented in the PETRIFY tool. Section 4 describes structural methods and use of integer linear programming in logic synthesis. Section 5 presents how Petri nets unfoldings and Boolean satisfiability problem (SAT) solvers can be used in the synthesis of asynchronous control logic. Section 6 briefly overviews some other related methodologies and outlines the important current and future research directions.

2 Synthesis Problem: Simple Examples and Signal Transition Graph Definition

We shall introduce the problem of synthesis of control circuits from Petri nets specifications using two simple but realistic design examples. This will also help us to present the two main types of control hardware that can be designed with the methods described in this paper. The first example, a simple data processing controller, will illustrate the design flow starting from an algorithmic, HDL-based, specification. The second one, an interface controller, will show the design starting from a waveform, Timing Diagram based, specification. Algorithmic and waveform specifications are most popular forms of behavioural notation amongst hardware designers. While describing the second example we will introduce our main specification model, Signal Transition Graph (STG).

2.1 A Simple Filter Controller

We illustrate a typical design flow by means of the example shown in Fig. 2. The algorithm describes a simple filter that reads data items from an input channel (IN) and writes the filtered data into an output channel (OUT) by averaging the last two samples, x and y . (Note that the first output value in this case may be invalid and should be ignored by the environment.) The interaction with the environment is asynchronous, using a four-phase protocol implemented by a pair of $\langle Request, Acknowledge \rangle$ signals, as shown in Fig. 3.

One of the possible implementations of the filter is depicted in the block diagram of Fig. 4. It contains two level-sensitive latches, x and y , and one adder (the averaging of x and y is achieved simply by a one-bit right shift of the bits

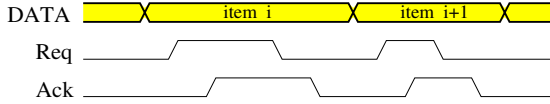


Fig. 3. Four-phase handshake protocol.

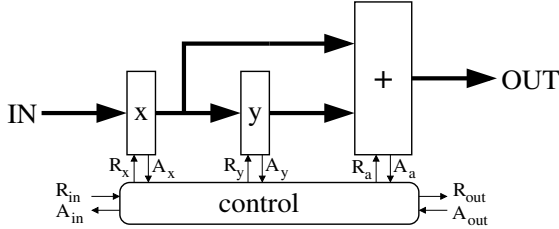


Fig. 4. Block diagram for the filter.

of the sum $x + y$). Each of the components operates according to a four-phase protocol as follows:

- The latches are transparent when R is high and opaque when low. A being high indicates that the data transfer through the latch has been completed.
- The adder starts its operation when R goes high. After a certain delay, signal A will be asserted, indicating that the addition has been finished and the output is valid. After that, R and A go low to complete the four-phase protocol.

The acknowledge signals of the latches and the adder can be implemented in many different ways, depending on how the blocks are designed. One way of doing that is by simply inserting a delay between R and A that mimics the worst-case delay of the corresponding block, as typically done for bundled-data components in micropipelines [64].

The signals $\langle R_{in}, A_{in} \rangle$ and $\langle R_{out}, A_{out} \rangle$ perform the synchronisation of the IN and OUT channels, respectively. R_{in} indicates the validity of IN. After A_{in} goes high, the environment is allowed to modify IN. On the other side, R_{out} and A_{out} should be able to control a level-sensitive latch in a similar way as described above for the latches x and y .

Synthesis of control. The synchronisation of the functional units depicted in Fig. 4 is performed by the *control* block, which is responsible for circulating the data items in the data-path in such a way that the required computations are performed as specified by the algorithm.

In this paper, we use a specially interpreted Petri nets, called Signal Transition Graphs (STGs), to specify the behaviour of asynchronous controllers. The transitions represent signal events (i.e., rising or falling edges of signals), whereas the arcs and places represent the causality relations among the events.

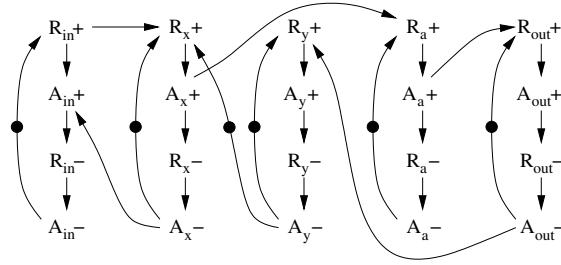


Fig. 5. Behavioural specification of the control.

Fig. 5 describes one possible behaviour of the control that results in a correct operation of the circuit. In this cases, the behaviour can be described by a *marked graph*, a subclass of Petri nets without choice. Marked graphs are often represented by omitting the places between transitions.

Each pair of req/ack signals commit a four-phase protocol, determined by the arcs $R^+ \rightarrow A^+ \rightarrow R^- \rightarrow A^- \rightarrow R^+$. The rest of the arcs are the ones that define how data items move along the data-path. For the sake of brevity, only a couple of them are discussed.

The arc $R_{in}^+ \rightarrow R_x^+$ indicates that the latch x can become transparent when there is some valid data at the IN channel. Moreover, the data can only be read once the latch y has captured the previous data from x . This is guaranteed by the arc $A_y^- \rightarrow R_x^+$.

On the other hand, the adder will start a new operation every time the latch x has acquired new data. This is indicated by the arc $A_x^+ \rightarrow R_a^+$. The result will be sent to the OUT channel when the addition has completed (arc $A_a^+ \rightarrow R_{out}^+$).

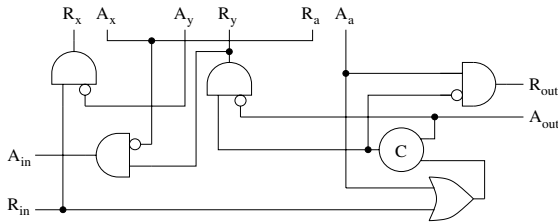


Fig. 6. Asynchronous controller for the filter.

From the specification of the control, a logic circuit can be synthesised. The circuit shown in Fig. 6 has been obtained by the PETRIFY tool.

2.2 VME Bus Controller

Our second example is a fragment of a VME bus slave interface [75]. It will help us to illustrate how the STG specification of an asynchronous controller

can be derived from its original Timing Diagram specification. Fig. 7(a) depicts the interface of a circuit that controls data transfers between a VME bus and a device. The main task of the bus controller is to open and close the data transceiver through signal d according to a given protocol to read/write data from/to the device.

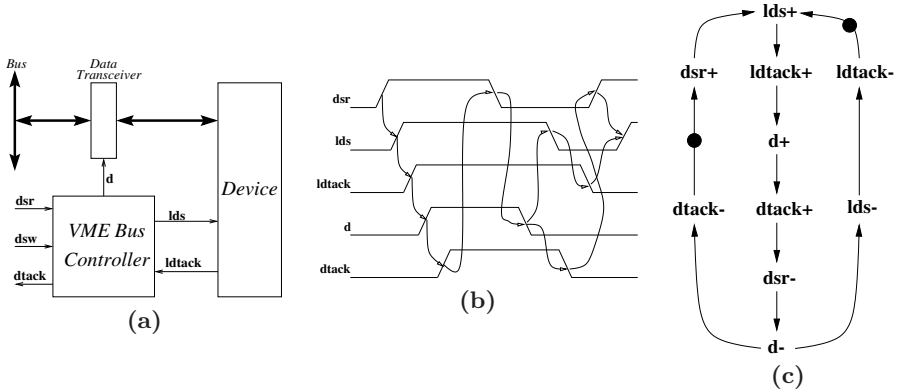


Fig. 7. VME bus controller: interface (a), the timing diagram for the read cycle (b) and the STG for the read cycle (c).

The input and output signals of the bus controller are as follows:

- dsr and dsw are input signals that request to do a read or write operation, respectively.
- $dtack$ is an output signal that indicates that the requested operation is ready to be performed.
- lds is an output signal to request the device to perform a data transfer.
- $ldtack$ is an input signal coming from the device indicating that the device is ready to perform the requested data transfer.
- d is an output signal that enables the data transceiver. When high, the data transceiver connects the device with the bus. The direction of the transfer (read or write) is defined by the high or low level of a special (RW) signal, which is part of the address/data bundle.

Fig. 7(b) shows a timing diagram of the read cycle. In this case, signal dsw is always low and not depicted in the diagram. The behaviour of the controller is as follows: a request to read from the device is received by signal dsr . The controller transfers this request to the device by asserting signal lds . When the device has the data ready ($ldtack$ high), the controller opens the transceiver to transfer data to the bus (d high). Once data has been transferred, dsr will become low indicating that the transaction must be finished. Immediately after, the controller will lower signal d to isolate the device from the bus. After that, the transaction will be completed by a return-to-zero of all interface signals, seeking for a maximum parallelism between the bus and the device operations.

Our controller also supports a write cycle with a slightly different behaviour. For the sake of simplicity, we have described in detail only the read cycle.

The model that will be used to specify asynchronous controllers is based on Petri nets [53, 49]. It is called *Signal Transition Graph* (STG) [55, 13]. Roughly speaking, an STG is a formal model for *timing diagrams*. Now we explain how to derive an STG from a timing diagram.

From timing diagrams to signal transition graphs. A timing diagram specifies the events (signal transitions) of a behaviour and their causality relations. An STG is a formal model for this type of specifications. In its simplest form, an STG can be considered as a causality graph in which each node represents an event and each arc a causality relation. An STG representing the behaviour of the read cycle for the VME bus is shown in Fig. 7(c). Rising and falling transitions of a signal are represented by the superscripts $+$ and $-$, respectively.

Additionally, an STG can also model all possible dynamic behaviours of the system. This is the rôle of the tokens held by some of the causality arcs. An event is *enabled* when it has at least one token on each input arc. An enabled event can *fire*, which means that the event occurs. When an event fires, a token is removed from each input arc and a token is put on each output arc. Thus, the firing of an event produces the enabling of another event. The tokens in the specification represent the initial state of the system.

The initial state in the specification of Fig. 7(c) is defined by the tokens on the arcs $dtack^- \rightarrow dsr^+$ and $ldtack^- \rightarrow lds^+$. In this state, there is only one event enabled, viz. dsr^+ . It is an event on an input signal that must be produced by the environment. The occurrence of dsr^+ removes a token from its input arc and puts a token on its output arc. In that state, the event lds^+ is enabled. In this case, it is an event on an output signal, that must be produced by the circuit modelled by this specification.

After firing the sequence of events $ldtack^+$, d^+ , $dtack^+$, dsr^- and d^- , two tokens are placed on the arcs $d^- \rightarrow dtack^-$ and $d^- \rightarrow lds^-$. In this situation, two events are enabled and can fire in any order independently from each other, i.e., these events are *concurrent*, which is naturally modelled by the STG.

Choice in signal transition graphs. In some cases, alternative behaviours, or modes, can occur depending on how the environment interacts with the system. In our example, the system will react differently depending on whether the environment issues a request to read or a request to write.

Typically, different behavioural modes are represented by different timing diagrams. For example, Fig. 8(a) and 8(b) depict the STGs corresponding to the read and write cycles, respectively. In these pictures, some arcs have been split and circles inserted in between. These circles represent *places* that can hold tokens. In fact, each arc going from one transition to another has an implicit place that holds the tokens located in that arc.

By looking at the initial markings, one can observe that the transition dsr^+ is enabled in the read cycle, whereas dsw^+ is enabled in the write cycle. The combination of both STGs models the fact that the environment can non-deterministically choose whether to start a read or a write cycle.

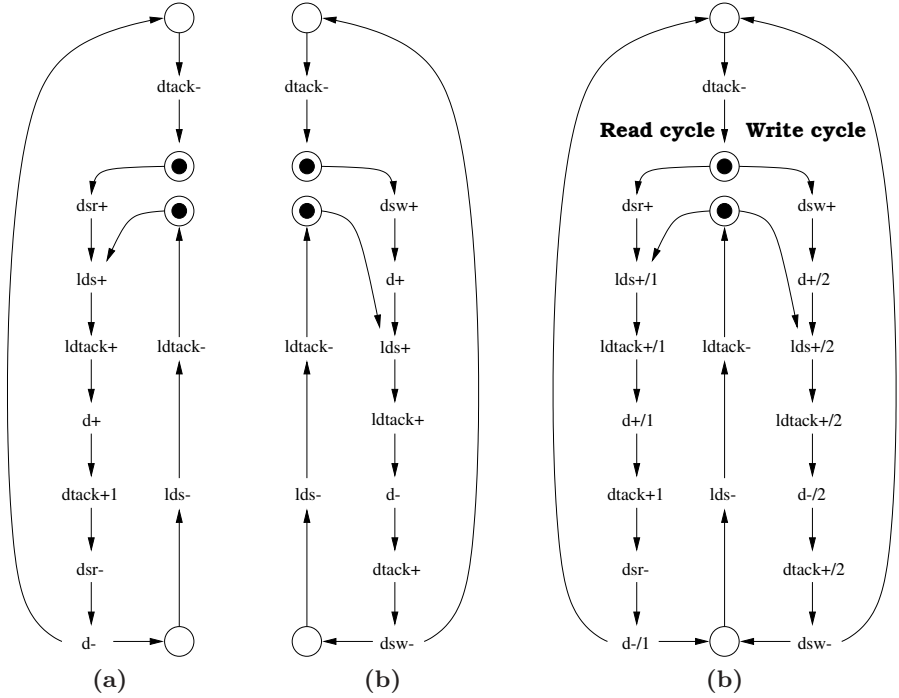


Fig. 8. VME bus controller: read cycle (a), write cycle (b), read and write cycles (c).

This combination can be expressed by a single STG with a choice place, as shown in Fig. 8(c). In the initial state, both transitions, dsr^+ and dsw^+ , are enabled. However, when one of them fires, the other is disabled since both transitions are competing for the token in the choice place. This type of choice is called *free choice* because the transitions, dsr^+ and dsw^+ , connected to the choice place have no other input places that could affect the process of choice making.

Here is where one can observe an important difference between the expressiveness of STGs and timing diagrams: the former are capable of expressing non-deterministic choices while the latter are not.

2.3 More Formal Definition of Signal Transition Graphs

To be able to introduce the methods of synthesis of asynchronous circuits in subsequent sections, we will need a more formal definition of an STG. STGs are a particular type of labelled Petri nets, where transitions are associated with the changes in the values of binary variables. These variables can for example be associated with wires, when modelling interfaces between blocks, or with input, output and internal signals in a control circuit.

A *net* is a triple $N \stackrel{\text{df}}{=} (P, T, F)$ such that P and T are disjoint sets of respectively *places* and *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. A

marking of N is a multiset M of places, i.e., $M : P \rightarrow \{0, 1, 2, \dots\}$. We adopt the standard rules about representing nets as directed graphs, viz. places are represented as circles, transitions as rectangles, the flow relation by arcs, and markings are shown by placing tokens within circles. As usual, $\bullet z \stackrel{\text{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \stackrel{\text{df}}{=} \{y \mid (z, y) \in F\}$ denote the *pre-* and *postset* of $z \in P \cup T$. We will assume that $\bullet t \neq \emptyset$, for every $t \in T$. A *net system* is a pair $\Sigma \stackrel{\text{df}}{=} (N, M_0)$ comprising a finite net N and an *initial* marking M_0 . We assume the reader is familiar with the standard notions of the theory of Petri nets, such as the *enabledness* and *firing* of a transition and marking *reachability*, as well as other standard notions and classification associated with Petri nets [49].

A *Signal Transition Graph (STG)* is a quadruple $\Gamma \stackrel{\text{df}}{=} (N, M_0, Z, \lambda)$, where

- $\Sigma = (N, M_0)$ is a Petri net (PN) based on a net $N = (P, T, F)$,
- Z is a finite set of binary signals, which generates a finite alphabet $Z^\pm = Z \times \{+, -\}$ of signal transitions
- $\lambda : T \rightarrow Z^\pm$ is a labelling function.

Labelling λ does not need to be 1-to-1 (some signal transitions may occur several times in the PN), and it may be extended to a partial function, in order to allow some transitions to be “dummy” ones (denoted by ϵ), that is to denote “silent events” that do not change the state of the circuit.

When talking about individual signal transitions, the following meaning will be associated with their labels. A label x^+ is used to denote the transition of signal x from 0 to 1 (rising edge), while x^- is used for a 1 to 0 transition (falling edge). In the following it will often be convenient to associate STG transitions directly with their labels, “bypassing” their Petri net identity. In such cases if the labelling is not 1-to-1 (so called multiple labelling), we will also use a subscript or an index separated by slash denoting the instance number of the x^\pm .

Sometimes, when reasoning on a pure event-based level, it will also be convenient to hide the direction of a particular edge and use x^\pm to denote either a x^+ transition or an x^- transition.

An STG inherits the basic operational semantics from the behaviour of its underlying Petri net. In particular, this includes: (i) the rules for transition enabling and firing, (ii) the notions of reachable markings, traces, and (iii) the temporal relations between transitions (precedence, concurrency, choice and conflict). Likewise, STGs also inherit the various structural (marked graph, free-choice, etc.) and behavioural properties (boundedness, liveness, persistency, etc.), and the corresponding classification of PNs. Namely:

- **Choice place.** A place is called a choice (or conflict) place if it has more than one output transition.
- **Marked graph and State machine.** A PN is called a *marked graph* (MG) if each place has exactly one input and one output transition. Dually, a PN is called a *state machine* (SM) if each transition has exactly one input and one output place. MGs have no choice. Safe SMs have no concurrency.

- **Free-choice.** A choice place is called *free-choice* if every its output transition has only one input place. A PN is *free-choice* if all its choice places are free-choice.
- **Persistency.** A transition $t \in T$ is called *non-persistent* if some reachable marking enables t together with another transition t' , and t becomes disabled after firing t' . Non-persistency of t with respect to t' is also called a *direct conflict* between t and t' . A PN is *persistent* if it does not contain any non-persistent transition.
- **Boundedness and safeness.** A PN *k-bounded* if for every reachable marking the number of tokens in any place is not greater than k (a place is called *k-bounded* if for every reachable marking the number of tokens in it is not greater than k). A PN is *bounded*, if there is a finite k for which it is *k-bounded*. A PN is *safe* if it is 1-bounded (a 1-bounded place is called a safe place).
- **Liveness.** A PN is *live* if for every transition t and every reachable marking M there is a firing sequence that leads to a marking M' enabling t .

The signal transition labelling of an STG may sometimes differentiate between input and non-input signals, thus forming two disjoint subsets, Z_I (for inputs) and Z_O (for non-inputs, or simply outputs), such that $Z = Z_I \cup Z_O$. An STG is called *autonomous* if it has no input signals (i.e., $Z_I = \emptyset$).

Graphically, an STG can either be represented in the standard form of a labelled PN, drawing transitions as bars or boxes and places as circles, or in the so-called STG shorthand form. The latter, as was first shown in the above examples, designates transitions directly by their labels and omits places that have only one input and one output transition.

Examples of STGs, in their shorthand notation, were shown in Fig. 8, describing a simple VME bus controller example. It was assumed in them that $Z_I = \{dsr, dsw, ldtack\}$ and $Z_O = \{lds, dtack, d\}$. The first two STGs, in Fig. 8(a) and 8(b), are marked graphs (they do not have choice on places). The third one, in Fig. 8(c), modelling both read and write operation cycles, is not a marked graph because it contains places with multiple input and output transitions. It is not a free-choice net either, because one of its choice places, the input to transitions $lds^+/1$ and $lds^+/2$, is not a free-choice place. The latter is however a *unique choice place* because whenever one of the above two transitions is enabled the other is not, which is guaranteed by the other choice place, which is a free-choice one. Thus, behaviourally, this net does not lead to dynamic conflicts (arbitration) or confusion, as it is free from any interference between choice and concurrency.

3 State-Based Synthesis from Signal Transition Graphs

The main purpose of this section is to present a state-based method to design asynchronous control circuits, i.e., those circuits that synchronise the operations performed by the functional units of the data-path through handshake protocols. The method uses the STG model of a circuit as its initial specification. The key

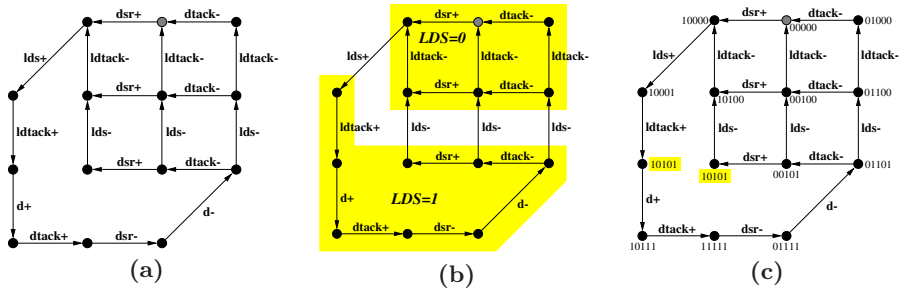


Fig. 9. Reachability graph of read cycle (a), its binary partitioning for signal *lds* (b), and the encodings of the reachable states (c). The order of signals in the binary encodings is: *dsr*, *dtack*, *ldtack*, *d*, *lds*.

steps in this method are the generation of a state graph, which is a binary encoded reachability graph of the underlying Petri net, and deriving Boolean equations for the output signals via their next state functions obtained from the state-graph. This method is surveyed here very briefly and informally, using our VME bus controller example. For more details the reader is referred to the book and the PETRIFY tool [18].

3.1 State Graphs

State space. An STG is a succinct representation of the behaviour of an asynchronous control circuit that describes the causality relations among the events. However, the state space of the system must be derived by exploring all possible firing orders of the events. Such exploration may result in a state space much larger than the specification.

Unfortunately, the synthesis of asynchronous circuits from STGs requires an exhaustive exploration of the state space. Finding efficient representations of the state space is a crucial aspect in building synthesis tools. Other techniques based on direct translation of Petri Nets into circuits or on approximations of the state space exist [42, 50], but usually produce circuits with area and performance penalty.

Going back to our example of the VME bus controller, Fig. 9(a) shows the reachability graph corresponding to the behaviour of the read cycle. The initial state is depicted in gray.

For simplicity, the write cycle will be ignored in the rest of this section. Thus, we will consider the synthesis of a bus controller that only performs read cycles.

Binary interpretation. The events of an asynchronous circuit are interpreted as rising and falling transitions of digital signals. A rising (falling) transition represents a switch from 0 (1) to 1 (0) of the signal value. Therefore, when considering each signal of the system, a binary value can be assigned to each state for that signal. All those states visited after a rising (falling) transition

and before a falling (rising) transition represent situations in which the signal value is 1 (0).

In general, the events representing rising and falling transitions of a signal induce a partition of the state space. As an example, let us take signal lds of the bus controller. Fig. 9(b) depicts the partition of states. Each transition from $LDS=0$ to $LDS=1$ is labelled by lds^+ and each transition from $LDS=1$ to $LDS=0$ is labelled by lds^- .

It is important to notice that rising and falling transitions of a signal must alternate. The fact that a rising transition of a signal is enabled when the signal is at 1 is considered a specification error. More formally, a specification with such problem is said to have an *inconsistent state coding*.

After deriving the value of each signal, each state can be assigned a binary vector that represents the value of all signals in that state. A transition system with a binary interpretation of its signals is called a *state graph* (SG). The SG of the bus controller read cycle is shown in Fig. 9(c).

3.2 Deriving Logic Equations

In this section we explain how an asynchronous circuit can be automatically obtained from a behavioural description. We have already distinguished two types of signals in a specification: inputs and outputs. Further, some of the outputs may be observable and some internal. Typically, observable outputs correspond to those included in the specification, whereas internal outputs correspond to those inserted during synthesis and not observable by the environment. Synthesising a circuit means providing an implementation for the output signals of the system.

This section gives an overview of the methods used for the synthesis of asynchronous circuits from an SG.

System behaviour. The specification of a system models a protocol between its inputs and outputs. At a given state, one or several of these two situations may happen:

- The system is waiting for an input event to occur. For example, in the state 00000 of Fig. 9(c), the system is waiting for the environment to produce a rising transition on signal $dscr$.
- The system is expected to produce a non-input (output or internal) event. For example, the environment is expecting the system to produce a rising transition on signal lds in state 10000.

In concurrent systems, several of these things may occur simultaneously. For example, in state 00101, the system is expecting the environment to produce $dscr^+$, whereas the environment is expecting the system to produce lds^- . In some other cases, such as in state 01101, the environment may be expecting the system to produce several events concurrently, e.g., $dtack^-$ and lds^- .

The particular order in which concurrent events will occur will depend on the delays of the components of the system. Most of the synthesis methods discussed

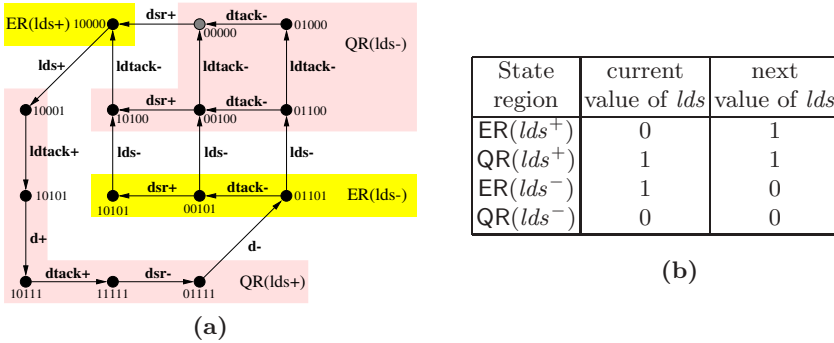


Fig. 10. Excitation and quiescent regions for signal lds (a) and the corresponding next-state function (b).

here aim at synthesising circuits whose correctness does not depend on the actual delays of the components. These circuits are called *speed-independent*.

A correct implementation of the output signals must be in such a way that signal transitions on those signals must be generated *if and only if* the environment is expecting them. Unexpected signal transitions, or not generating signal transitions when expected, may produce circuit malfunctions.

Excitation and Quiescent Regions. Let us take one of the output signals of the system, say lds . According to the specification, the states can be classified into four regions:

- The *positive excitation region*, $ER(lds^+)$, includes all those states in which a rising transition of lds is enabled.
- The *negative excitation region*, $ER(lds^-)$, includes all those states in which a falling transition of lds is enabled.
- The *positive quiescent region*, $QR(lds^+)$, includes all those states in which signal lds is at 1 and lds^- is not enabled.
- The *negative quiescent region*, $QR(lds^-)$, includes all those states in which signal lds is at 0 and lds^+ is not enabled.

Fig. 10(a) depicts these regions for signal lds . It can be easily deduced that $ER(lds^+) \cup QR(lds^-)$ and $ER(lds^-) \cup QR(lds^+)$ are the sets of states in which signal lds is at 0 and 1, respectively.

Next-State Functions. Excitation and quiescent regions represent sets of states that are behaviourally equivalent from the point of view of the signal for which they are defined. The semantics of these regions are the following:

- $ER(lds^+)$ is the set of states in which lds is at 0 and the system must change it to 1.
- $ER(lds^-)$ is the set of states in which lds is at 1 and the system must change it to 0.

- $QR(lds^+)$ is the set of states in which lds is at 1 and the system must not change it.
- $QR(lds^-)$ is the set of states in which lds is at 0 and the system must not change it.

According to this definition, the behaviour of each signal can be determined by calculating the *next value* expected at each state of the SG. This behaviour can be modelled by Boolean equations that implement the so-called *next-state* functions (see Fig. 10(b)).

Let us consider again the bus controller and try to derive a Boolean equation for the output signal lds . A 5-variable Karnaugh map for Boolean minimisation is depicted in Fig. 11. Several things can be observed in that table. There are many cells of the map with a *don't care* (–) value. These cells represent binary encodings not associated to any of the states of the SG. Since the system will never reach a state with those encodings, the next-state value of the signal is irrelevant.

dsr,dtack		lds=0			
		00	01	11	10
ldtack,d	00	0	0	-	1
	01	-	-	-	-
	11	-	-	-	-
	10	0	0	-	0

dsr,dtack		lds=1			
		00	01	11	10
ldtack,d	00	-	-	-	1
	01	-	-	-	-
	11	-	1	1	1
	10	0	0	-	1

CSC conflict →

Fig. 11. Karnaugh map for the minimisation of signal lds .

The shadowed cells correspond to states in the excitation regions of the signal. The rest of cells correspond to states in some of the quiescent regions. If we call f_{lds} the next-state function for signal lds , here are some examples on the value of f_{lds} :

$$\begin{array}{ll}
 f_{lds}(10000) = 1 & \text{state in } ER(lds^+) \\
 f_{lds}(10111) = 1 & \text{state in } QR(lds^+) \\
 f_{lds}(00101) = 0 & \text{state in } ER(lds^-) \\
 f_{lds}(01000) = 0 & \text{state in } QR(lds^-)
 \end{array}$$

3.3 State Encoding

At this point, the reader must have noticed a peculiar situation for the value of the next-state function for signal lds in two states with the same binary encoding: 10101. This binary encoding is assigned to the shadowed states in Fig. 9(c).

Unfortunately, the two states belong to two different regions for signal lds , namely to $ER(lds^-)$ and $QR(lds^+)$. This means that the binary encoding of

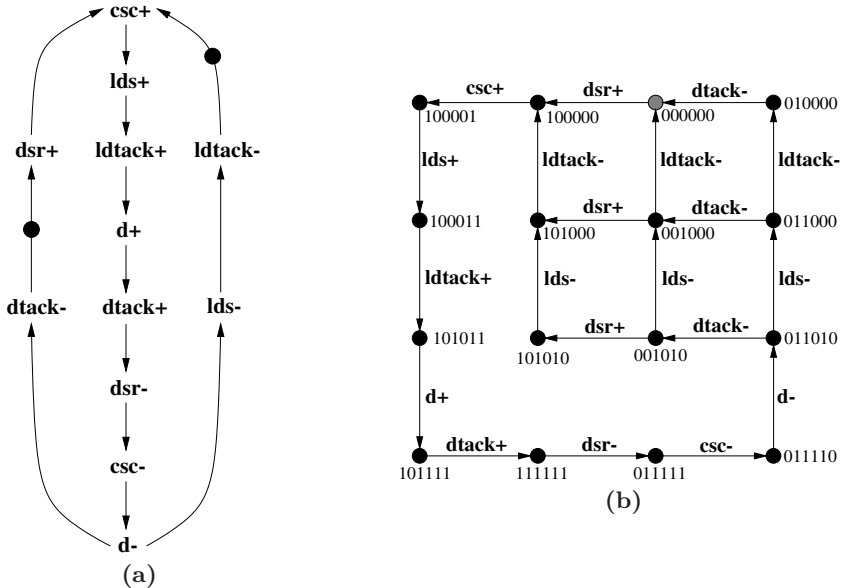


Fig. 12. An STG (a) and its SG (b) satisfying the CSC property.

the SG signals alone cannot determine the future behaviour of *lds*. Hence, an ambiguity arises when trying to define the next-state function. This ambiguity is illustrated in the Karnaugh map of Fig. 11.

Roughly speaking, this phenomenon appears when the system does not have enough memory to “remember” in which state it is. When this occurs, the system is said to violate the *Complete State Coding* (CSC) property. Enforcing CSC is one of the most difficult problems in the synthesis of asynchronous circuits.

Fig. 12 presents a possible solution for the SG of the VME bus controller. It consists of inserting a new signal, *csc*, that adds more memory to the system. After the insertion, the two conflicting states are disambiguated by the value of *csc*, which is the last value in the binary vectors of Fig. 12.

Now Boolean minimisation can be performed and logic equations can be obtained (see Fig. 13). In the context of Boolean equations representing gates we shall liberally use the “=” sign to denote “assignment”, rather than mathematical equality. Hence *csc* on the left-hand side of the last equation stands for the *next* value of signal *csc*, while *csc* on the right-hand side corresponds to its current value. The resulting circuit contains cycles: the combinational feedbacks play the rôle of local memory in the system.

The circuit shown in Fig. 13 is said to be speed-independent, i.e., it works correctly regardless of the delays of its components. For this to be true, it is required that each Boolean equation is implemented as one *complex gate*. This roughly means that the internal delays within each gate are negligible and do not produce any externally observable spurious behaviour. However, the external delay of the gates can be arbitrarily long.

$$\begin{aligned}
lds &= d + csc \\
dtack &= d \\
d &= ldtack \cdot csc \\
csc &= dsr \cdot (csc + \overline{ldtack})
\end{aligned}$$

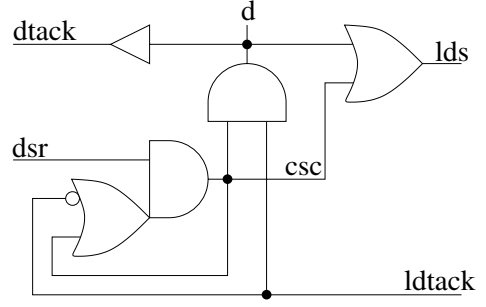


Fig. 13. Logic equations and implementation of the VME bus controller.

Note that signal *dtack* is merely implemented as a buffer, and a wire is enough to preserve that behaviour. But note that the specification indicates that the transitions of *dtack* must occur always *after* the transitions of *d*. For this reason, the resulting equation is $dtack = d$ and not vice versa. Thus, the buffer introduces the required delay to enforce the specified causality.

3.4 Properties for Implementability

In conclusion to this section let us summarise the main properties required for the STG specification to be implementable as a speed-independent circuit [18]:

- *Boundedness* of the STG that guarantees the SG to be finite.
- *Consistency* of the STG, that ensures that the rising and falling transitions of each signal alternate in all possible runs of the specification.
- *Completeness of state encoding* (CSC) that ensures that there are no two different states with the same signal encoding but different behaviour of the output or internal signals.
- *Persistency* of signal transitions in such a way that no signal transition can be disabled by another signal transition, unless both signals are inputs. This property ensures that no short glitches, known as *hazards*, will appear at the disabled signals. (Arbitration is implemented by ‘factoring out’ the arbiter into the environment and using a special circuit able to resolve meta-stability.)

4 Synthesis Using Structural Methods, Linear Programming and STG Decomposition

4.1 Rationale

Structural methods provide a way to avoid the state space explosion problem, given that they rely on succinct representations of the state space. The main benefit of using structural methods is the ability to deal with large and highly concurrent specifications, that cannot be tackled by state-based methods. On the

other hand, structural methods are usually conservative and approximate, and can only be exact when the behaviour of the specifications is restricted in some sense. For instance, in [66] structural methods for the synthesis of asynchronous circuits are presented for the class of *marked graphs*, a very restricted class of Petri nets where choices are not allowed. In this section we present structural methods to solve some of the main problems in the synthesis of asynchronous control circuits from well-formed specifications.

As it was explained in previous sections, the synthesis of asynchronous circuits from an STG can be separated into two steps [18]: (i) checking and (possibly) enforcing implementability conditions and (ii) deriving the next-state function for each signal generated by the system. Most of the existing CAD tools for synthesis perform steps (i) and (ii) at the underlying state graph level, thus suffering from the state space explosion problem.

In order to avoid the state explosion problem, structural methods for steps (i) and (ii) have been proposed in the literature. Approaches like the ones presented in [66, 50, 9, 8] can be considered purely structural. Among the methods applied by these approaches, graph theoretic-based and linear algebraic are the essential techniques. The work presented in this section uses both linear algebraic methods and graph theoretic-based methods.

Regarding step (i), in this section an encoding technique to ensure implementability is presented. It is inspired by the work of René David [20]. The main idea is to insert a new set of signals in the initial specification in a way that unique encoding is guaranteed in the transformed specification.

To the best of our knowledge, the results reported in [38, 29] are the first ones that use linear algebraic techniques to approach the encoding problem. In the former approach, a complete characterisation of the encoding problem is presented, provided that, like in Section 5, unfoldings are used to represent the underlying state space of the net. Linear algebraic methods to verify the encoding are presented in this section, where the computation of the unfolding is not performed, at the expense of checking only sufficient conditions for synthesis. However, the experimental results indicate that this approach is highly accurate and often provides a significant speed-up compared with [38, 39]. One can imagine a design flow where the methods presented in this section are used to pre-process the specifications, and complete methods like the ones presented in Section 5 are only used when purely structural methods fail.

Another alternative to alleviate the state space explosion problem is by using decomposition techniques. We apply them when performing step (ii). More specifically, in this section an algorithm for computing the set of signals needed to synthesise a given signal is presented, which also uses linear algebraic techniques. This allows to project the behaviour into that set of signals and perform the synthesis on the projection.

In summary, this section covers the two important steps (i) and (ii) in the synthesis of asynchronous circuits: it proposes powerful methods for checking CSC/USC and a method for decomposing the specification into smaller ones *while preserving the implementability conditions*.

4.2 Structural Technique to Ensure a Correct Encoding

The first example of use of structural methods is presented in this section. The technique is inspired by previous work on using a special type of cells, called *David cells*. This type of cells, first introduced in [20], were used in [69] to mimic the token flow of a Petri net. Fig. 14 depicts a very simple example on how these cells can be abutted to build a distributor that controls the propagation of activities along a ring.

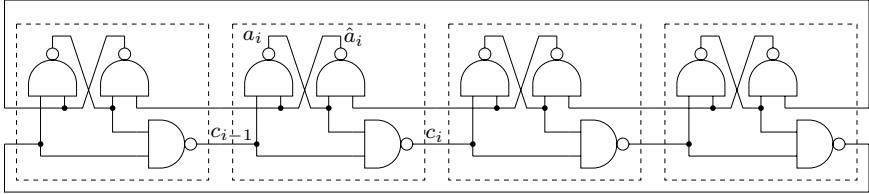


Fig. 14. Distributor built from David cells [42].

The behaviour of one of the cells in the distributor can be summarised by the following sequence of events:

$$\begin{aligned}
 & \cdots \rightarrow \underbrace{c_{i-1}^-}_{i\text{-th cell excitation}} \rightarrow \underbrace{a_i^+ \rightarrow \hat{a}_i^-}_{i\text{-th cell setting}} \rightarrow \\
 & \rightarrow \underbrace{\hat{a}_{i-1}^+ \rightarrow a_{i-1}^- \rightarrow c_{i-1}^+}_{(i-1)\text{-th cell resetting}} \rightarrow \underbrace{c_i^-}_{(i+1)\text{-th cell excitation}} \rightarrow \cdots
 \end{aligned}$$

Let us explain how one can use David Cells to ensure a correct encoding of the system specified by a given STG. The main idea is to add a new signal for each place of the original net. The semantics of the new signal inserted is to mimic the token flow of the corresponding place in the original net. The technique is shown in Fig. 15.

For instance place p_4 induces the creation of signal sp_4 . Moreover, provided that when transition $dtack^+$ is enabled, it adds a token to p_4 , in the transformed net it will induce that near (preceding) $dtack^+$ there must be an sp_4^+ . A similar reason makes to have sp_4^- near (following) $dscr^-$. New transitions are inserted in a special way: internal signal transitions must not be inserted in front of an input signal transition. The reason for that is to try to preserve the I/O interface (see more on this in [7]).

The derived STG is guaranteed to have a correct encoding (in the example, the right STG is guaranteed to satisfy the USC property). The theory underlying the technique can be found in [9]. It can be applied for any STG with the underlying free-choice live and safe Petri net (FCLSPN).

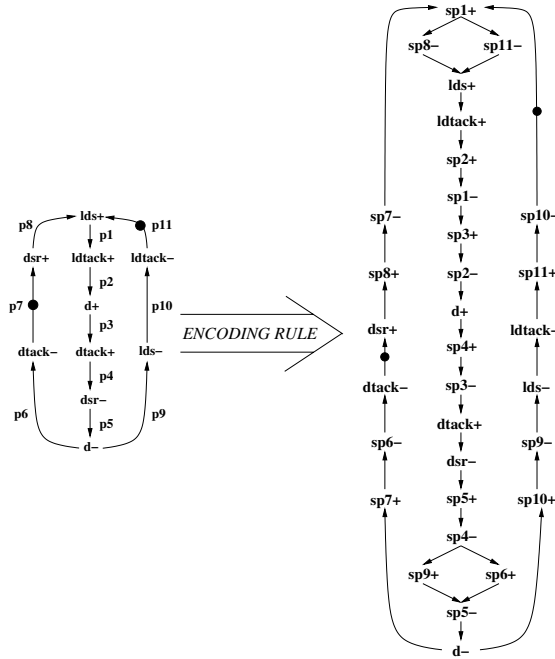


Fig. 15. Encoding rule applied to the VME Bus Controller example.

4.3 ILP Models for Fast Encoding Verification

The main drawback of this technique is that a correct encoding is ensured at the expense of inserting a lot of new signals into the net, and thus the final implementation can be very inefficient in terms of area and/or performance. Therefore it would be nice to have an *oracle* that could tell us when the application of the encoding technique is needed, provided that the computation of the state space and subsequent checking cannot be done for the specification at hand due to efficiency reasons.

This section presents Integer Linear Programming (ILP) models as oracles that we can use to verify the encoding in an STG. The good news is that when we query such oracles, usually it takes short time for them to answer, even for very large STGs. The bad news is that they are not perfect oracles: we can only trust them when they say “Yes, your STG is correctly encoded”.

In this section we assume some basic knowledge of *Linear Programming* (see, e.g., [56]). The rest of this section has three main parts: first it is shown how to use linear algebraic techniques for deciding whether a given marking M is reachable in a Petri net. Second, using this technique, models for finding encoding conflicts are presented, and third, experimental results are shown.

Approximation of the reachability set of a PN. Computing the reachability graph from a given PN is a very hard problem, because the size of the reachability graph

may grow exponentially with respect to the size of the PN, or it even can be infinite. The main reason is that the concurrency in the PN leads to a blow up in the reachability graph. The reader can find in [65] an in-depth discussion on the rôle of concurrency in relation to the size of the reachability graph.

Therefore, it is interesting to approach the problem of reachability using other models or techniques. In this section we describe how to use ILP techniques to compute approximations of reachable markings of a PN.

Given a firing sequence $M_0 \xrightarrow{\sigma} M$ of a PN N , the number of tokens for each place p in M is equal to the number of tokens of p in M_0 plus the number of tokens added by the input transitions of p appearing in σ minus the tokens removed by the output transitions of p appearing in σ , which can be expressed as the following *token conservation equation*:

$$M(p) = M_0(p) + \sum_{t \in \bullet p} \#(\sigma, t) F(t, p) - \sum_{t \in p \bullet} \#(\sigma, t) F(p, t) .$$

Definition 1 (Incidence matrix of a PN). *The matrix $\mathbf{N} \in \{-1, 0, 1\}^{|P| \times |T|}$ defined by $\mathbf{N}(p, t) \stackrel{\text{df}}{=} F(p, t) - F(t, p)$ is called the incidence matrix of N .*

Definition 2 (Parikh vector). *Let σ be a feasible sequence of N . The vector $\sigma \stackrel{\text{df}}{=} (\#(\sigma, t_1), \dots, \#(\sigma, t_n))$ is called the Parikh vector of σ .*

Using the previous definitions, the token conservation equations for all the places in the net can be written in the following matrix form:

$$M = M_0 + \mathbf{N} \cdot \sigma .$$

This equation allows to approximate the reachability set of a Petri net by means of an ILP:

Definition 3 (Marking Equation). *If a marking M is reachable from M_0 , then there exists a sequence σ such that $M_0 \xrightarrow{\sigma} M$, and the marking equation*

$$M = M_0 + \mathbf{N} \cdot X$$

has at least one solution $X \in \mathbb{N}^{|T|}$.

Note that the marking equation provides only a necessary condition for reachability. If the marking equation is infeasible, then M is not reachable from M_0 , but the inverse does not hold in general: there are markings satisfying the marking equation which are not reachable. Those markings are said to be *spurious* [59]. Fig. 16(a,b,c) presents an example of spurious marking: the Parikh vector $\sigma = (320011)$ and the marking $M = (00020)$ are a solution of the marking equation shown in Fig. 16(b) for the Petri net in Fig. 16(a)¹. However, M is not reachable: only sequences visiting *negative* markings can lead to M . Fig. 16(c) depicts the graph containing the reachable markings and the spurious markings (shadowed). This graph is called the *potential reachability graph*. The initial marking is represented by the state (10000).

¹ Both in the figure and the explanation, we abuse the notation and skip the commas in the definition of Parikh vectors and markings.

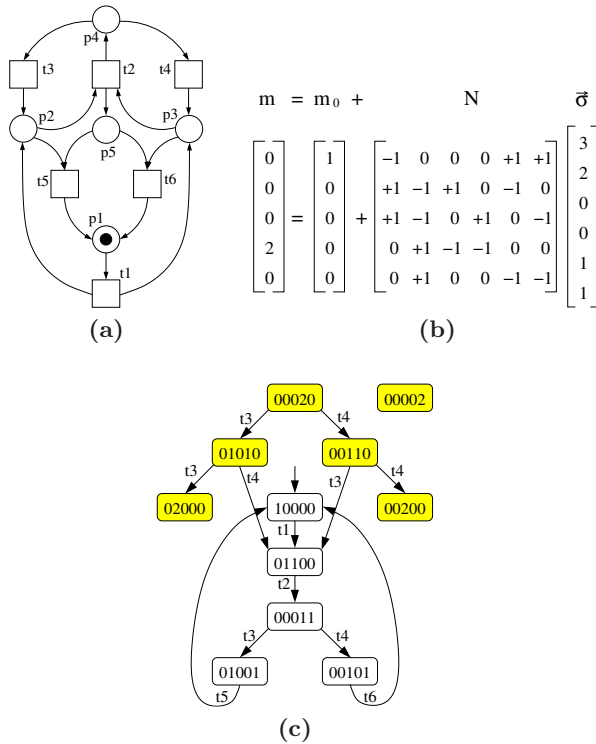


Fig. 16. A Petri net (a), a spurious solution $M = (00020)^T$ (b), and the potential reachability graph (c).

ILP models to find encoding conflicts. Let us explain with the example of the VME Bus Controller how to derive an ILP formulation that detects USC/CSC conflicts in a given STG.

The incidence matrix of the STG corresponding to the VME example is as follows:

	lds^+	dsr^+	$ldtack^+$	$ldtack^-$	d^+	$dtack^-$	$dtack^+$	lds^-	drs^-	d^-
p_1	+1	0	-1	0	0	0	0	0	0	0
p_2	0	0	+1	0	-1	0	0	0	0	0
p_3	0	0	0	0	+1	0	-1	0	0	0
p_4	0	0	0	0	0	0	+1	0	-1	0
p_5	0	0	0	0	0	0	0	0	+1	-1
p_6	0	0	0	0	0	-1	0	0	0	+1
p_7	0	-1	0	0	0	+1	0	0	0	0
p_8	-1	+1	0	0	0	0	0	0	0	0
p_9	0	0	0	0	0	0	0	-1	0	+1
p_{10}	0	0	0	-1	0	0	0	+1	0	0
p_{11}	-1	0	0	+1	0	0	0	0	0	0

The initial marking of the underlying Petri net is $M_0 \stackrel{\text{df}}{=} (00000010001)$, and the vector $x = (1110000000)$ is a solution of the marking equation ($M_1 = M_0 + \mathbf{N}x$). It means that the sequence of transitions corresponding to the Parikh vector x is fireable at M_0 , and it leads to M_1 , where $M_1 = (01000000000)$. From M_1 , the vector $z = (0100111011)$ is a solution of the marking equation, ($M_2 = M_1 + \mathbf{N}z$), where $M_2 = (00000001100) \neq M_1$. The non-zero positions of vector z correspond to transitions d^+ , $dtack^+$, $dscr^-$, d^- , $dtack^-$ and $dscr^+$. Looking at vector z , one can realise that for each signal appearing in it, the same number of rising and falling transitions of the signal appear (for instance, d^+ and d^- occur once). This type of sequences are called *complementary sequences*. The importance of finding complementary sequences is due to the fact that they connect two markings (M_1 and M_2 in the example) that have the same encoding, since that each signal appearing in the sequence *ends up with the same value that it had at the beginning*. The reader can assign any meaningful value to each signal in marking M_1 and check that M_2 will have the same encoding.

So, according to the marking equation, there are two different markings, M_1 and M_2 , such that M_2 is reachable from M_1 by firing a complementary sequence, i.e., both markings have the same encoding. We found an USC conflict. The corresponding ILP model is:

ILP model for USC checking:

Reachability conditions:

$$M_1 = M_0 + \mathbf{N}x$$

$$M_2 = M_1 + \mathbf{N}z$$

$$M_1, M_2, x, z \geq 0, x, z \in \mathbb{Z}^{|T|}$$

z is complementary seq.

$$M_1 \neq M_2$$

(1)

Note, that as it was said in the previous section, the marking equation provides only sufficient conditions for a marking to be reachable. Therefore the markings M_1 and M_2 , that are solution for model (1), can indeed be spurious, and the corresponding model will incorrectly use them as example of encoding conflicts. This is why in the introduction we said that our ILP models are non-perfect oracles: only when the model finds no solution (a conflict between two markings) one can be sure that the STG is free of conflicts. On the contrary, when they find a conflict, only for very restricted classes of nets (marked graphs or live, safe and cyclic free-choice nets [21]) one can be sure that the conflict is a real one.

Now let us show how to find CSC conflicts using ILP techniques. Informally, for a given signal a of the STG, a CSC conflict exists for a if the following conditions hold: let a_i^\pm be a transition of signal a . Then, a CSC conflict exists if: (i) M_2 is reachable from M_1 , (ii) M_1 and M_2 have the same code, (iii) a_i^\pm is enabled in M_1 and (iv) for every transition a_j^\pm of signal a , a_j^\pm is not enabled in M_2 . For safe systems, the enabledness of a transition x at a marking M can be characterised by the sum of tokens of the places in $\bullet x$ at M : x is enabled at M

if and only if the sum of tokens of the places in $\bullet x$ is equal to the number of places in $\bullet x$:

ILP model for CSC checking:

$$\begin{aligned}
 (i) \quad & \boxed{\text{Reachability conditions (same as in (1))}} \\
 (ii) \quad & \mathbf{z \text{ is complementary seq.}} \\
 (iii) \quad & \sum_{p \in \bullet a_i^\pm} M_1(p) = |\bullet a_i^\pm| \\
 (iv) \quad & \forall a_j^\pm : \sum_{p \in \bullet a_j^\pm} M_2(p) < |\bullet a_j^\pm|
 \end{aligned} \tag{2}$$

Note that the constraint $M_1 \neq M_2$ is not needed in (2). If we continue with the example of the VME Bus Controller, it can be shown that the USC conflict described in the previous section is also a CSC conflict for signal d . Given that it has been shown before that the assignments $x = (1110000000)$ and $z = (0100111011)$ satisfy the first two constraints, now we show that constraints (iii) and (iv) are also satisfied by x and z . The former constraint is satisfied because

$$\sum_{p \in \bullet d^+} M_1(p) = M_1(p_2) = 1 = |\{p_2\}| = |\bullet d^+|,$$

and constraint (iv) is also satisfied since

$$\sum_{p \in \bullet d^+} M_2(p) = M_2(p_2) = 0 < 1 = |\{p_2\}| = |\bullet d^+|.$$

Note that constraint (iv) is not verified for transition d^- , because the consistency of the STG is assumed. Thus, a CSC conflict has been detected in the VME Bus Controller example.

Experimental Results on Using ILP to Verify the Encoding. The ILP methods presented have been implemented in MOEBIUS, a tool for the synthesis of speed-independent circuits. The experiments have been performed on a *PentiumTM* 4/2.53 GHz and 512M RAM.

The experiments for CSC/USC detection are presented in Tables 1 and 2. Each table reports the CPU time of each approach in seconds. We use ‘time’ and ‘mem’ to indicate that the algorithm had not completed within 10 hours or produced a memory overflow, respectively. The following tools were compared:

- CLP: the approach presented in [38] for the verification of USC/CSC. It uses non-linear integer programming methods and works on STG unfolding.
- SAT: the approach presented in [39] for the verification of CSC². It uses a satisfiability solver and works on STG unfolding.
- ILP: the approach presented in this section.

From the results one can conclude, as it was expected, that checking USC is easier than checking CSC, given the different nature of the two problems: for

² Checking for USC is not implemented in our version of SAT

Table 1. CSC detection for well-structured STGs.

Benchmark	$ P $	$ T $	$ Z $	CLP	SAT	ILP
PpWk(2,9)	71	38	19	< 1	< 1	< 1
PpWk(2,12)	95	50	25	< 1	< 1	< 1
PpWkCsc(2,9)	72	38	19	3	< 1	< 1
PpWkCsc(2,12)	96	50	25	246	1	< 1
PpWk(3,6)	70	38	19	< 1	< 1	< 1
PpWk(3,9)	106	56	28	11	< 1	< 1
PpWk(3,12)	142	74	37	933	< 1	< 1
PpWkCsc(3,6)	72	38	19	3	< 1	< 1
PpWkCsc(3,9)	108	56	28	2075	< 1	< 1
PpWkCsc(3,12)	144	74	37	time	1	< 1
PpARB(2,9)	86	48	23	< 1	< 1	< 1
PpARB(2,12)	110	60	29	< 1	< 1	< 1
PpARBcsc(2,9)	88	48	23	41	< 1	< 1
PpARBcsc(2,12)	112	60	29	1022	16	< 1
PpARB(3,6)	92	54	25	< 1	< 1	< 1
PpARB(3,9)	128	72	34	< 1	< 1	< 1
PpARB(3,12)	164	90	43	< 1	< 1	< 1
PpARBcsc(3,6)	95	54	25	61	< 1	< 1
PpARBcsc(3,9)	131	72	34	time	2	< 1
PpARBcsc(3,12)	167	90	43	time	16	1
TANGRAMCsc(3,2)	142	92	38	< 1	< 1	1
TANGRAMCsc(4,3)	321	202	83	< 1	< 1	9
ART(10,9)	216	198	99	< 1	< 1	< 1
ART(20,9)	436	398	199	5	10	< 1
ART(30,9)	656	598	299	38	82	< 1
ART(40,9)	876	798	399	138	265	< 1
ART(50,9)	1096	998	499	377	630	1
ARTCsc(10,9)	752	630	315	time	861	182
ARTCsc(20,9)	1532	1270	635	time mem		1623
ARTCsc(30,9)	2312	1910	955	time mem		5413
ARTCsc(40,9)	3092	2550	1275	time mem		12602
ARTCsc(50,9)	3872	3190	1595	time mem		25210

verifying USC only one ILP model is needed to be solved, whereas for verifying CSC n models are needed, where n is the number of non-input signals in the STG. Moreover, when some encoding conflict exists, the ILP solver can find it in short time. This is explained by the fact that proving the absence of encoding conflicts requires an exhaustive exploration of the *branch-and-bound* tree visited by ILP solvers.

The speed-up shown by ILP with respect to the unfolding approach of SAT or CLP are because in ILP approximations of the state space are used, whereas SAT or CLP (as will be explained in the next section) are exact. However, our conservative approach has proven to be highly accurate in the experimental results.

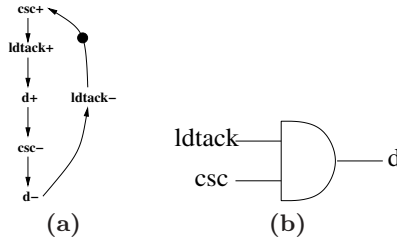
4.4 Computing the Necessary Support for a Given Signal

In this section we are going to adapt model (2) to derive a decomposition method for the synthesis of a given signal. The main idea is to try to compute those signals in the STG that are needed to ensure that a given signal will be free of encoding conflicts, if we abstract away of the rest of the STG. We will call such a set of signals a *support*.

Table 2. USC detection for well-structured STGs.

Benchmark	$ P $	$ T $	$ Z $	CLP	ILP
PpWk(3,9)	106	56	28	10	< 1
PpWk(3,12)	142	74	37	876	< 1
PpWkCsc(3,9)	108	56	28	2002	< 1
PpWkCsc(3,12)	144	74	37	time	1
PpARB(3,9)	128	72	34	< 1	< 1
PpARB(3,12)	164	90	43	< 1	< 1
PpARBCsc(3,9)	131	72	34	time	1
PpARBCsc(3,12)	167	90	43	time	1
TANGRAM(3,2)	142	92	38	< 1	1
TANGRAM(4,3)	321	202	83	< 1	6
ART(40,9)	876	798	399	146	1
ART(50,9)	1096	998	499	328	2
ARTCsc(40,9)	3092	2550	1275	time	851
ARTCsc(50,9)	3872	3190	1575	time	1387

Let us use as example the STG shown in Fig. 12(a), where a new signal (*csc*) has been inserted in the original STG of the VME Bus Controller to solve the encoding conflict. A possible support for signal *d* is $\{ldtack, csc\}$. Fig. 17(a) shows the projection induced by this support, and the final implementation of *d* is shown in Fig. 17(b). The rest of this section is devoted to explaining how to compute efficiently a support for a given output signal *a*.

**Fig. 17.** Projection of the STG in Fig. 12 for signal *d* (a) and a circuit implementing *d* (b).

The computation of a support can be performed iteratively: starting from an initial assignment, ILP techniques can be used to guide the search. Suppose we have an initial candidate set of signals $Z' \subseteq Z$, candidate to be a support of a given signal *a*. A way of determining whether Z' is a support for signal *a* is by solving the following ILP problem:

ILP model for checking support:

$$\boxed{(i), (iii) \text{ and } (iv) \text{ from } (2)} \quad (3)$$

z is complementary seq. for signals in Z'

If (3) is infeasible, then Z' is enough for implementing *a*. Otherwise the set Z' must be augmented (from signals in $Z \setminus Z'$) with more signals until (3) is

infeasible. Moreover if (3) is feasible, adding a complemented signal b from $Z \setminus Z'$ will not turn the problem infeasible because z is still a complementary sequence for signals in $Z' \cup \{b\}$. On the contrary, adding an uncomplemented signal will assign a different code to markings M_1 and M_2 of (3). Therefore, the uncomplemented signals in z will be the candidates to be added to Z' . The algorithm for finding a support set for a non-input signal a is the following:

Algorithm for the calculation of support:

Support (STG S , Signal a) **returns** support of a

```

 $Z' := \text{Trig}(a) \cup \{a\}$ 
while (3) is infeasible do
    Let  $b$  be an uncomplemented signal in  $z$ 
     $Z' := Z' \cup \{b\}$ 
endwhile
return  $Z'$ 
    
```

where $\text{Trig}(a)$ is the set of signals that directly cause the switching of signal a . In the next section we are going to present an example of using this algorithm for the synthesis of the VME Bus Controller STG specification.

4.5 Synthesis of the VME Bus Controller Using Structural Methods

Let us show how to use the structural methods to synthesise the VME example. In addition to the methods presented in this section, we use Petri net transformations for stepwise transformation and projection. For a formal presentation of the kit of transformations used in the example, see [9].

First, as shown in Section 4.3, we can use the ILP model (1) to realise that the original STG of the VME Bus controller has encoding conflicts. Consequently we apply the encoding technique presented in Section 4.2 to enforce CSC.

Afterwards, in order to derive a efficient implementation, we can try to eliminate as many signals inserted by the encoding technique as possible, while keeping a correct encoding. The idea is to eliminate a signal and only accept the removal if the transformed STG still has a correct encoding. Fig. 18 shows how the removal of the first five signals is done, using the USC ILP model (1) as an oracle.

The process can be iterated until no more signals can be removed. The final STG is shown in the centre of Fig. 19. From that STG, the algorithm for support computation is run for every output signal, and the corresponding projection is found. This is shown also in Fig. 19.

And finally, from each projection the corresponding circuit is obtained. Given that the projections are usually small (the support for a given signal is often very small in practice, and the corresponding projections are usually quite small), state-based algorithms for synthesis introduced in Section 3 can be applied. The final synthesis of each projection is shown in Fig. 20.

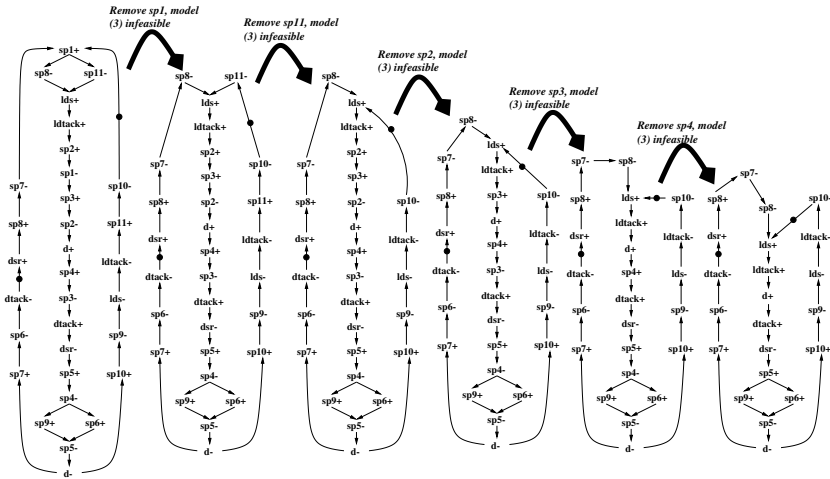


Fig. 18. Greedy removal of signals sp_1 , sp_{11} , sp_2 , sp_3 and sp_4 .

Table 3 shows the results of experiments on synthesis to check the quality of the generated circuits. The column ‘Lit’ reports the number of literals, in factored form, of the netlist. The results are compared with the circuits obtained by PETRIFY [18], a state-based synthesis tool, on the same controllers. From the reported CPU time, the time needed for computing a support and the corresponding projection was negligible compared with the time needed for deriving logic equations. Table 3 shows that the quality of the circuits obtained by the ILP-based technique is comparable to that of the circuits obtained by PETRIFY. Moreover it is clear that the structural approach can deal with larger specifications.

Table 3. Support computation, projection and synthesis compared to state-based approach.

benchmark	states	$ P $	$ T $	$ Z $	Lit.		CPU	
					P _{fy}	ILP	P _{fy}	ILP
PfWkCsc(2,6)	8192	47	26	19	57	57	5	1
PfWkCsc(2,9)	524.288	71	38	19	87	87	49	2
PfWkCsc(3,9)	2.7×10^7	106	56	28	mem	130	mem	3
PfWkCsc(3,12)	2.2×10^{11}	142	74	37	time	117	time	3
PfArbCsc(2,6)	61440	62	36	17	77	77	21	83
PfArbCsc(2,9)	3.9×10^6	110	60	29	107	107	185	59
PfArbCsc(3,9)	3.3×10^9	131	72	34	163	165	10336	289
PfArbCsc(3,12)	1.7×10^{12}	167	90	43	time	210	time	608
TangramCsc(3,2)	426	142	92	38	97	103	56	146
TangramCsc(4,3)	9258	321	202	83	mem	247	mem	7206

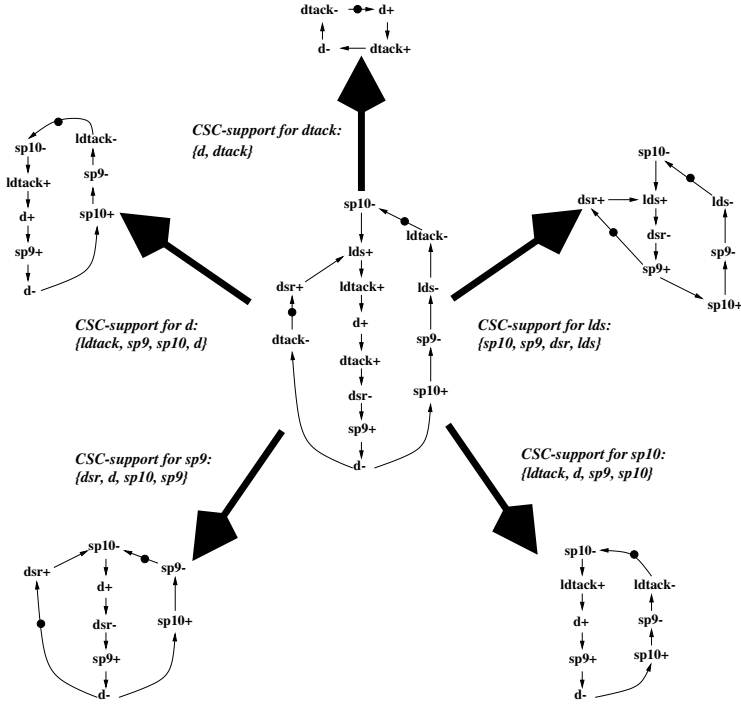


Fig. 19. Support computation and projection for the VME Bus Controller example.

4.6 Conclusions

Several examples of using structural methods are presented in this section. We have given intuition on how these methods are used for the problem of synthesis of control circuits from STG specifications. Although in some cases they can provide only sufficient conditions, in general those methods are highly accurate and provide a significant speed-up with respect to other approaches, as has been demonstrated by the experimental results shown for the problem of verifying the encoding.

In conclusion, structural methods are necessary for being able to handle large and concurrent specifications. We advocate for their use, either isolated or in combination with approaches like the one presented in the next section.

5 Synthesis Using Petri Net Unfoldings

While the state-based approach is relatively simple and well-studied, the issue of computational complexity for highly concurrent STGs is quite serious due to the state space explosion problem. This puts practical bounds on the size of control circuits that can be synthesised using such techniques, which are often restrictive, especially if the STG models are not constructed manually by a designer but rather generated automatically from high-level hardware descriptions.

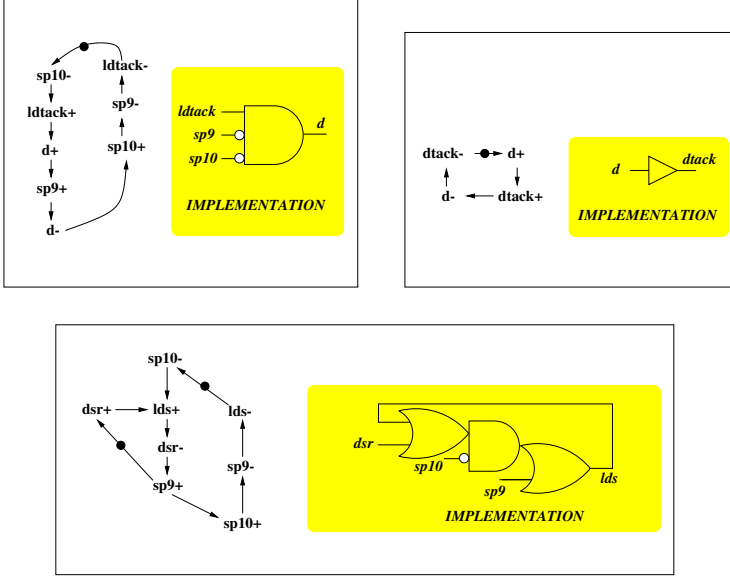


Fig. 20. Speed-independent synthesis of the VME Bus Controller.

In order to alleviate this problem, Petri net analysis techniques based on causal partial order semantics, in the form of Petri net unfoldings, are applied to circuit synthesis. In particular, the following tasks are addressed: (i) detection of encoding conflicts; (ii) resolution of encoding conflicts; and (iii) derivation of Boolean equations for output signals. We show that the notion of an encoding conflict can be characterised in terms of satisfiability of a Boolean formula (SAT), and the resulting algorithms solving tasks (i) and (iii) achieve significant speedups compared with methods based on state graphs. Moreover, we propose a framework for resolution of encoding conflicts (task (ii)) based on *conflict cores*.

5.1 STG Unfoldings

A *finite and complete unfolding prefix* π of an STG Γ is a finite acyclic net which implicitly represents all the reachable states of Γ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* Γ , by successive firings of transition, under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. The unfolding is infinite whenever Γ has an infinite run; however, if Γ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix. We denote by B , E and $E_{cut} \subseteq E$ the sets of conditions, events and cut-off events of the prefix, respectively. Fig. 21(b) shows

a finite and complete unfolding prefix (with the only cut-off event is depicted as a double box) of the STG shown in Fig. 21(a).

Efficient algorithms exist for building such prefixes [25, 31, 36, 37], which ensure that the number of non-cut-off events in a complete prefix can never exceed the number of reachable states of Γ . However, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional ‘diamonds’ as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} nodes, whereas the complete prefix will coincide with the net itself.

Due to its structural properties (such as acyclicity), the reachable markings of Γ can be represented using *configurations* of π . A configuration C is a downward-closed set of events (being downward-closed means that if $e \in C$ and f is a causal predecessor of e , then $f \in C$) without structural conflicts (i.e., for all distinct events $e, f \in C$, $\bullet e \cap \bullet f = \emptyset$). Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of concurrent events is not important.

After starting π from the implicit initial marking (whereby one puts a single token in each condition which does not have an incoming arc) and executing all the events in C , one reaches the marking denoted by $Cut(C)$. We denote by $Mark(C)$ the corresponding marking of Γ , reached by firing a transition sequence corresponding to the events in C . It is remarkable that each reachable marking of Γ is $Mark(C)$ for some configuration C , and, conversely, each configuration C generates a reachable marking $Mark(C)$. This property is a primary reason why various behavioural properties of Γ can be re-stated as the corresponding properties of π , and then checked, often much more efficiently (in particular, one can easily check the consistency and deadlock-freeness of Γ [57, 35]). The experimental results in Table 4 demonstrate that high levels of compression are indeed achieved in practice.

For the unfolding of a consistent STG we define by $Code_z(C)$ the value (0 or 1) corresponding to signal z in the encoding of the state $Mark(C)$; we also define $Out_z(C)$ to be 1 if $z \in Out(M)$ and 0 otherwise, and $Nxt_z(C) \stackrel{\text{df}}{=} Code_z(C) \oplus Out_z(C)$, where ‘ \oplus ’ is the ‘exclusive or’ operation.

5.2 Visualisation and Resolution of State Encoding Conflicts

A number of methods for resolution of CSC conflicts have been proposed so far (see, e.g., [16] for a brief review). The techniques in [67, 75] introduce constraints within an STG, called *lock relation* and *coupledness relation*, which provide some guidance. These techniques recognise that if all pairs of signals in the STG are ‘locked’ using a chain of handshaking pairs then the STG satisfies the CSC property. The synthesis tool PETRIFY uses the theory of regions [16] for this purpose.

The above techniques work reasonably well. However, they may produce sub-optimal circuits or fail to solve the problem in certain cases, e.g., when

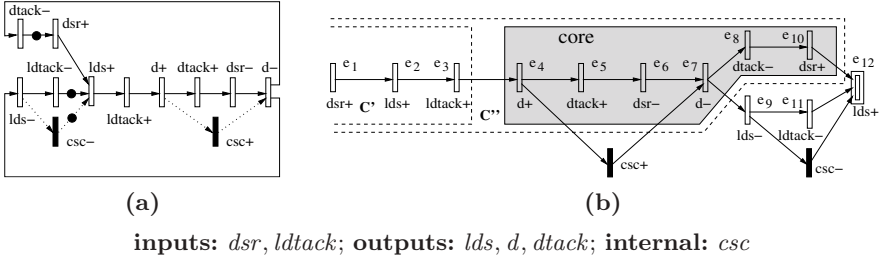


Fig. 21. VME bus controller: STG for the read cycle, with the dotted lines appearing in the final STG satisfying the CSC property (a) and unfolding prefix with a conflict pair of configurations and a new signal *csc* resolving the CSC conflict (b). The order of signals in the binary encodings is: *dsr*, *dtack*, *lds*, *ldtack*, *d*.

a controller specification is defined in a compact way using a small number of signals. Such specifications often have CSC conflicts that are classified as irreducible by PETRIFY. Therefore, manual design may be required for finding good synthesis solutions, particularly in constructing interface controllers, where the quality of the solution is critical for the system's performance.

According to a practising designer [54], the synthesis tool should offer a way for the user to understand the characteristic patterns of a circuit's behaviour and the cause of each encoding conflict and allow one to interactively manipulate the model by choosing where in the specification to insert new signals. The visualisation method presented here is aimed at facilitating a manual refinement of an STG with CSC conflicts, and works on the level of unfolding prefixes. In order to avoid the explicit enumeration of encoding conflicts, they are visualised as *cores*, i.e., sets of transitions 'causing' one or more of them. All such cores must eventually be eliminated by adding new signals that resolve the encoding conflicts to yield an STG satisfying the CSC property. Optionally, our method can also work in a completely automatic or semi-automatic manner, making it possible for the designer to see what is going on and intervene at any stage during the process of CSC conflict resolution.

5.3 Encoding Conflicts in a Prefix

A CSC conflict can be represented as an unordered *conflict pair* of configurations $\langle C', C'' \rangle$ whose final states are in CSC conflict, as shown in Fig. 21(b). In Section 5.6 a SAT-based technique for detecting CSC conflicts is described. Essentially, it allows for efficiently finding such conflict pairs in STG unfolding prefixes.

Note that the set of all conflict pairs may be quite large, e.g., due to the following 'propagation' effect: if C' and C'' can be expanded by the same event e then $\langle C' \cup \{e\}, C'' \cup \{e\} \rangle$ is also a conflict pair (unless these two configurations enable the same set of output signals). Therefore, it is desirable to reduce the number of pairs needed to be considered, e.g., as follows. A conflict pair $\langle C', C'' \rangle$ is called *concurrent* if $C' \not\subseteq C''$, $C'' \not\subseteq C'$ and $C' \cup C'$ is a configuration.

Proposition 1 ([45]). *Let $\langle C', C'' \rangle$ be a concurrent CSC conflict pair. Then $C \stackrel{\text{df}}{=} C' \cap C''$ is such that either $\langle C, C' \rangle$ or $\langle C, C'' \rangle$ is a CSC conflict pair.*

Thus concurrent conflict pairs are ‘redundant’ and should not be considered. The remaining conflict pairs can be classified as follows:

Conflicts of type I are such that $C_1 \subset C_2$ (Fig. 21(b) illustrates this type of CSC conflicts).

Conflicts of type II are such that $C_1 \setminus C_2 \neq \emptyset \neq C_2 \setminus C_1$ and there exist $e' \in C_1 \setminus C_2$ and $e'' \in C_2 \setminus C_1$ such that $e' \# e''$.

The following notion is crucial for the proposed approach:

Definition 4. *Let $\langle C', C'' \rangle$ be a conflict pair. The corresponding complementary set is defined as $\mathcal{CS} \stackrel{\text{df}}{=} C' \Delta C''$, where Δ denotes the symmetric set difference. \mathcal{CS} is a core if it cannot be represented as the union of several disjoint complementary sets. A complementary set is of type I/II if the corresponding conflict pair is of type I/II, respectively. \diamond*

For example, the core corresponding to the conflict pair shown in Fig. 21(b) is $\{e_4, \dots, e_8, e_{10}\}$ (note that for a conflict pair $\langle C', C'' \rangle$ of type I, such that $C' \subset C''$, the corresponding core is simply $C'' \setminus C'$).

One can show that every complementary set \mathcal{CS} can be partitioned into $C_1 \setminus C_2$ and $C_2 \setminus C_1$, where $\langle C', C'' \rangle$ is a conflict pair corresponding to \mathcal{CS} . Moreover, if \mathcal{CS} is of type I then one of these parts is empty, while the other is \mathcal{CS} itself. An important property of complementary sets is that for each signal $z \in Z$, the difference between the numbers of z^+ - and z^- -labelled events in \mathcal{CS} is the same in these two parts (and is 0 if \mathcal{CS} is of type I). This suggests that a complementary set can be eliminated by introduction of a new internal signal and insertion of its transition into this set, as this would violate the stated property.

It is often the case that the same complementary set corresponds to different conflict pairs, so the designer can save time by analysing the cores rather than the full list of CSC conflicts, which can be much longer.

5.4 Framework for Visualisation and Resolution of Encoding Conflicts

The visualisation is based on showing the designer the cores in the STG’s unfolding prefix. Since every element of a core is an instance of the STG’s transition, the cores can easily be mapped from the prefix to the STG. For example, the core $\{e_4, \dots, e_8, e_{10}\}$ in Fig. 21(b) can be mapped to the set of transitions $\{d^+, dtack^+, dsr^-, d^-, dtack^-, dsr^+\}$ of the original STG shown in Fig. 21(a).

Cores are important for resolution of encoding conflicts. By introducing an additional internal signal and insertion of its transition, say csc^+ , one can destroy a core eliminating thus the corresponding encoding conflicts. To preserve the consistency of the STG, the signal transition’s counterpart, csc^- , must also

be added to the specification *outside the core*, in such a way that it is neither concurrent to nor in structural conflict with csc^+ . It is sometimes possible to insert csc^- into another core thus eliminating it also, as shown in Fig. 22(b). Another restriction is that an inserted signal transitions cannot trigger an input signal transition (the reason is that this would impose constraints on the environment which were not present in the original STG, making it ‘wait’ for the newly inserted signal). More about the formal requirements for the correctness of inserting a new transition can be found in [18].

The core in Fig. 21(b) can be eliminated by inserting a new signal, csc^+ , somewhere in the core, e.g., concurrently to e_5 and e_6 between e_4 and e_7 , and by inserting its complement outside the core, e.g., concurrently to e_{11} between e_9 and e_{12} . (Note that concurrent insertion of these two transitions avoids an increase in the latency of the circuit, where each transition is assumed to contribute a unit delay.) The final STG satisfying the CSC property is shown in Fig. 21(a) with dotted lines taken into account.

It is often the case that cores overlap. In order to minimise the number of inserted signals, and thus the area and latency of the circuit, it is advantageous to insert a signal in such a way that as many cores as possible are eliminated by it. That is, a signal should be inserted into *the intersection of several cores* whenever possible.

To assist the designer in exploiting core overlaps, another key feature of our method, viz. the *height map* showing the quantitative distribution of the cores, is employed in the visualisation process. The events located in conflict cores are highlighted by shades of colours. The shade depends on the *altitude* of an event, i.e., on the number of cores it belongs to. (The analogy with a topographical map showing the altitudes may be helpful here.) The greater the altitude, the darker the shade. ‘Peaks’ with the highest altitude are good candidates for insertion of a new signal, since they correspond to the intersection of maximum number of cores.

Using this representation, the designer can select an area for insertion of a new signal and obtain a local, more detailed description of the cores overlapping with the selection. When an appropriate core cluster is chosen, the designer can decide how to insert a new signal transition optimally, taking into account the design constraints and his/her knowledge of the system being developed.

The overview of the process of resolution of CSC conflicts is shown in Fig. 22(a). Given an STG, a finite and complete prefix of its unfolding is constructed, and the cores are computed. If there are none, the process stops. Otherwise, the height map is shown to the designer, who chooses a set of overlapping cores. In phases one and two, an additional signal transition splitting the core is inserted together with its counterpart. The inserted transitions are then transferred to the STG, and the process is repeated. Depending on the number of conflict cores, the resolution process may involve several cycles.

After completion of phase one, the height map is updated. The altitudes of the events in the core cluster where the new signal transition has been inserted are made negative, to prompt the designer that if the counterpart transition is

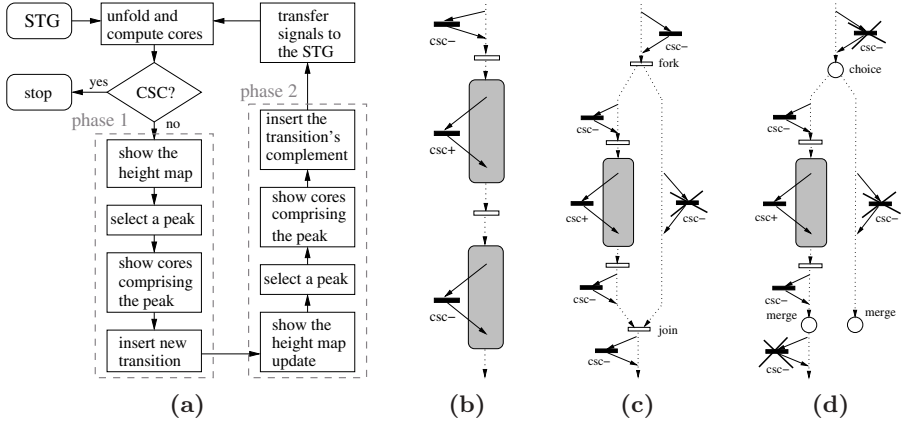


Fig. 22. The process of resolution of encoding conflicts (a) and strategies for eliminating conflict cores (b–d). Several possibilities are shown for insertion of csc^- , but only one of them should be used. (The positions where csc^- is *not* allowed are shown as transitions that are crossed out.)

inserted there, some of the cores in the cluster will reappear. Moreover, in order to ensure that the insertion of the counterpart transition preserves consistency, the areas where it cannot be inserted (in particular, the events concurrent to or in structural conflict with this transition) are faded out.

Typical cases in STG specifications are schematically illustrated in Fig. 22(b–d). Cores ‘in sequence’, can be eliminated in a ‘one-hot’ manner as depicted in Fig. 22(b). Each core is eliminated by one signal transition, and its complement is inserted outside the core, preferably, into another non-adjacent one³.

An STG that has a core in one of the concurrent branches can also be tackled in a ‘one-hot’ way, as shown in Fig. 22(c). Note that in order to preserve the consistency the transition’s counterpart cannot be inserted into the concurrent branch, but can be inserted before the fork transition or after the join one. In a branch which is in a structural conflict with another branch, the transition’s counterparts must be inserted in the same branch somewhere between the choice and the merge points, as shown in Fig. 22(d).

Obviously, the described cases do not cover all possible situations and all possible insertions (e.g., one can sometimes insert a new signal transition before the choice point and its counterparts into *each* branch, etc.), but we hope they do give an idea how the cores can be eliminated. [45] presents this method of resolution of CSC conflicts using STG unfoldings in more detail.

5.5 Boolean Satisfiability

Boolean satisfiability problem (SAT) has great theoretical interest as the canonical \mathcal{NP} -complete problem. Though it is very unlikely that it can be solved in

³ The union of two adjacent cores is usually a complementary set which will not be destroyed if both the transition and its counterpart are inserted into it.

polynomial time, there are algorithms which can solve many interesting SAT instances quite efficiently. SAT solvers have been successfully applied to many practical problems such as AI planning, ATPG, model checking, etc. The research in SAT has led to algorithms which routinely solve SAT instances generated from industrial applications with tens of thousands or even millions variables [78].

Thus it is often advantageous to re-state the problem at hand in terms of SAT, and then apply an existing SAT solver. In this paper, the SAT approach will be used for detection of CSC conflicts in Section 5.6 and derivation of equations for logic gates of the circuit in Section 5.7.

The *Boolean satisfiability problem (SAT)* consists in finding a *satisfying assignment*, i.e., a mapping $A : \text{Var}_\varphi \rightarrow \{0, 1\}$ defined on the set of variables Var_φ occurring in a given Boolean expression φ such that φ evaluates to 1. This expression is often assumed to be given in the *conjunctive normal form (CNF)* $\bigwedge_{i=1}^n \bigvee_{l \in L_i} l$, i.e., it is represented as a conjunction of *clauses*, which are disjunctions of *literals*, each literal l being either a variable or the negation of a variable. It is assumed that no two literals in the same clause correspond to the same variable.

Some of the leading SAT solvers, e.g., zCHAFF [48], can be used in the *incremental mode*, i.e., after solving a particular SAT instance the user can modify it (e.g., by adding and/or removing a small number of clauses) and execute the solver again. This is often much more efficient than solving these related instances as independent problems, because on the subsequent runs the solver can use some of the useful information (e.g., learnt clauses, see [78]) collected so far. In particular, such an approach can be used to compute *projections* of assignments satisfying a given formula, as described in sequel.

Let $V \subseteq \text{Var}_\varphi$ be a non-empty set of variables occurring in a formula φ , and Proj_V^φ be the set of all restricted assignments (or projections) $A|_V$ such that A is a satisfying assignment of φ . Using the incremental SAT approach it is possible to compute Proj_V^φ , as follows.

Step 0: $\mathcal{A} := \emptyset$.

Step 1: Run the SAT solver for φ .

Step 2: If φ is unsatisfiable then return \mathcal{A} and terminate.

Step 3: Add $A|_V$ to \mathcal{A} , where A is the computed satisfying assignment.

Step 4: Append to φ a new clause $\bigvee_{v \in V \wedge A(v)=1} \neg v \vee \bigvee_{v \in V \wedge A(v)=0} v$.

Step 5: Go back to Step 1.

Suppose now that we are interested in finding only the minimal elements of Proj_V^φ , assuming that $A|_V \leq A'|_V$ if $(A|_V)(v) \leq (A'|_V)(v)$, for all $v \in V$. The above procedure can then be modified by changing Step 4 to:

Step 4': Append to φ a new clause $\bigvee_{v \in V \wedge A(v)=1} \neg v$.

Similarly, if we were interested in finding all the maximal elements of Proj_V^φ , then one could change Step 4 to:

Step 4'': Append to φ a new clause $\bigvee_{v \in V \wedge A(v)=0} v$.

Moreover, in the latter two cases, before terminating an additional pass over the elements stored in \mathcal{A} should be made in order to eliminate any non-minimal (or non-maximal) projections.

5.6 Detection of State Encoding Conflicts Using SAT

Let C' and C'' be two configurations of the unfolding of a consistent STG and z be an output signal. C' and C'' are in *Complete State Coding conflict for z (CSC^z conflict)* if $Code_x(C') = Code_x(C'')$ for all $x \in Z$ and $Nxt_z(C') \neq Nxt_z(C'')$. This notion is very similar to the notion of a CSC conflict; in particular, each CSC^z conflict is a CSC conflict, and each CSC conflict is a CSC^z conflict for some output signal z , i.e., the problem of detection of CSC conflicts is easily reducible to the problem of detection of CSC^z conflicts, and we will mostly concentrate on the latter problem. A CSC^z conflict can be represented as an unordered *conflict pair* of configurations $\langle C', C'' \rangle$ whose final states are in CSC^z conflict; for example, the conflict pair of configurations shown in Fig. 21(b) is in CSC^{lds} and CSC^d conflict.

Constructing a SAT Instance. We adopt the following naming conventions. The variable names are in the lower case and names of formulae are in the upper case. Names with a single prime (e.g., conf'_e and \mathcal{CONF}') are related to C' , and ones with double prime (e.g., conf''_e) are related to C'' . If there is no prime then the name is related to both C' and C'' . If a formula name has a single prime then the formula does not contain occurrences of variables with double primes, and the counterpart double prime formula can be obtained from it by adding another prime to every variable with a single prime. The subscript of a variable points to which element of the STG or the prefix the variable is related, e.g., conf'_e and conf''_e are both related to the event e of the prefix. By a name without a subscript we denote the list of all variables for all possible values of the subscript, e.g., conf' denotes the list of variables conf'_e , where e runs through the set $E \setminus E_{cut}$.

The following Boolean variables will be used in the proposed translations:

- For each event $e \in E \setminus E_{cut}$, we create two Boolean variables, conf'_e and conf''_e , tracing whether $e \in C'$ and $e \in C''$ respectively.
- For each signal $x \in Z$, we create a variable code_x to trace the value of x . Since the values of all the signals must match at the final states of C' and C'' , we use the same set of variables for both configurations.
- For each condition $b \in B \setminus E_{cut}^\bullet$ which is an instance of a place from P_Z^1 (defined later), we create two Boolean variables, cut'_b and cut''_b , tracing whether $b \in \text{Cut}(C')$ and $b \in \text{Cut}(C'')$ respectively.
- For each event $e \in E$ which is an instance of the output signal z for which the CSC^z condition is being checked, we create two Boolean variables, en'_e and en''_e , tracing whether e is ‘enabled’ by C' and C'' respectively. Note that unlike conf' and conf'' , such variables are also created for the cut-off events.

Our aim is to build a Boolean formula \mathcal{CSC}^z such that: (i) \mathcal{CSC}^z is satisfiable iff there is a CSC^z conflict; and (ii) for every satisfying assignment, the two sets

of non-cut-off events of the prefix, $C' \stackrel{\text{df}}{=} \{e \in E \setminus E_{\text{cut}} \mid \text{conf}'_e = 1\}$ and $C'' \stackrel{\text{df}}{=} \{e \in E \setminus E_{\text{cut}} \mid \text{conf}''_e = 1\}$, constitute a conflict pair $\langle C', C'' \rangle$ of configurations. CSC^z will be the conjunction of constraints described below.

For example, these variables will assume the following values for the CSC^d conflict depicted in Fig. 21(b) (the order of signals in the binary codes is: dsr , $dtack$, lds , $ldtack$, d): $\text{conf}' = 111000000000$, $\text{conf}'' = 111111110100$, $\text{code} = 10110$, $\text{en}'_{e_4} = 1$, $\text{en}'_{e_7} = 0$, and $\text{en}''_{e_4} = \text{en}''_{e_7} = 0$ (the values of cut' and cut'' are not shown).

Configuration constraints. The role of first two constraints, $CONF'$ and $CONF''$, is to ensure that C' and C'' are both legal configurations of the prefix (not just arbitrary sets of events). $CONF'$ is defined as the conjunction of the formulae

$$\bigwedge_{e \in E \setminus E_{\text{cut}}} \bigwedge_{f \in \bullet(\bullet e)} (\text{conf}'_e \Rightarrow \text{conf}'_f) \quad \text{and} \quad \bigwedge_{e \in E \setminus E_{\text{cut}}} \bigwedge_{f \in E_e} \neg(\text{conf}'_e \wedge \text{conf}'_f),$$

where $E_e \stackrel{\text{df}}{=} ((\bullet e) \bullet \setminus \{e\}) \setminus E_{\text{cut}}$. The former formula ensures that if $e \in C'$ then all the direct causal predecessors of e are also in C' , which in turn ensures that C' is a downward closed set of events. The latter one ensures that C' contains no structural conflicts. (One should be careful to avoid duplication of clauses when generating this formula.) $CONF''$ is defined similarly.

$CONF'$ and $CONF''$ can be transformed into the CNF by applying the rules $x \Rightarrow y \equiv \neg x \vee y$ and $\neg(x \wedge y) \equiv \neg x \vee \neg y$.

Encoding constraint. First we describe an important STG transformation allowing to capture the current value of each signal in the STG's marking. For each signal $z \in Z$, a pair of complementary places, p_z^0 and p_z^1 , tracing the value of z is added as follows. For each z^+ -labelled transition t , $p_z^0 \in \bullet t$ and $p_z^1 \in t \bullet$, and for each z^- -labelled transition t' , $p_z^1 \in \bullet t'$ and $p_z^0 \in t' \bullet$. Exactly one of these two places is marked at the initial state, accordingly to the initial value of signal z . One can show that at any reachable state of an STG augmented with such places, p_z^0 (respectively, p_z^1) is marked iff the value of z is 0 (respectively, 1). Thus, if a transition labelled by z^+ (respectively, z^-) is enabled then the value of z is 0 (respectively, 1), which in turn guarantees the consistency of the augmented STG. Such a transformation can be done completely automatically (one can easily determine the initial values of all the signals from the unfolding prefix). For a consistent STG, it does not restrict the behaviour and yields an STG with an isomorphic state graph; for a non-consistent STG, the transformation restricts the behaviour and may lead to (new) deadlocks. In what follows, we assume that the tracing places are present in the STG, and denote $P_Z^0 \stackrel{\text{df}}{=} \{p_z^0 \mid z \in Z\}$, $P_Z^1 \stackrel{\text{df}}{=} \{p_z^1 \mid z \in Z\}$, and $P_Z \stackrel{\text{df}}{=} P_Z^0 \cup P_Z^1$.

The role of encoding constraints, $CODE'$ and $CODE''$, is to ensure that the signal codes of the final markings of configurations C' and C'' are equal. To build a formula establishing the value code_z of each signal $z \in Z$ at the final state of C' , we observe that $\text{code}_z = 1$ iff $p_z^1 \in \text{Mark}(C')$, i.e., iff $b \in \text{Cut}(C')$ for some

p_z^1 -labelled condition b (note that the places in P_Z cannot contain more than one token). The latter can be captured by the constraint:

$$\bigwedge_{z \in Z} (\text{code}_z \iff \bigvee_{b \in B_z} \text{cut}'_b),$$

where $B_z \stackrel{\text{df}}{=} \{B \setminus E_{\text{cut}}^\bullet \mid b \text{ is an instance of } p_z^1\}$. We then define \mathcal{CODE}' as the conjunction of the last formula and

$$\bigwedge_{z \in Z} \bigwedge_{b \in B_z} (\text{cut}'_b \iff \bigwedge_{e \in \bullet b} \text{conf}'_e \wedge \bigwedge_{e \in b \bullet \setminus E_{\text{cut}}} \neg \text{conf}'_e),$$

which ensures that $b \in \text{Cut}(C')$ iff the event ‘producing’ b has fired, but no event ‘consuming’ b has fired. (Note that since $|\bullet b| \leq 1$, $\bigwedge_{e \in \bullet b} \text{conf}'_e$ in this formula is either the constant 1 or a single variable.) One can see that if C' is a configuration and \mathcal{CODE}' is satisfied then the value of signal z at the final state of C' is given by code_z . \mathcal{CODE}'' is defined similarly.

The use of the same variables code in both \mathcal{CODE}' and \mathcal{CODE}'' ensures that the encodings of the final states of C' and C'' are the same, if both constraints are satisfied.

It is straightforward to build the CNF of \mathcal{CODE}' :

$$\bigwedge_{z \in Z} \left((\neg \text{code}_z \vee \bigvee_{b \in B_z} \text{cut}'_b) \wedge \bigwedge_{b \in B_z} (\text{code}_z \vee \neg \text{cut}'_b) \wedge \right. \\ \left. \bigwedge_{b \in B_z} \left(\bigwedge_{e \in \bullet b} (\neg \text{cut}'_b \vee \text{conf}'_e) \wedge \bigwedge_{e \in b \bullet \setminus E_{\text{cut}}} (\neg \text{cut}'_b \vee \neg \text{conf}'_e) \wedge (\text{cut}'_b \vee \bigvee_{e \in \bullet b} \neg \text{conf}'_e \vee \bigvee_{e \in b \bullet \setminus E_{\text{cut}}} \text{conf}'_e) \right) \right),$$

and the CNF of \mathcal{CODE}'' can be built similarly.

Next-state constraint. The role of this constraint is to ensure that $\text{Nxt}_z(C') \neq \text{Nxt}_z(C'')$. Since all the other constraints are symmetric w.r.t. C' and C'' , one can rewrite it as $\text{Nxt}_z(C') = 0 \wedge \text{Nxt}_z(C'') = 1$. Moreover, it follows from the definition of Nxt_z that $\text{Nxt}_z(C) \equiv \neg \text{Code}_z(C) \iff \text{Out}_z(C)$, and so the next-state constraint can be rewritten as the conjunction of $\text{Code}_z(C') \iff \text{Out}_z(C')$ and $\neg \text{Code}_z(C'') \iff \text{Out}_z(C'')$.

We observe that an output signal z is enabled by $\text{Mark}(C')$ iff there is a z^+ - or z^- -labelled event $e \notin C'$ ‘enabled’ by C' , i.e., such that $C' \cup \{e\}$ is a configuration (note that e can be a cut-off event). We then define the formula $\mathcal{NEXTZERO}'$, ensuring that $\text{Nxt}_z(C') = 0$, as the conjunction of

$$\text{code}'_z \iff \bigvee_{e \in E_z} \text{en}'_e \quad \text{and} \quad \bigwedge_{e \in E_z} (\text{en}'_e \iff \bigwedge_{f \in \bullet(\bullet e)} \text{conf}'_f \wedge \bigwedge_{f \in (\bullet e) \bullet \setminus E_{\text{cut}}} \neg \text{conf}'_f),$$

where $E_z \stackrel{\text{df}}{=} \{e \in E \mid e \text{ is an instance of } z^\pm\}$. The former conjunct ensures that $\text{Code}_z(C') \iff \text{Out}_z(C')$ (it takes into account that z is enabled by the final

state of C' iff at least one its instance is enabled by C') and the latter one states for each instance e of z that e is enabled by C' iff all the events ‘producing’ tokens in $\bullet e$ are in C' but no events ‘consuming’ tokens from $\bullet e$ (including e itself) are in C' .

The formula $\mathcal{NEX}TONE''$, ensuring that $Nxt_z(C'') = 1$, is defined as the conjunction of

$$\neg \text{code}_z'' \iff \bigvee_{e \in E_z} \text{en}_e''$$

and a constraint ‘computing’ en_e'' , which is similar to that for $\mathcal{NEX}TZERO'$. Now the next-state constraint can be expressed as $\mathcal{NEX}TZERO' \wedge \mathcal{NEX}TONE''$.

The CNF of $\mathcal{NEX}TZERO'$ is

$$\begin{aligned} & (\neg \text{code}_z' \vee \bigvee_{e \in E_z} \text{en}_e') \wedge \bigwedge_{e \in E_z} (\text{code}_z' \vee \neg \text{en}_e') \wedge \\ & \bigwedge_{e \in E_z} \left(\bigwedge_{f \in \bullet(\bullet e)} (\neg \text{en}_e' \vee \text{conf}_f') \wedge \bigwedge_{f \in (\bullet e) \bullet \setminus E_{cut}} (\neg \text{en}_e' \vee \neg \text{conf}_f') \wedge (\text{en}_e' \vee \bigvee_{f \in \bullet(\bullet e)} \neg \text{conf}_f' \vee \bigvee_{f \in (\bullet e) \bullet \setminus E_{cut}} \text{conf}_f') \right), \end{aligned}$$

and the CNF of $\mathcal{NEX}TONE''$ can be built similarly.

Translation to SAT. Finally, the problem of detection of CSC^z conflicts can be formulated as the SAT problem for the formula

$$CSC^z \stackrel{\text{def}}{=} CONF' \wedge CONF'' \wedge CODE' \wedge CODE'' \wedge \mathcal{NEX}TZERO' \wedge \mathcal{NEX}TONE'',$$

and the CSC problem is reduced to checking the CSC^z condition for each output signal z . In principle, the CSC problem can also be reduced to a *single* SAT instance [39], but according to our experiments the method presented here tends to be more efficient.

Computing All Cores. The method for resolution of CSC conflicts described in Section 5.2 requires to compute all conflict cores. This can be done by computing all the solutions of CSC^z for all output signals z using the incremental SAT approach. However, as the same complementary set can correspond to multiple conflict pairs, this approach is unnecessarily expensive. A better approach would be to eliminate all the solutions corresponding to a newly computed complementary set \mathcal{CS} each time it is computed, by appending new clauses to the formula. This can be done as follows. For each event $e \in E \setminus E_{cut}$ we create a variable cs_e , and the following constraint is added to the formula:

$$\left(\bigwedge_{e \in E \setminus E_{cut}} (\text{cs}_e \iff (\text{conf}_e' \oplus \text{conf}_e'')) \right) \wedge \bigvee_{e \in E \setminus E_{cut}} \begin{cases} \neg \text{cs}_e & \text{if } e \in \mathcal{CS} \\ \text{cs}_e & \text{otherwise.} \end{cases}$$

Note that the first part of this constraint is the same for all the computed complementary sets, and thus can be generated just once. The CNF of this constraint is

$$\begin{aligned}
 & (\neg \text{conf}'_e \vee \text{conf}''_e \vee \text{cs}_e) \wedge (\text{conf}'_e \vee \neg \text{conf}''_e \vee \text{cs}_e) \wedge \\
 & (\text{conf}'_e \vee \text{conf}''_e \vee \neg \text{cs}_e) \wedge (\neg \text{conf}'_e \vee \neg \text{conf}''_e \vee \neg \text{cs}_e) \wedge \bigvee_{e \in E \setminus E_{\text{cut}}} \begin{cases} \neg \text{cs} & \text{if } e \in \mathcal{CS} \\ \text{cs} & \text{otherwise} \end{cases} .
 \end{aligned}$$

The Case of Prefixes without Structural Conflicts. In many cases the performance of the proposed method can be improved by exploiting specific properties of the Petri net underlying an STG Γ . For instance, if Γ is free from dynamic choices (in particular, this is the case for marked graphs) then the union of any two configurations of its unfolding is also a configuration. This observation can be used to reduce the search space. Indeed, according to Proposition 2 below, it is then enough to look only for those cases when the configurations C' and C'' being tested are ordered in the set-theoretical sense.

Proposition 2 ([39]). *Let $\langle C', C'' \rangle$ be a conflict pair of configurations in the unfolding of a consistent STG Γ satisfying $C' \not\subseteq C''$, $C'' \not\subseteq C'$ and $C' \cup C''$ is a configuration. Then $C \stackrel{\text{df}}{=} C' \cap C''$ is such that either $\langle C, C' \rangle$ or $\langle C, C'' \rangle$ is a conflict pair.*

Note that freeness from structural conflicts can easily be detected: it is enough to check that $|b^\bullet| \leq 1$, for all conditions b of the prefix.

Since we do not know in advance whether $C' \subseteq C''$ or $C'' \subseteq C'$ (and the order does matter because the suggested implementation of the next-state constraint breaks the symmetry), a new Boolean variable, v_\subseteq , is introduced. If its value is 1 then the former possibility is checked, otherwise the latter possibility is tried out. This is captured by the constraint

$$\bigwedge_{e \in E \setminus E_{\text{cut}}} ((v_\subseteq \rightarrow (\text{conf}'_e \rightarrow \text{conf}''_e)) \wedge (\neg v_\subseteq \rightarrow (\text{conf}''_e \rightarrow \text{conf}'_e))) ,$$

which should be added to the formula. Note that it can easily be transformed into the CNF by applying the rule $x \rightarrow y \equiv \neg x \vee y$.

Experimental Results. We implemented our method using the zCHAFF SAT solver [48]. All the experiments were conducted on a PC with a *Pentium*TM IV/2.8GHz processor and 512M RAM.

A few classes of benchmarks have been attempted (the STGs with names containing the occurrence of ‘CSC’ satisfy the CSC property, the others exhibit CSC conflicts). The first group of examples comes from the real design practice. They are as follows:

- LAZYRING and RING – Asynchronous Token Ring Adapters described in [10, 44]. LAZYRINGCSC and RINGCSC have been obtained by resolving CSC conflicts in these test cases.
- DUP4PH, DUP4PHCSC, DUP4PHMTR, DUP4PHMTRCSC, DUPMTRMOD, DUPMTRMODUTG, and DUPMTRMODCSC – control circuits for the Power-Efficient Duplex Communication System described in [28].

Table 4. Experimental results: checking CSC.

Problem	Net			States	Prefix			Time, [s]		
	S	T	In/Out		B	E	E _{cut}	PfY	CLP	SAT
Real-Life STG s										
LAZYRING	35	32	5/6	160	87	66	5	1	<1	<1
LAZYRINGCsc	42	37	5/7	187	88	71	5	1	<1	<1
RING	147	127	11/17	16508	763	498	59	694	<1	<1
RINGCsc	185	172	11/18	16320	650	484	55	837	15	<1
DUP4PH	133	123	12/15	169	144	123	11	13	<1	<1
DUP4PHCsc	135	123	12/15	171	146	123	11	13	<1	<1
DUP4PHMTR	109	96	10/12	121	117	96	8	8	<1	<1
DUP4PHMTRCsc	114	105	10/16	149	122	105	8	9	<1	<1
DUPMTRMod	129	100	10/11	345	199	132	10	89	<1	<1
DUPMTRModUTG	116	165	10/11	323	344	218	65	286	<1	<1
DUPMTRModCsc	152	115	10/17	321	228	149	13	116	<1	<1
CfSymCscA	85	60	8/14	6672	1341	720	56	153	16	2
CfSymCscB	55	32	8/8	690	160	71	6	6	<1	<1
CfSymCscC	59	36	8/10	2416	286	137	10	11	<1	<1
CfSymCscD	45	28	4/10	414	120	54	6	3	<1	<1
CfAsymCscA	128	112	8/26	147684	1808	1234	62	1551	439	11
CfAsymCscB	128	112	8/24	147684	1816	1238	62	2602	471	10
Marked Graphs										
PpWk(2,3)	23	14	0/7	$5 \cdot 2^5 = 160$	41	23	1	<1	<1	<1
PpWk(2,6)	47	26	0/13	$5 \cdot 2^5 = 10240$	119	62	1	5	<1	<1
PpWk(2,9)	71	38	0/19	$5 \cdot 2^5 > 6 \cdot 10^5$	233	119	1	43	<1	<1
PpWk(2,12)	95	50	0/25	$5 \cdot 2^5 > 4 \cdot 10^7$	383	194	1	494	1	<1
PpWkCsc(2,3)	24	14	0/7	$2^7 = 128$	38	20	1	<1	<1	<1
PpWkCsc(2,6)	48	26	0/13	$2^{13} = 8192$	110	56	1	4	<1	<1
PpWkCsc(2,9)	72	38	0/19	$2^{19} > 5 \cdot 10^5$	218	110	1	43	3	<1
PpWkCsc(2,12)	96	50	0/25	$2^{25} > 3 \cdot 10^7$	362	182	1	2076	264	<1
PpWk(3,3)	34	20	0/10	$13 \cdot 2^7 = 1664$	63	35	1	1	<1	<1
PpWk(3,6)	70	38	0/19	$13 \cdot 2^{16} > 8 \cdot 10^5$	183	95	1	103	<1	<1
PpWk(3,9)	106	56	0/28	$13 \cdot 2^{25} > 4 \cdot 10^8$	357	182	1	2121	12	<1
PpWk(3,12)	142	74	0/37	$13 \cdot 2^{34} > 2 \cdot 10^{11}$	585	296	1	mem	1031	<1
PpWkCsc(3,3)	36	20	0/10	$2^{10} = 1024$	57	29	1	1	<1	<1
PpWkCsc(3,6)	72	38	0/19	$2^{19} > 5 \cdot 10^5$	165	83	1	44	3	<1
PpWkCsc(3,9)	108	56	0/28	$2^{28} > 2 \cdot 10^8$	327	164	1	7936	2285	<1
PpWkCsc(3,12)	144	74	0/37	$2^{37} > 10^{11}$	543	272	1	mem	time	<1
STG s with Arbitration										
PpArb(2,3)	48	32	2/13	$291 \cdot 2^4 = 4656$	110	66	2	7	<1	<1
PpArb(2,6)	72	44	2/19	$291 \cdot 2^{10} > 2 \cdot 10^5$	218	120	2	57	<1	<1
PpArb(2,9)	96	56	2/25	$291 \cdot 2^{16} > 10^7$	362	192	2	1726	<1	<1
PpArb(2,12)	120	68	2/31	$291 \cdot 2^{22} > 10^9$	542	282	2	11493	<1	<1
PpArbCsc(2,3)	48	32	2/13	$207 \cdot 2^4 = 3312$	110	66	2	3	<1	<1
PpArbCsc(2,6)	72	44	2/19	$207 \cdot 2^{10} > 2 \cdot 10^5$	218	120	2	41	2	<1
PpArbCsc(2,9)	96	56	2/25	$207 \cdot 2^{16} > 10^7$	362	192	2	316	153	<1
PpArbCsc(2,12)	120	68	2/31	$207 \cdot 2^{22} > 8 \cdot 10^8$	542	282	2	mem	12745	<1
PpArb(3,3)	71	48	3/19	$1647 \cdot 2^6 > 10^5$	188	114	3	97	<1	<1
PpArb(3,6)	107	66	3/28	$1647 \cdot 2^{15} > 5 \cdot 10^7$	368	204	3	1726	<1	<1
PpArb(3,9)	143	84	3/37	$1647 \cdot 2^{24} > 2 \cdot 10^{10}$	602	321	3	mem	<1	<1
PpArb(3,12)	179	102	3/46	$1647 \cdot 2^{33} > 10^{13}$	890	465	3	mem	<1	<1
PpArbCsc(3,3)	71	48	3/19	$297 \cdot 2^8 = 76032$	118	114	3	43	1	<1
PpArbCsc(3,6)	107	66	3/28	$297 \cdot 2^{17} > 3 \cdot 10^7$	368	204	3	1186	379	<1
PpArbCsc(3,9)	143	84	3/37	$297 \cdot 2^{26} > 10^{10}$	602	321	3	27512	time	<1
PpArbCsc(3,12)	179	102	3/46	$297 \cdot 2^{35} > 10^{13}$	890	465	3	mem	time	<1

- CFSYMCSCA, CFSYMCSCB, CFSYMCSCC, CFSYMCSCD, CFASYMCSCA, and CFASYMCSCB – control circuits for the Counterflow Pipeline Processor described in [72].

Some of these STGs, although built by hand, are quite large in size. The results for this group are summarised in the first part of Table 4. Two other groups, PPWK(m, n) and PPARB(m, n), contain scalable examples of STGs modelling m pipelines weakly synchronised without arbitration (in PPWK(m, n)) and with arbitration (in PPARB(m, n)). (See [40] for a more detailed description.) The former offers the possibility of studying the effect of the optimisation described in Section 5.6 (all STGs in the PPWK(m, n) series are marked graphs, and so their prefixes contain no structural conflicts). These benchmarks come in pairs: for each test case satisfying the CSC property there is a very similar one exhibiting CSC conflicts. This allowed us to test the algorithm on almost identical specifications with and without encoding conflicts. The results for these two groups are summarised in the last two parts of Table 4.

The meaning of the columns is as follows (from left to right): the name of the problem; the number of places, transitions, and input and output signals in the original STG; the number of conditions, events and cut-off events in the complete prefix; the number of reachable states in the STG; the time spent by a special version of the PETRIFY tool, which did not attempt to resolve the encoding conflicts it had identified; the time spent by the integer programming algorithm proposed in [38]; and the time spent by the proposed method. We use ‘mem’ if there was a memory overflow and ‘time’ to indicate that the test had not stopped after 15 hours. We have not included in the table the time needed to build complete prefixes, since it did not exceed 0.1sec for all the attempted STGs.

Although performed testing was limited in scope, one can draw some conclusions about the performance of the proposed algorithm. In all cases the proposed method solved the problem relatively easily, even when it was intractable for the other approaches. In some cases, it was faster by several orders of magnitude. The time spent on all of these benchmarks was quite satisfactory – it took just 11 seconds to solve the hardest one. Overall, the proposed approach was the best, especially for hard problem instances.

5.7 Logic Synthesis Based on Unfolding Prefixes

In Section 5.6, the CSC conflict detection problem was solved by reducing it to SAT. More precisely, given a finite and complete prefix of an STG’s unfolding, one can build for each output signal z a formula CSC^z which is satisfiable iff there is a CSC^z conflict. Here we modify that construction in the way described below. We assume a given consistent STG satisfying the CSC property, and consider in turn each output signal z .

Let C' and C'' be two configurations of the unfolding of a consistent STG, z be an output signal, and X is some set of signals. C' and C'' are in *Complete State Coding conflict* for z w.r.t. X (CSC_X^z conflict) if $Code_x(C') = Code_x(C'')$ for all

$x \in X$ and $Nxt_z(C') \neq Nxt_z(C'')$. The notion of CSC_X^z is a generalisation of the notion of CSC^z conflict (indeed, the latter can be obtained from the former by choosing X to be the set of all signals in the STG). X is a *support* of an output signal z if no two configurations of the unfolding are in CSC_X^z conflict. In such a case the next-state value of z at each reachable state of the STG is determined without ambiguity by the encoding of this state restricted to X , i.e., z can be implemented as a gate with the support X . A support X of z is *minimal* if no set $X' \subset X$ is a support of z . In general, a signal can have several distinct minimal supports.

The starting point of the proposed approach is to consider the set \mathcal{NSUPP}^z of all sets of signals which are *non-supports* of z . Within the Boolean formula CSC_{nsupp}^z , which we are going to construct, non-supports are represented by variables $\text{nsupp} \stackrel{\text{def}}{=} \{\text{nsupp}_x \mid x \in Z\}$, and, for a given assignment A , the set of signals $X = \{x \mid A(\text{nsupp}_x) = 1\}$ is identified with the projection $A|_{\text{nsupp}}$. The key property of CSC_{nsupp}^z is that $\mathcal{NSUPP}^z = \text{Proj}_{\text{nsupp}}^{CSC_{\text{nsupp}}^z}$, and so it is possible to use the incremental SAT approach to compute \mathcal{NSUPP}^z . However, for our purposes it will be enough to compute the maximal non-supports $\mathcal{NSUPP}_{\text{max}}^z \stackrel{\text{def}}{=} \max_{\subseteq} \mathcal{NSUPP}^z$ which can then be used for computing the set

$$\text{SUPP}_{\text{min}}^z \stackrel{\text{def}}{=} \min_{\subseteq} \{X \subseteq Z \mid X \not\subseteq X', \text{ for all } X' \in \mathcal{NSUPP}_{\text{max}}^z\}$$

of all the minimal supports of z (another incremental SAT run will be needed for this).

$\text{SUPP}_{\text{min}}^z$ captures the set of all possible supports of z , in the sense that any support is an extension of some minimal support, and vice versa, any extension of any minimal support is a support. However, the simplest equation is usually obtained for some minimal support, and this approach was adopted in our experiments. Yet, this is not a limitation of our method as one can also explore some or all of the non-minimal supports, which can be advantageous, e.g., for small circuits and/or when the synthesis time is not of paramount importance (this would sometimes allow to find a simpler equation). On the other hand, not all minimal supports have to be explored: if some minimal support has many more signals compared with another one, the corresponding equation will almost certainly be more complicated, and so too large supports can safely be discarded. Thus, as usual, there is a trade-off between the execution time and the degree of design space exploration, and our method allows one to choose an acceptable compromise. Typically, several ‘most promising’ supports are selected, the equations expressing Nxt_z as a function of signals in these supports are obtained (as described below), and the simplest among them is implemented as a logic gate.

Suppose now that X is one of the chosen supports of z . In order to derive an equation expressing Nxt_z as a function of the signals in X , we build a Boolean formula \mathcal{EQN}_X^z which has a variable code_x for each signal $x \in X$ and is satisfiable iff these variables can be assigned values in such a way that there is a configuration C in the prefix such that $\text{Code}_x(C) = \text{code}_x$, for all $x \in X$. Now, using the incremental SAT approach one can compute the projection of the set of reachable encodings onto X (differentiating the stored solutions according to

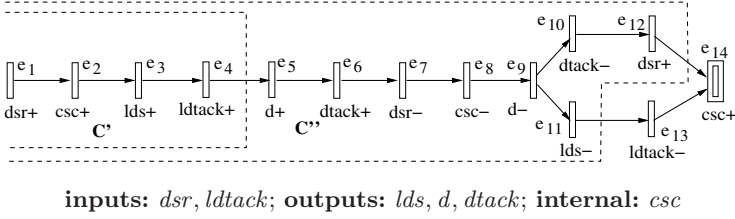


Fig. 23. An STG unfolding illustrating a $CSC_{\{dsr, ldtack\}}^{csc}$ conflict between configurations C' and C'' . Note that e_{14} is not enabled by C'' (since $e_{13} \notin C''$), and thus $Nxt_{csc}(C') = 1 \neq Nxt_{csc}(C'') = 0$. The order of signals in the binary encodings is: $dsr, ldtack, dtack, lds, d, csc$.

the value of the next-state function for z), and feed the result to a Boolean minimiser.

To summarise, the proposed method is executed separately for each output signal z and has three main stages: (I) computing the set \mathcal{NSUPP}_{\max}^z of maximal non-supports of z ; (II) computing the set \mathcal{SUPP}_{\min}^z of minimal supports of z ; and (III) deriving an equation for a chosen support X of z . In the sequel, we describe each of these three stages in more detail.

It should be noted that the size of the truth table for Boolean minimisation and the number of times a SAT solver is executed in our method can be exponential in the number of signals in the support. Thus, it is crucial for the performance of the proposed algorithm that the support of each signal is relatively small. However, in practice it is anyway difficult to implement as an atomic logic gate a Boolean expression depending on more than, say, eight variables. (Atomic behaviour of logic gates is essential for the speed-independence of the circuit, and a violation of this requirement can lead to hazards [14, 18].) This means that if an output signal has only ‘large’ supports then the specification must be changed (e.g., by adding new internal signals) to introduce ‘smaller’ supports. Such transformations are related to the *technology mapping* step in the design cycle for asynchronous circuits (see, e.g., [18]); we do not consider them here.

Computing Maximal Non-supports. Suppose that we want to compute the set of all maximal non-supports of an output signal z . At the level of a branching process, a CSC_X^z conflict can be represented as an unordered *conflict pair* of configurations $\langle C', C'' \rangle$ whose final states are in CSC_X^z conflict, as shown in Fig. 23.

As already mentioned, our aim is to build a Boolean formula CSC_{nsupp}^z such that $Proj_{\text{nsupp}}^{CSC_{\text{nsupp}}^z} = \mathcal{NSUPP}^z$, i.e., after assigning arbitrary values to the variables **nsupp**, the resulting formula is satisfiable iff there is a CSC_X^z conflict, where $X \stackrel{\text{df}}{=} \{x \mid \text{nsupp}_x = 1\}$.

The target formula CSC_{nsupp}^z is very similar to the formula CSC^z built in Section 5.6, with the following changes. For each signal $x \in Z$, instead of a vari-

able code_x we create two Boolean variables, code'_x and code''_x , tracing the values of $\text{Code}_x(C')$ and $\text{Code}_x(C'')$ respectively; \mathcal{CODE}' and \mathcal{CODE}'' are amended accordingly. Moreover, we create for each signal $x \in Z$ a variable nsupp_x indicating whether x belongs to a non-support.

Now we need to ensure that $\text{code}'_x = \text{code}''_x$ whenever $\text{nsupp}_x = 1$. This can be expressed by the following constraint:

$$\bigwedge_{x \in Z} \left(\text{nsupp}_x \Rightarrow (\text{code}'_x \iff \text{code}''_x) \right),$$

with the CNF

$$\bigwedge_{x \in Z} \left((\neg \text{code}'_x \vee \text{code}''_x \vee \neg \text{nsupp}_x) \wedge (\text{code}'_x \vee \neg \text{code}''_x \vee \neg \text{nsupp}_x) \right).$$

This completes the construction of $\mathcal{CSC}_{\text{nsupp}}^z$. For example, its satisfying assignment (except the variables cut' and cut'') for the $\mathcal{CSC}_{\{dsr, ldtack\}}^{\text{csc}}$ conflict depicted in Fig. 23 is as follows: $\text{conf}' = 111100000000$, $\text{conf}'' = 111111111110$, $\text{code}' = 110101$, $\text{code}'' = 110000$, $\text{nsupp} = 110000$, $\text{en}'_{e_2} = \text{en}'_{e_8} = \text{en}'_{e_{14}} = 0$, $\text{en}''_{e_2} = \text{en}''_{e_8} = \text{en}''_{e_{14}} = 0$.

Now the problem of computing the set $\mathcal{NSUPP}_{\text{max}}^z$ of maximal non-supports of z can now be formulated as a problem of finding the maximal elements of the projection $\text{Proj}_{\text{nsupp}}^{\mathcal{CSC}_{\text{nsupp}}^z}$. It can be solved using the incremental SAT approach, as described in Section 5.5.

Computing Minimal Supports. Let $\mathcal{NSUPP}_{\text{max}}^z$ be the set of maximal non-supports computed in the first stage of the method. Now we need to compute the set $\mathcal{SUPP}_{\text{min}}^z$ of the minimal supports of z . This can be achieved by computing the set of minimal assignments for the Boolean formula

$$\bigwedge_{\text{nsupp}^* \in \mathcal{NSUPP}_{\text{max}}^z} \left(\bigvee_{x \in Z: \text{nsupp}_x^* = 0} \text{supp}_x \right),$$

which is satisfied by an assignment A iff for all maximal non-supports nsupp^* in $\mathcal{NSUPP}_{\text{max}}^z$, $A \not\leq \text{nsupp}^*$. This again can be done using the incremental SAT approach, as described in Section 5.5. Note that this Boolean formula is much smaller than that for the first stage of the method (it contains at most $|Z|$ variables), and thus the corresponding incremental SAT problem is much simpler.

Deriving an Equation. Suppose that X is a (not necessarily minimal) support of z . We need to express Nxt_z as a Boolean function of signals in X . This can be done by generating a truth table for z as a Boolean function of signals in X , and then applying Boolean minimisation.

The set of encodings appearing in the first column of the truth table coincides with the projections of the formula

$$\mathcal{EQN}_X^z \stackrel{\text{df}}{=} \text{CONF}' \wedge \text{CODE}'_X$$

onto the set of variables $\{\text{code}_x \mid x \in X\}$, where CODE'_X is CODE' restricted to the set of signals X (i.e., all the conjunctions of the form $\bigwedge_{x \in Z} \dots$ are replaced by $\bigwedge_{x \in X} \dots$). It also can be computed using the incremental SAT approach, as described in Section 5.5. Note that at each step of this computation, the SAT solver returns information not only about the next element of the projection, but also the values of all the other variables in the formula. That is, along with the restriction of some reachable encoding onto the set X we have an information about a configuration C via which it can be reached. Thus, the value of Nxt_z on this element of the projection can be computed simply as $\text{Nxt}_z(C)$. This essentially completes the description of our method.

Optimisations. In [40] we describe optimisations which can significantly reduce the computation effort required by our method. In particular, we suggest a heuristic helping to compute a part of a signal's support without running the SAT solver, based on the fact that any support for an output z must include all the *triggers* of z , i.e., those signals whose firing can enable z . (The information about triggers can be derived from the finite and complete prefix.) Moreover, one can speed up the computation in the case of prefixes without structural conflicts, as described in Section 5.6.

Experimental Results. We implemented our method using the zCHAFF SAT solver [48] and the ESPRESSO Boolean minimiser [5], and the benchmarks from Section 5.6 satisfying the CSC property were attempted. All the experiments were conducted on a PC with a *Pentium*TM IV/2.8GHz processor and 512M RAM.

The experimental results are summarised in Table 5, where the meaning of the columns is as follows: the total number of equations obtained by our method (this is equal to the total number of minimal supports for all the output signals and gives a rough idea of the explored design space); the time spent by the PETRIFY tool; and the time spent by the proposed method. We use ‘mem’ if there was a memory overflow and ‘time’ to indicate that the test had not stopped after 15 hours. (Table 4 provides additional data about the benchmarks.)

Although the performed testing was limited in scope, one can draw some conclusions about the performance of the proposed algorithm. In all cases the proposed method solved the problem relatively easily, even when it was intractable for PETRIFY. In some cases, it was faster by several orders of magnitude. The time spent on all these benchmarks was quite satisfactory – it took less than 50 seconds to solve the hardest one (CFASYMCSCA); note however, that in that case a total of 450 equations were obtained, i.e., more than 9 equations per second.

Table 5. Experimental results.

<i>Real-Life STG s</i>				
Problem	Eqns (SAT)	Time, [s]		
		PfY	SAT	
LAZYRINGCsc	14	1	<1	
RINGCsc	63	850	3	
DUP4PHCsc	48	20	<1	
DUP4PHMTRCsc	46	13	<1	
DUPMTRMODCsc	165	125	1	
CfSYMCscA	60	163	16	
CfSYMCscB	34	10	<1	
CfSYMCscC	18	13	<1	
CfSYMCscD	16	3	<1	
CfASYMCscA	450	1448	48	
CfASYMCscB	93	2323	17	

<i>Marked Graphs</i>			
Problem	Eqns (SAT)	Time, [s]	
		PfY	SAT
PpWkCsc(2,3)	7	<1	<1
PpWkCsc(2,6)	13	4	<1
PpWkCsc(2,9)	19	44	<1
PpWkCsc(2,12)	25	2082	<1
PpWkCsc(3,3)	10	1	<1
PpWkCsc(3,6)	19	43	<1
PpWkCsc(3,9)	28	7380	<1
PpWkCsc(3,12)	37	time	1

<i>STG s with Arbitration</i>			
Problem	Eqns (SAT)	Time, [s]	
		PfY	SAT
PpArBCsc(2,3)	18	4	<1
PpArBCsc(2,6)	24	42	<1
PpArBCsc(2,9)	30	315	<1
PpArBCsc(2,12)	36	3840	1
PpArBCsc(3,3)	29	45	<1
PpArBCsc(3,6)	38	1001	<1
PpArBCsc(3,9)	47	24941	1
PpArBCsc(3,12)	56	mem	2

It is important to note that these improvements in memory and running time come *without any reduction in quality of the solutions*. In fact, our method is *complete*, i.e., it can produce all the valid complex-gate implementations of each signal. However, in our implementation we restricted the algorithm to only minimal supports. Nevertheless, the explored design space was quite satisfactory: as the ‘Eqns’ column in Table 5 shows, in many cases our method proposed quite a few alternative implementations for signals. In fact, among the list of solutions produced by our tool there was always a solution produced by PETRIFY (with, perhaps, only minor differences due to the non-uniqueness of the result of Boolean minimisation). Overall, the proposed approach turned out to be clearly superior, especially for hard problem instances.

5.8 Conclusion and Future Work

We have proposed a complex-gate design flow for asynchronous circuits based on STG unfolding prefixes comprising: (i) a SAT-based algorithm for detection of encoding conflicts; (ii) a framework for visualisation and resolution of encoding

conflicts; and (iii) an algorithm for derivation of Boolean equations for the gates implementing the circuit based on the incremental SAT approach.

Note that in all the test cases (Table 4) the size of the complete prefix was relatively small. This can be explained by the fact that STGs usually contain a lot of concurrency but relatively few choices, and thus the prefixes are in many cases not much bigger than the STGs themselves. For the scalable benchmarks, one can observe that the complete prefixes exhibited polynomial (in fact, quadratic) growth, whereas the number of reachable states grew exponentially. As a result, the unfolding-based method had a clear advantage over that based on state graphs, both in terms of memory usage and running time. The experimental results demonstrated that the devised algorithms could handle quite large specifications in relatively short time, obtaining high-quality solutions. Moreover, the proposed approach is applicable to all bounded Petri nets, without any structural restrictions such as Marked Graph of Free-Choice constraint.

An important observation one can make is that the combination ‘unfolder & solver’ turns out to be quite powerful. It has already been used in a number of papers (see, e.g., [30, 38]). Most of ‘interesting’ problems for safe Petri nets are *PSPACE*-complete [23], and unfolding such a net allows to reduce this complexity class down to *NP* (or even *P* for some problems, e.g., checking consistency). Though in the worst case the size of a finite and complete unfolding prefix can be exponential in the size of the original Petri net, in practice it is often relatively small. In particular, according to our experiments, this is almost always the case for STGs. A problem formulated for a prefix can usually be translated into some canonical problem, e.g., an integer programming one [38], a problem of finding a stable model of a logic program [30], or SAT as here. Then an appropriate solver can be used for efficiently solving it.

The presented framework for interactive refinement aimed at resolution of encoding conflicts is based on the visualisation of conflict cores, which are sets of transitions ‘causing’ state encoding conflicts. Cores are represented at the level of the unfolding prefix, which is a convenient model for understanding the behaviour of the system due to its simple branching structure and acyclicity.

The advantage of using cores is that only those parts of STGs which cause encoding conflicts, rather than the complete list of CSC conflicts, are considered. Since the number of cores is usually much smaller than the number of encoding conflicts, this approach saves the designer from analysing large amounts of information. Resolution of encoding conflicts requires the elimination of cores by introducing additional signals into the STG. The refinement contains several interactive steps aimed at helping the designer to obtain a customised solution. The case studies demonstrate the positive features of the interactive refinement process.

Heuristics for signal insertion based on the height map and exploiting the intersections of cores use the most essential information about encoding conflicts, and thus should be quite efficient. In fact, the conflict resolution procedure can be automated either partially or completely. However, in order to obtain an optimal solution, a semi-automated resolution process should be employed. For

example, the tool might suggest the areas for insertion of new signal transitions, which are to be used as guidelines. Yet, the designer is free to intervene at any stage and choose an alternative location, in order to take into account the design constraints.

We view these results as encouraging. In future work we intend to include also the technology mapping step into the described design flow, as well as incorporate other methods for resolving encoding conflicts (concurrency reduction [17], timing assumption [18], etc.) into the proposed framework for visualisation and resolution of encoding conflicts.

6 Other Related Work and Future Directions

There has been a large amount of research in hardware design using Petri nets in the last few years. This chapter has covered only the main advances made recently in logic synthesis from STGs, and some of them, such as the topic of STG decomposition, only briefly.

The reader is however encouraged to look broader and for that we briefly list here a number of relevant and interesting developments.

- **STG Decomposition.** The idea of reducing complexity in logic synthesis from STG by STG decomposition is not new. For example, in [14] the contraction method was introduced in which the logic equations for output signal were derived from the projections of the state graph on the set of relevant signals forming the support of the derived function. This idea has been recently developed further in [71], in order to remove some restrictions on the class of the STG (live and safe free choice). It also approaches the decomposition problem in a powerful equivalence framework, which is a bisimulation with angelic nondeterminism. Another attempt in this direction, perhaps in a more practical context of the HDL-based design flow was recently reported in [77].
- **Implementability Checking in Polynomial Time.** Another important source of complexity reduction is a search for polynomial algorithms for various stages in asynchronous logic synthesis for restricted STG classes, in particular for free-choice nets. Such an algorithm has been developed in [24].
- **Optimisation in Direct Mapping from STG.** The advantages of the direct mapping of Petri nets to circuits can be exploited in the STG level, although at extra cost in circuit area. The direct mapping does not however affect performance negatively. In fact in many cases, direct mapping offers solutions where the latency between input and output signal transitions is minimal. New techniques of translating STGs into circuits using David cells, structured into ‘tracker’ and ‘bouncer’, also include optimisation of the logic size [61].
- **Synthesis from STGs in Restricted Bases.** While logic synthesis of speed-independent circuits in complex gates provides a satisfactory solution for modern CMOS design technologies, in the future it may not be reliable enough to guarantee correct operation. The effects of delays in wires

and parametric instabilities may require a much more conservative approach to the implementation of control circuits. In this respect, advances in the synthesis of circuits that are monotonic [62], i.e., having no “zero-delay” inverters on the inputs, and free from isochronic forks [63] are important.

- **Synthesis with Relative Timing Assumptions.** Unlike the above, sometimes designing circuits under conservative assumptions can lead to significant wastage of area, speed and power. More optimistic considerations can be made about delays in the system, for example based on the knowledge of actual delays in the data path or in the environment, or due to information about relative speeds of system components. Use of relative timing has been investigated under the notion of lazy transition systems [15].
- **Synthesis from Delay-Insensitive Process Specifications.** A potential way to automatic compilation of HDLs based on communicating processes to asynchronous circuits may be via an important semantical link between delay-insensitive (DI) process algebras and Petri nets. Such a link has been established and developed to the level of tool support in [33]. A particularly interesting contribution has been the definition of the DI process decomposition which helps avoiding CSC conflicts in the STG that is constructed automatically from a process-algebraic specification [34].

Acknowledgements

We would like to thank Alex Bystrov, Michael Kishinevsky, Alex Kondratyev, Maciej Koutny, Luciano Lavagno and Agnes Madalinski for contributing to this research at various stages. This research was partially supported by EU Framework 5 ACiD-WG and EPSRC grants GR/M99293, GR/M94366 (MOVIE) and GR/R16754 (BESST).

References

1. A. Allan, et. al., 2001 Technology Roadmap for Semiconductors, *Computer*, January 2002, pp. 42-53.
2. K. van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of International Series on Parallel Computation. Cambridge University Press, 1993.
3. E. Best and B. Grahlmann. PEP – more than a Petri Net Tool. *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS’96)*, Springer-Verlag, Lecture Notes in Computer Science 1055, Springer-Verlag (1996) 397-401.
4. I. Blunno and L. Lavagno. Automated synthesis of micro-pipelines from behavioral VERILOG HDL, *Proc. of IEEE Symp. on Adv. Res. in Async. Cir. and Syst. (ASYNC 2000)*, IEEE CS Press, pp. 84–92.
5. R. Brayton, G. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli: *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers (1984).
6. A. Bystrov and A. Yakovlev. Asynchronous Circuit Synthesis by Direct Mapping: Interfacing to Environment, *Proc. ASYNC’02*, Manchester, April 2002.

7. J. Carmona and J. Cortadella. Input/Output Compatibility of Reactive Systems. In *Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Portland, Oregon, USA, November 2002. Springer-Verlag.
8. J. Carmona and J. Cortadella. ILP Models for the Synthesis of Asynchronous Control Circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, San Jose, California, USA, November 2003.
9. J. Carmona, J. Cortadella, and E. Pastor. A structural encoding technique for the synthesis of asynchronous circuits. *Fundamenta Informaticae*, pages 135–154, April 2001.
10. C. Carrion and A. Yakovlev: Design and Evaluation of Two Asynchronous Token Ring Adapters. Tech. Rep. CS-TR-562, School of Comp. Sci., Univ. of Newcastle (1996).
11. Daniel M. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.
12. T. Chelcea, A. Bardsley, D. Edwards and S.M. Nowick. A burst-mode oriented back-end for the Balsa synthesis system, *Proc. of Design, Automation and Test in Europe (DATE'02)*, IEEE CS Press, pp. 330-337.
13. T.-A. Chu, C. K. C. Leung, and T. S. Wanuga. A design methodology for concurrent VLSI systems. In *Proc. International Conf. Computer Design (ICCD)*, pages 407-410. IEEE Computer Society Press, 1985.
14. T.-A. Chu: *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD Thesis, MIT/LCS/TR-393 (1987).
15. J. Cortadella, M.Kishinevsky, S.M. Burns, K.S. Stevens, A. Kondratyev, L. Lavagno, A. Taubin, A. Yakovlev. Lazy Transition Systems and Asynchronous Circuit Synthesis with Relative Timing Assumptions. *IEEE Trans. of CAD*, Vol. 21, No. 2, Feb. 2002, pages 109-130.
16. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev: A Region-Based Theory for State Assignment in Speed-Independent Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16(8) (1997) 793–812.
17. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev: Automatic Handshake Expansion and Reshuffling Using Concurrency Reduction. *Proc. of HWPN'98*, (1998) 86–110.
18. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev: *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer Verlag (2002).
19. W. J. Dally and J. W. Poulton: *Digital Systems Engineering*. Cambridge University Press (1998).
20. R. David. Modular design of asynchronous circuits defined by graphs. *IEEE Transactions on Computers*, 26(8):727–737, August 1977.
21. J. Desel and J. Esparza. Reachability in cyclic extended free-choice systems. *TCS 114, Elsevier Science Publishers B.V.*, 1993.
22. D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12-18, 2002.
23. J. Esparza: Decidability and Complexity of Petri Net Problems – an Introduction. In: *Lectures on Petri Nets I: Basic Models*, W. Reisig and G. Rozenberg (Eds.). LNCS 1491 (1998) 374–428.
24. J. Esparza. A Polynomial-Time Algorithm for Checking Consistency of Free-Choice Signal Transition Graphs, *Proc. of the 3rd Int. Conf. Applications of Concurrency to System Design (ACSD'03)*, IEEE CS Press, June 2003, pp. 61-70.
25. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. *FMSD* 20(3) (2002) 285–310.

26. D. Ferguson and M. Hagedorn, The Application of NULL Convention Logic to Microcontroller/Microconverter Product, Second ACiD-WG Workshop, Munich, 2002. URL: <http://www.scism.sbu.ac.uk/ccsv/ACiD-WG/Workshop2FP5/Programme/>.
27. S. Furber, Industrial take-up of asynchronous design, Keynote talk at the Second ACiD-WG Workshop, Munich, 2002. URL: <http://www.scism.sbu.ac.uk/ccsv/ACiD-WG/Workshop2FP5/Programme/>.
28. S. B. Furber, A. Efthymiou, and M. Singh: A Power-Efficient Duplex Communication System. Proc. of *AINT'00*, TU Delft, The Netherlands (2000) 145–150.
29. F. García Vallés and J.M. Colom. Structural analysis of signal transition graphs. In D. Holdt In B. Farwer and M.O. Stehr, editors, *Proceedings of the Workshop Petri Nets in System Engineering (PNSE'97). Modelling, Verification and Validation*, pages 123–134, Hamburg (Germany). September 25–26, September 1997. Published as report n 205 of the Computer Science Department of the University of Hamburg.
30. K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. *Fundamentae Informaticae* 37(3) (1999) 247–268.
31. K. Heljanko, V. Khomenko, and M. Koutny: Parallelization of the Petri Net Unfolding Algorithm. Proc. of *TACAS'2002*, LNCS 2280 (2002) 371–385.
32. L.A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.
33. H.K. Kapoor, M.B. Josephs and D.P. Furey: Verification and Implementation of Delay-Insensitive Processes in Restrictive Environments. Proc. of *ICACSD'04*, IEEE Comp. Soc. Press (2004) to appear.
34. H.K. Kapoor and M.B. Josephs: Automatically decomposing specifications with concurrent outputs to resolve state coding conflicts in asynchronous logic synthesis. Proc. of *DAC'04*, 2004 (to appear).
35. V. Khomenko and M. Koutny: LP Deadlock Checking Using Partial Order Dependencies. Proc. of *CONCUR'2000*, LNCS 1877 (2000) 410–425.
36. V. Khomenko and M. Koutny: Towards An Efficient Algorithm for Unfolding Petri Nets. Proc. of *CONCUR'2001*, LNCS 2154 (2001) 366–380.
37. V. Khomenko, M. Koutny, and V. Vogler: Canonical Prefixes of Petri Net Unfoldings. Proc. of *CAV'2002*, LNCS 2404 (2002) 582–595. Full version: *Acta Informatica* 40(2) (2003) 95–118.
38. V. Khomenko, M. Koutny and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Proc. of *DATE'02*, IEEE Comp. Soc. Press (2002) 338–345.
39. V. Khomenko, M. Koutny, and A. Yakovlev: Detecting State Coding Conflicts in STG Unfoldings Using SAT. Proc. of *ICACSD'03*, IEEE Comp. Soc. Press (2003) 51–60. Full version: to appear in Special Issue on Best Papers from ICACSD'2003, *Fundamenta Informaticae*.
40. V. Khomenko, M. Koutny, and A. Yakovlev: Logic Synthesis Avoiding State Space Explosion. Proc. of *ICACSD'04*, IEEE Comp. Soc. Press (2004) to appear. Full version: Tech. Rep. CS-TR-813, School of Comp. Science, Univ. of Newcastle. URL: <http://homepages.cs.ncl.ac.uk/victor.khomenko/home.formal/papers/papers.html>.
41. D. J. Kinniment, B. Gao, A. Yakovlev and F. Xia: Towards asynchronous A-D conversion. Proc. of *ASYNC'00*, IEEE Comp. Soc. Press (2000) 206–215.
42. Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.
43. A. Kondratyev and K. Lwin. Design of asynchronous circuits using synchronous CAD tools. *IEEE Design and Test of Computers*, 19(4):107–117, 2002.

44. K.S.Low and A.Yakovlev: Token Ring Arbiters: an Exercise in Asynchronous Logic Design with Petri Nets. Tech. Rep. CS-TR-537, School of Comp. Sci., Univ. of Newcastle (1995).
45. A.Madalinski, A.Bystrov, V.Khomenko, and A.Yakovlev: Visualisation and Resolution of Coding Conflicts in Asynchronous Circuit Design. Proc. of DATE'03, IEEE Comp. Soc. Press (2003) 926–931. Full version: Special Issue on Best Papers from DATE'2003, IEE Proceedings: Computers & Digital Techniques 150(5) (2003) 285–293.
46. K.L.McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of CAV'92, LNCS 663 (1992) 164–174.
47. G. De Micheli. *Synthesis and Optimisation of Digital Circuits*, McGraw-Hill, 1994.
48. S.Moskewicz, C.Madigan, Y.Zhao, L.Zhang and S.Malik: CHAFF: Engineering an Efficient SAT Solver. Proc. of DAC'01, ASME Technical Publishing (2001) 530–535.
49. T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.
50. E. Pastor, J. Cortadella, A. Kondratyev, and O. Roig. Structural methods for the synthesis of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 17(11):1108–1129, November 1998.
51. S.S. Patil and J.B. Dennis. The description and realization of digital systems. In *Proceedings of the IEEE COMPCON*, pages 223–226, 1972.
52. M.A. Peña and J. Cortadella, Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits, *Proc. of IEEE Symp. on Adv. Res. in Async. Cir. and Syst. (ASYNC'96)*, IEEE CS Press, pp. 222–232.
53. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn, Institut für Instrumentelle Mathematik, 1962. (technical report Schriften des IIM Nr. 3).
54. P. Riocreux: *Private communication*. UK Asynchronous Forum (2002).
55. L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.
56. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.
57. A.Semenov: *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding*. PhD Thesis, University of Newcastle upon Tyne (1997).
58. D. Shang, F. Xia and A. Yakovlev. Asynchronous Circuit Synthesis via Direct Translation, *Proc. Int. Symp. on Cir. and Syst. (ISCAS'02)*, Scottsdale, Arizona, May 2002.
59. Manuel Silva, Enrique Teruel, and José Manuel Colom. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:309–373, 1998.
60. J. Sparsø and S. Furber, Edt., *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001
61. D. Sokolov, A. Bystrov and A. Yakovlev. STG optimisation in the direct mapping of asynchronous circuits, Proc. Design and Test in Europe (DATE), March 2003, 932–937.
62. N. Starodoubtsev, S. Bystrov, M. Goncharov, I. Klotchkov and A. Smirnov. Towards Synthesis of Monotonic Circuits from STGs, In Proc. of 2nd Int. Conf. Applications of Concurrency to System Design (ACSD'01), IEEE CS Press, June 2001, pp. 179–180.
63. N. Starodoubtsev, S. Bystrov, and A. Yakovlev. Monotonic circuits with complete acknowledgement, Proc. of ASYNC'03, Vancouver, IEEE CS Press, pp. 98–108.

64. Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720-738, June 1989.
65. A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297-322, 1992.
66. P. Vanbekbergen. *Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications*. PhD thesis, Catholic University of Leuven, 1993.
67. P. Vanbekbergen, F. Catthoor, G. Goossens and H. De Man: Optimised Synthesis of Asynchronous Control Circuits form Graph-Theoretic Specifications. Proc. of *ICCAD'90*, IEEE Comp. Soc. Press (1990) 184-187.
68. V. I. Varshavsky and V. B. Marakhovsky. Asynchronous control device design by net model behavior simulation. In J. Billington and W. Reisig, editors, *Application and Theory of Petri Nets 1996*, volume 1091 of *Lecture Notes in Computer Science*, pages 497-515. Springer-Verlag, June 1996.
69. V. I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.
70. Thomas Villiger, Hubert Ksclin, Frank K. Grkaynak, Stephan Oetiker, and Wolfgang Fichtner. Self-timed ring for globally-asynchronous locally-synchronous systems. Proc. *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 141-150. IEEE Computer Society Press, May 2003.
71. W. Vogler and R. Wollowski. Decomposition in asynchronous circuit design. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *Lecture Notes in Computer Science*, pages 152-190. Springer-Verlag, 2002.
72. A. Yakovlev: Designing Control Logic for Counterflow Pipeline Processor Using Petri nets. *FMSD* 12(1) (1998) 39-71.
73. A. Yakovlev, S. Furber and R. Krenz, Design, Analysis and Implementation of a Self-timed Duplex Communication System, CS-TR-761, Dept. Computing Science, Univ. of Newcastle upon Tyne, March 2002. URL: http://www.cs.ncl.ac.uk/people/alex.yakovlev/home.informal/some_papers/duplex-TR.ps.
74. A. Yakovlev and A. 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.
75. A. Yakovlev and A. Petrov: Petri Nets and Asynchronous Bus Controller Design. Proc. of *ICATPN'90*, (1990) 244-262.
76. A. Yakovlev, V. Varshavsky, V. Marakhovsky and A. Semenov. Designing an asynchronous pipeline token ring interface, *Proc. of 2nd Working Conference on Asynchronous Design Methodologies, London, May 1995*, IEEE Comp. Society Press, N.Y., 1995, pp. 32-41.
77. T. Yoneda and C. Myers. Synthesis of Speed Independent Circuits based on Decomposition, *Proceedings of ASYNC 2004*, Heraklion, Greece, IEEE CS Press, April 2004.
78. L. Zhang and S. Malik: The Quest for Efficient Boolean Satisfiability Solvers. Proc. of *CAV'02*, LNCS 2404 (2002) 17-36.