HIGH LEVEL MODELLING AND DESIGN OF ASYNCHRONOUS INTERFACE LOGIC

A.V. Yakovlev[a] A.M. Koelmans[a] L.Lavagno[b]

[a]Department of Computing Science, University of Newcastle upon Tyne, UK
[b]Dipartimento di Elettronica, Politecnico di Torino, Italy

Computing Science, University of Newcastle upon Tyne
Technical Report Series No. 460, November 1993

Abstract

Asynchronous digital interface circuits exhibit a high degree of concurrency. Self-timed implementation is the most appropriate design discipline for such circuits. Their complexity demands that a formal design methodology, amenable to automation, is used to design them. Existing specification models suffer from severe limitations when it comes to describing the circuit function at a high level, which requires decomposing the specification into intercommunicating sub-modules and synthesizing a logic circuit implementation of that function. We propose a new methodology to design asynchronous circuits that is divided in two stages: abstract synthesis and logic synthesis. The first stage is carried out by refining an abstract model, based on logic predicates describing the correct input-output behavior of the circuit, into a labeled Petri net and then into a formalization of timing diagrams (the Signal Transition Graph). This refinement involves hierarchical decomposition of the initial implementation until its size can be handled by automated logic synthesis tools, as well as replacing symbolic events occurring on the input-output ports of the labeled Petri net with up and down transitions occurring on the input-output wires of a circuit implementation.

keywords Self-timed circuits, concurrency models, token-ring LAN adaptor, bus controller, FIFO buffer, trace theory, process, specification, discriminator, labelled Petri net, signalling expansion, signal transition graphs, implementation.

1 Introduction

Modern technologies allow the construction of VLSI circuits whose internal behaviour exhibits a high degree of parallelism. Such circuits tend to suffer from undesired phenomena such as electronic arbitration, metastability, and higher values of wire versus gate delay ratios, all of which can cause signal discrepancies, parametric instabilities, and other problems. In order to ensure that they operate correctly under the presence of such phenomena, their design is performed in *speed-independent* or *self-timed* fashion [36]. The generic name for such approaches has traditionally been *asynchronous design methodologies*, which includes any technique which produces circuits free from a global clock signal. Additional advantages of asynchronous circuits are higher modularity, which allows more efficient design maintenance and re-usability of components, and lower power consumption. Since there is a huge market for portable computing equipment, low power consumption is now considered to be a critical factor in favour of asynchronous circuits. It results from the fact that asynchronous designs do not pay a penalty for the distribution of clock signals, and that their behaviour mimics the paradigm of "lazy evaluation".

During the last few years a large number of formal techniques and software tools to synthesize asynchronous control and interface circuits have been developed, among others [35, 6, 26, 23, 50, 15, 19, 44, 17, 3, 31]. These techniques all differ in the way in which a circuit is modelled, specified, verified and synthesized. The question of modelling, both high-level and low-level, in both the behavioural and the structural domain, appears to be the key issue. We therefore devote part of our background section to this topic. Our main aim is, however, to demonstrate the use of formal models of concurrency for the synthesis of speed-independent circuit implementations from high-level, abstract, concurrent behavioural specifications, using the basic units of an interface adaptor structure as an example.

Petri nets, a well-known model for the dynamic behaviour of concurrent and asynchronous systems [30], have proven to be a useful tool for the specification of asynchronous control circuits. The explicit notion of conditions and events in Petri nets creates a good framework for defining the paradigms of the behaviour of a circuit: causality, parallelism, choice, and conflicts. The events of the net can be annotated with a specific interpretation, such as transitions of signals. This type of net has been named Signal Transition Graphs. Such graphs, introduced in [35, 6], and other closely related models (e.g., Change Diagrams [17]), have recently become very popular as a formalism to be used in the automated synthesis of asynchronous circuits [26, 19, 28, 38, 20, 16]. This is because of their descriptive simplicity and similarity to timing diagrams, a formalism widely used among circuit designers.

The key step in the development of Signal Transition Graphs is the interpretation of Petri net transitions as rising or falling edges of input and output signals of the specified circuit. The specification is *consistent* if the corresponding reachability graph[1] can be labeled with a string of signal values that match the transitions (e.g., a rising transition of a signal requires the signal to have value 0 in the predecessor and value 1 in the successor marking label).

A consistent labeled reachability graph (also called *State Graph*) can be directly implemented as an asynchronous circuit if and only if the signals specified by the STG completely describe the state of the circuit. Then the STG is said to have the Complete State Coding property. Otherwise, some adequate technique for adding such signals to an otherwise consistent specification must be applied [50, 44, 22].

Although existing automated synthesis tools provide good support in may cases, they are still limited in power. One of the problems is the complexity of the state graph representation of the semantics of signal transition graphs. The state graph unfolds all the true concurrency fragments into a form of interleaving, which makes both verification and circuit implementation rather complex. There have been some attempts to use pure concurrency semantics to solve the completeness problem [50, 44, 22], but these are restricted to special classes of Petri nets. Another partially resolved problem is synthesis of speed-independent circuits. So far, no universal technique has emerged to produce an efficient circuit for a given library of logical gates and latches. Some work in this domain has been reported in [46, 19, 28, 1, 53]. We do not claim any technical results in this area, and assume that the logic synthesis task is solved by any one of those approaches.

The primary subject of this paper is the link between the high-level, abstract synthesis and low-level, logical synthesis of asynchronous control circuits. We show that, by using the

[1] A graph with a node for each marking of the Petri net and an edge for each transition firing from a marking to another.

language of Petri nets at both stages, we can preserve the behavioural semantics. This is the crucial advantage of our methodology. At the same time, by allowing the modelling of the circuit behaviour at a level of abstraction that is higher than the signal transition graph, we look for better ways of capturing the control flow, which can be rather complex. For example, as will be shown later, some of the characteristics of the abstract behaviour can provide useful heuristics for solving the problem of Complete State Coding. The latter has been shown to be the most crucial problem in the process of ensuring the implementability of the initial specification [8, 20, 50, 44, 28, 22].

The paper is organised as follows[2]. We first present a brief review of a number of other models. We classify and evaluate them based on their capacity to describe certain types of self-timed modules and systems. None of those models satisfies our requirement of high-level causality-based modeling coupled with hierarchical decomposability. We then examine in greater detail the notion of the basic high-level object, whose interconnection describes the circuit function: the *discriminator*. A discriminator is composed by an outer behavioural specification, in terms of a characteristic predicate of the set of traces observed at its ports, together with an inner behavioural model, defined by a labelled Petri net. We discuss the Signal Transition Graph model and how it can be obtained from a labelled Petri net describing a given discriminator. We then present some important issues in analysing its semantics, that is, verification that a specification is consistent and complete. We describe the synthesis procedure from the Signal Transition Graph description. Throughout the paper we use a design example: a controller for a FIFO buffer of capacity k, which is part of a general structure for a typical interface adaptor for a LAN token ring architecture. From this we deduce that two different design strategies are to be applied to the FIFO buffer unit and bus controller unit.

2 Models for Self-timed Hardware Design

A self-timed system is usually defined as a collection of self-timed modules, or elements, which communicate through asynchronous protocols [36]. Such a system does not require a global clock signal. All system-level events are ordered in time by the causal relationships among the module actions. The order established by the designer must be preserved in the circuit that is eventually generated, thereby guaranteeing correct operation.

The pioneering work of D.E. Muller on speed-independent circuits [29, 27] suggests some useful formalisms for self-timed systems. For example, the Muller state diagram, can specify the concurrent switching behaviour of circuit subcomponents by using the excitation mechanism. The state of the circuit is defined by a string of values of its gate outputs. A gate is excited if its value in the label is different from the value of its associated Boolean function (e.g., a *nor* gate with output and inputs all at 0). An excited gate may change its output value, thus adding an edge between the corresponding labels to the state diagram. This model, though an excellent tool for theoretical analysis of circuit behaviour, is too complex to be used as a specification language due to its exponential size in the number of gates.

The revival of interest in self-timed systems has resulted in a number of more efficient notations, together with their formal characterisation. These are based on various modelling techniques at different levels of abstraction.

[2] Note for reviewers/editors. We did not include much background material, e.g. on asynchronous circuits, Signals Transition Graphs, etc., because we were assuming that a review paper would be part of the issue. If this is not the case, then we can easily put together a section reviewing the major results in the area and include it in the final version.

Low-level modelling At the lower levels, models can be classified into the following groups [49, 5]:

1. The delay model of an element (delay model 'in the small'), such as:
   - unbounded delay model, which itself can be:
     - pure, or transport, delay (every input change propagates to the output but "shifted" by d time units),
     - simple inertial delay (input pulses whose length is less than d are filtered out, while those longer than d appear at the output shifted by d units),
     - pure chaos delay (behaves nondeterministically either as pure or inertial delay), and
     - distributed inertial (given a number k, this can be modelled as a series interconnection of k simple, one-stage, inertial delays, which therefore can store a "train" of pulses [18]);
   - bounded delay (minimum and maximum delay constraints are specified).
2. The delay model of the circuit (delay model 'in the large'), such as the feedback delay model [12, 43], the gate delay model [29] or the gate+wire delay model [42, 5].
3. The environment-circuit interaction model, such as:
   - fundamental mode ([12, 43] the inputs can change their values only after the internal transitions have stabilised), and
   - input-output mode ([29] "reactive" behaviour, in which the environment may change the input state even if the circuit has not reached a stable state);
4. Circuit switching semantics ('race' model [5]) such as:
   - multiple winner (all the excited elements switch simultaneously),
   - general multiple winner (any subset of the set of excited elements may switch first), and
   - extended multiple winner(similar to the previous one, but every changing signal passes through a third, undefined, state before reaching its final value).

These approaches all differ in their generality, representation and complexity of the associated analysis and synthesis procedures ([48]). For example, the most strict model, in terms of independence from delay variations, would be based upon:

1. the assumption that gates and wires had unbounded delays,
2. the environment acted in input-output mode, and
3. the race model was the extended multiple winner one.

Circuits that operate correctly (i.e., according to their specification) when using this model are informally called *delay-insensitive* circuits. In many practical cases, however, such modelling requirements would be too strict, and unlikely to produce an efficient circuit. Many authors have in fact shown that no useful delay-insensitive circuit can be built using "basic" gates (such as those found in common standard-cell libraries). Hence the unbounded gate and wire delay assumption can be applied only to more “complex” gates (such as, for example, arbiters or complete handshaking elements), that need to be carefully designed by hand.

In this paper we assume the circuit to be modelled with a combination of inertial unbounded gate delay, general multiple winner, and input-output mode. These choices seem to best combine conservative but tight modeling choices (inertial unbounded delay with general multiple winner) and generality (input-output mode). Circuits that operate correctly when using this model are informally called *speed-independent* circuits.

High-level modelling At the higher levels, there have been a number of different approaches, typically falling into one of the following categories: finite-state machines, process algebras or explicit causality models.

The first group [31, 7] essentially builds on the traditional Huffman model of asynchronous circuits, which is similar to the standard synchronous approach. This model decomposes the circuit into a block of combinational logic and a set of feedback wires implementing its state. Its main drawback is the fundamental mode assumption, and hence a basic difficulty in dealing with arbitrary reactive interaction between the circuit and its environment.

The second group [23, 14] makes use of various compositional and transformational techniques, based upon the description of a circuit as a collection of communicating processes. This approach makes assumptions about the set of basic components, whose external interaction protocol does not depend on wire delays, but whose internal delays must be carefully matched and controlled. The actual realisation of these assumptions in the final layout may cause major difficulties. Another possible shortcoming is its inability to explicitly represent causality at the event level, i.e. the ordering between the positive and negative edges of signals.

The third approach [35, 6, 44, 17], based upon Petri nets, avoids the main problems of the first two groups. This uses the model of Signal Transition Graphs (STGs), which is the interpretation of Petri nets by means of the signal transitions. A set of analysis methods based on state-transition and net unfolding semantics have been developed. Efficient synthesis methodologies, including both state assignment and logical equation synthesis, have been reported. These generate hazard-free circuits from STGs under certain restrictions imposed on the structural and behavioural subclasses of STGs, logical elements and delay models [20].

The STG approach has the disadvantage of being rather low level. The initial specification of a circuit at the level of signal transition ordering can be a good formalism to build upon, if one starts from detailed descriptions such as timing diagrams. But if the the desired control behaviour is abstract enough, e.g. when a set of binary signals has not yet been defined, a more symbolic notation is preferable. For example, if the problem is to design a data buffer of a given capacity k with data read and data write operations, in such a way that when data is written into the buffer it cannot be overwritten until it has been read, the original requirement would be only that ”the number of writing actions for this buffer should at no time exceed the number of reading actions by k, or be less than the number of reading actions”.

The STG approach is not a particularly good formalism to compose and decompose specifications. One needs to build a structural model in order to define which interconnections correspond to which STG signals. It would be more convenient to perform behavioural composition at a more abstract level, to avoid the complexity and diversity of the potential STG equivalents of the same abstract behaviour.

Our technique addresses the high-level behavioural composition, by using a standard definition of behaviour of interconnected Petri nets. The basic object in our hierarchy will be called a *discriminator*. A discriminator is viewed from the outside (that is, from the user's perspective, or from the environment's perspective) as specified using a *characteristic predicate* that describes the sequences of valid input-output behaviors. Its "internals" are modeled initially as a labeled Petri net, where each transition is labeled by some communication action between the circuit and the environment. Straightforward implementation of this model would require a translation of the abstract actions into up and down transitions of signals. The task may be too complex to be solved at this level, though. So we use a hierarchical decomposition approach, that iteratively refines the labelled Petri net into an interconnection of simpler ones, until these can be expanded at the signal transition level and synthesized separately. The circuit obtained by structural composition of the sub-modules will then satisfy the specification by construction.

Hence our proposed design procedure comprises two major stages:

- abstract, or symbolic, synthesis of the control circuit, and
- logic synthesis at the gate level, from a binary encoded behavioural specification.

The first stage consists of the following steps:

1. Abstract decomposition of functionally independent units, using formalisms such as symbolic events, traces of events, characteristic predicates on traces, and labelled Petri nets. Each unit is characterized as a certain type of *discriminator*.
2. If direct translation into a circuit is too difficult, the unit is further subdivided until either a standard self-timed circuit element for each sub-unit can be found, or until it is possible to create an internal dynamic description for each new sub-unit discriminator.

The result of this stage is an interconnection of discriminators, i.e. abstract components with a number of symbolic ports. The structural part of this description is a netlist. The behaviour is defined by the parallel composition of individual behavioural descriptions of the discriminators in terms of labelled Petri nets.

During the second stage, the designer performs the following steps:

1. Conversion of the internal abstract behavioural description of each discriminator into its binary equivalent by means of *signalling expansion*, which is defined in terms of a Signal Transition Graph.
2. Verification of the signalling expansion with respect to its correctness and completeness and correcting it if required, thus providing the final behavioural model for subsequent logic synthesis.
3. Derivation of boolean functions characterizing the result of the design process.

During the latter phases, the designer may wish to use software tools for circuit synthesis from signal transition graphs and related models [38, 16]. Although existing tools are usually adequate, they are still limited in power. Certain classes of useful behaviour defined by signal transition graphs (e.g., most forms of fair mutual exclusion) require a substantial manual synthesis effort, which demonstrates the need for more extensive research. Another problem of most existing automated algorithms is that they are based on the state graph, which in the worst case has exponential size with respect to a Signal Transition Graph. This means that the complexity of such algorithms can be too high for practical large examples.

3 Abstract synthesis

We now present a formal approach to the definition of behaviour of a circuit in abstract terms. At this level we will neglect issues such as encoding of operations using signal levels or transitions on wires. Every mechanism is considered capable of performing a set of symbolic actions (e.g., read, write, strobe, acknowledge, ...). An interconnection of such mechanisms can communicate by performing *shared actions*, on a hypothetic underlying medium.

Our target is a design methodology for digital circuits, so at some point we will be forced to encode symbolic actions using signal values and transitions. Such a postponement of the low-level synthesis issues helps mitigate the complexity problems associated with semantic representations. It is crucial to perform a large part of the compositional and formal analysis work at the symbolic level, so that logic synthesis can be done automatically on smaller components of the overall system.

3.1 Theoretical background

We consider an abstract model of a mechanism with a finite set of nodes which are labelled with distinct symbols from an alphabet $A = \{a_1, a_2, \cdots, a_n\}$. To each labelled node we relate an event whose occurrence manifests itself by adding an appropriate symbol to the sequence of symbols of previous events. The semantic formalism that we use for the abstract symbolic specification of mechanisms is trace theory [34]. Although this model captures concurrency in its interleaving form, which in itself is inadequate to express the idea of true concurrency [24], it appears to be powerful enough to allow proving the correctness of high- and low-level design methodologies.

3.1.1 Processes

A finite-length string of symbols is called a *trace*[3]. The set of all traces with symbols of alphabet A is denoted by A^*.

A *process* is a pair $\Sigma = \langle A, X\rangle$, where A is an alphabet and X is a non-empty *prefix-closed* set of traces, $X \subseteq A^*$, i.e. $X \neq \emptyset$, $X = \mathrm{pref}(X)$ where $\mathrm{pref}(X)$ denotes a set X extended with all prefixes of traces in X including the empty trace ϵ.

The key operations on processes are *projection* and *weaving* (which is often called *synchronisation*).

The projection of trace σ on alphabet A, denoted by $t\lceil A$, is obtained by removing from σ all symbols that are not in A. The projection of process $\Sigma = \langle B, X\rangle$ on A, denoted by $\Sigma\lceil A$, is defined by $\langle B \cap A, \{\sigma\lceil A \mid \sigma \in X\}\rangle$.

The weave of processes $\Sigma 1 = \langle A1, X1\rangle$ and $\Sigma 2 = \langle A2, X2\rangle$, denoted by $\Sigma 1 \text{ w } \Sigma 2$, is a process defined by

$$\langle A1 \cup A2, \{\sigma \in (A1 \cup A2)^* \mid \sigma\lceil A1 \in X1 \ \wedge \ \sigma\lceil A2 \in X2\}\rangle$$

The projection operator is interpreted as an abstraction of a process with respect to a subset of its components, and the weaving operator represents the composition of a pair of processes, which yields a new process. The reason to use the synonym "synchronisation" for "weaving"

[3]The limitation to finite-length strings simplifies the theory at the price of reducing the type of specifiable behaviours. Finite traces allow the specification of so-called *safety* properties only (e.g., no two processes will ever gain simultaneous access to a critical region) but not of so-called *liveness* properties (e.g., a requesting process will be granted access to the resource eventually).

is that in their weaving the pair of processes either act completely independently, when they operate on their own disjoint events, or in strict compliance, if the event is shared.

It is often convenient to hide the shared events among two processes after their composition. The communication between them thus becomes a local point-to-point mechanism that is internal to the composite process and facilitates hierarchical modelling. This can be achieved by a superposition of two operators: the weave of the processes involved, and the projection of the weave onto the disjoint union $((A \cup B) \setminus (A \cap B))$ of the events of the processes.

3.1.2 Specification

There are many ways in which the intended behaviour of a process can be specified. One could, for example, use the enumeration of all allowed traces on its events. This approach is perhaps the most natural as a starting point for a synthesis procedure when one does not know the internal dependencies between events. It simply represents the external behaviour of the mechanism. However, such an enumeration is simply impossible for processes that can perform an unbounded number of events (the most interesting ones in practice).

A compact way to formalize knowledge about a process is to define a *characteristic predicate* specifying what traces belong to the trace set of the process, thereby avoiding the enumeration of the traces themselves.

A pair $\langle A, \Pi \rangle$ is called the *one-to-one specification* of a process $\Sigma = \langle A, X \rangle$ iff

$$\Sigma = \langle A, \{\sigma \mid \sigma \in A^* \wedge (\forall \sigma 1 : \sigma 1 \leq \sigma : \Pi(\sigma 1))\} \rangle$$

This definition of specification establishes a strong conformity between the process and its description in terms of a characteristic predicate. Every trace satisfying the predicate must be implementable in the process, and every trace implemented by the process must satisfy the predicate.

For practical purposes explained below, we need a more general notion of specification, in which the predicate constrains the process only on a subset of its events. Basically, to allow the decomposition of a process into simpler processes we will introduce additional, internal events. The composite process is considered to satisfy its specification when its *external* events satisfy it.

A pair $\langle A, \Pi \rangle$ is called the *(top-down) specification* of a process $\Sigma = \langle A', X \rangle$ such that $A \subseteq A'$ iff

$$\Sigma \lceil A = \langle A, \{\sigma \mid \sigma \in A^* \wedge (\forall \sigma 1 : \sigma 1 \leq \sigma : \Pi(\sigma 1))\} \rangle$$

A process satisfying a top-down specification is called an implementation of the top-down specification. A one-to-one specification is a special case of a top-down specification. The latter is important for the abstract synthesis of a Petri net implementation from a given top-down specification. Unless we emphasize the fact that the specification is one-to-one, we assume that it is a top-down specification.

In order to manipulate processes and their compositions in terms of their specifications, we use the so called '*Conjunction-Weave Rule*' (CWR) [34], which states that if $\langle A1, \Pi 1 \rangle$ and $\langle A2, \Pi 2 \rangle$ are specifications of processes $\Sigma 1$ and $\Sigma 2$, respectively, then the process $\Sigma 1 \text{ w } \Sigma 2$ (the weaving of $\Sigma 1$ and $\Sigma 2$) is specified by $\langle A1 \cup A2,\ \forall \sigma : \Pi 1(\sigma \lceil A1) \wedge \Pi 2(\sigma \lceil A2) \rangle$ or, briefly, $\langle A1 \cup A2,\ \Pi 1 \wedge \Pi 2 \rangle$.

3.1.3 Labelled Petri nets and their processes

We now introduce the concept of abstract implementation of a process. Such an implementation defines the internal causal relationship between the events on the process boundary. This is equivalent to replacing an abstract specification that states "every p is followed by a q" with an implementation where every occurrence of p *causes*, through some physical connection, an occurrence of q. The set of traces of a given process can hence be viewed as *generated* by an underlying formal dynamic model (for example, by a labelled Petri net [33]) rather than *defined* by a predicate.

A *labelled Petri net* (LPN) is a triple $\langle N, A, \gamma \rangle$:

- N is a Petri net, defined as $\langle P, T, H, M^0 \rangle$, where P is a finite set of places, T is a finite set of transitions (P and T are disjoint), $H \subseteq T \times P \cup P \times T$ is the flow relation, and M^0 is the initial marking that is a function $M^0 : P \rightarrow \omega$, $\omega = \{0, 1, 2, ...\}$, which associates each place with a non-negative integer;
- A is an alphabet of events; and
- $\gamma : T \rightarrow A$ is a partial function that labels transitions from the set T by symbols in A.

Unless we are specifically interested in the underlying Petri net, we denote the whole LPN as N, so $N = \langle P, T, H, M^0, A, \gamma \rangle$. Graphically, an LPN is represented as a bipartite directed graph with two types of vertices, circles for places and bars for transitions (Figure 1). The places are connected to transitions and vice versa by arcs, which indicate the flow relation. The initial marking places a number of tokens into a place according to the value of function M^0. We assume the standard one-to-one relationship between a mapping into the set of non-integers and a multiset, which allows us to use an alternative representation of the initial marking as a multiset of places, in which the number of copies of each marked place $p \in P$ is equal to $M^0(p)$. The labelling function assigns a label from alphabet A to a transition, which also bears its unique name as a member of set T. Thus, since γ is a partial function, some of the transitions are not labelled by any symbols. When it comes to analysing the sequences of actions an LPN may generate, such unlabelled transitions do not contribute any symbols. The mechanism for generating sequences of transitions and their labels is due to the dynamic behaviour of ordinary Petri nets [30], which is determined by the *firing rule*. The firing rule states that a transition $t \in T$ is *enabled* at marking M^0 if all its predecessor places are marked, i.e. for each such place $M^0(p) > 0$. An enabled transition t *may fire*, producing a new marking M with one less token in each predecessor place and one more token in each successor place. Formally, $M(p) = M^0(p) - 1$ if $(p, t) \in H, (t, p) \notin H$, $M(p) = M^0(p) + 1$ if $(t, p) \in H, (p, t) \notin H$, and $M(p) = M^0(p)$, otherwise. The firing of transition t in marking M^0 leading to marking M is denoted by $M^0[t > M$. If we denote the set of possible markings by $\mathcal{M}$, we can define the *firing relation* of the net as $r(N) \subseteq \mathcal{M} \times T \times \mathcal{M}$. Thus, $M^0[t > M \in r(N)$. It is clear that the new marking M can lead to similar firings of transitions, thus generating a *firing sequence* from M^0: $M^0[t_1 > M_1[t_2 > M_2...[t_n > M_n = M$. Therefore, the firing relation can be extended to the *reachability relation*: $R(N) \subseteq \mathcal{M} \times T^* \times \mathcal{M}$, where T^* is the set of all sequences of transitions in T, including the empty sequence. The set of markings M reachable through $R(N)$ is called the reachability set of the Petri net. Let this be denoted by $\mathcal{R}(N)$. A set of firing sequences generated by net N through $R(N)$ is called the *firing language* of the Petri net, and is defined as $L(N) = \{\tau | (M^0, \tau, M) \in R(N)\}$. This set fully describes the behaviour of the Petri net in terms of the so called *interleaving semantics* of concurrency. The reason behind this

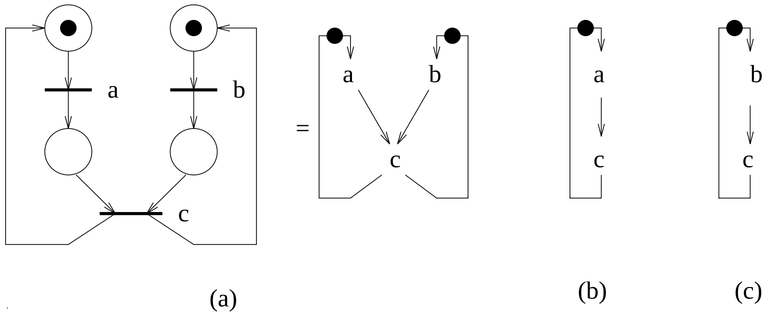

Figure 1: Example of a labelled Petri net

term is simple: for any marking which enables several transitions, we consider all possible firing sequences. This gives rise to a number of different interleavings of transitions, which produce a number of possible firing sequences.

If we look at the whole LPN as a labelled net, we can think about its behaviour in terms of labelled transition sequences. This is precisely the way in which the LPN can generate a set of traces over the alphabet A. We call this set the *set of traces* of the LPN. It is denoted as $L_A(N)$ and formally defined through a mapping $\Gamma : L(N) \rightarrow A^*$. That is, for any non-empty $\tau = \tau' t \in L(N)$ we define $\Gamma(\tau) = \Gamma(\tau')\gamma(t)$ if $\gamma(t)$ is defined, or $\Gamma(\tau) = \Gamma(\tau')$, otherwise. Also, for empty strings, $\Gamma(\epsilon) = \epsilon$. Thus, $L_A(N) = \{\sigma | \sigma \in A^* \wedge \exists \tau \in L(N) \wedge \sigma = \Gamma(\tau)\}$.

Using the above notation we define the process generated by an LPN $N = \langle P, T, H, M^0, A, \gamma\rangle$ as a pair $\langle A, L_A(N)\rangle$. Since $L(N)$ is prefix-closed by construction (a firing sequence belongs to $L(N)$ only if its prefixes do), $L_A(N)$ is also prefix-closed, because the trace labelling mapping Γ is applied to *all* transition sequences without changing the order of the transitions.

The LPN shown in Figure 1(a) generates the process

$$\langle\{a,\ b,\ c\}, \{\epsilon,\ a,\ b,\ ab,\ ba,\ abc,\ bac, \cdots\}\rangle$$

which, as can be seen intuitively, can be specified by $\langle\{a,\ b,\ c\},\ \Pi1 \wedge \Pi2\rangle$, such that $\Pi1 = (0 \leq l(\sigma\lceil a) - l(\sigma\lceil c) \leq 1)$ and $\Pi2 = (0 \leq l(\sigma\lceil b) - l(\sigma\lceil c) \leq 1)$, where the notation $l(\sigma\lceil a)$ is used for the length of a trace $\sigma\lceil\{a\}$, i.e. the number of occurrences of a in the trace σ.

It is clear that the above process can be obtained using CWR by weaving two processes with specifications of the form $\langle\{a, c\}, \Pi1\rangle$ and $\langle\{b, c\}, \Pi2\rangle$

In terms of LPNs, the weaving operation amounts to identifying those symbols, and hence their transitions, which are common between the two processes. So we could obtain a composite implementation, shown in Figure 1(a), by weaving the two LPNs given in Figure 1(b) and (c), assuming that the transition c is identical in both LPNs. As this example shows, it would obviously be more convenient to perform weaving of processes implemented by LPNs directly on their LPNs. In the general case, the composition operator should be able to deal with the LPNs in which the labelling functions are not necessarily one-to-one mappings of transitions to event labels. This is formally justified below.

3.1.4 Composition of labelled Petri Nets

The parallel composition of LPNs is defined analogous to [33].

For an LPN $\langle P, T, H, M^0, A, \gamma\rangle$, for any $a \in A$, $T(a) = \{t \mid t \in T : \gamma(t) = a\}$ and $H(a) = \{h \mid h \in H : (h = (t,p) \vee h = (p,t)) \wedge \gamma(t) = a\}$. They are naturally generalised to sets:

$T(A) = \bigcup_{a \in A} T(a)$ and $H(A) = \bigcup_{a \in A} H(a)$. Also, $T^\bullet = \{t \mid t \in T : \gamma(t) = undefined\}$ and $H^\bullet = \{(p,t),(t,p) | \ (p,t),(t,p) \in H : \gamma(t) = undefined\}$.

Let two abstract behaviours be implemented by two LPNs: $N1 = \langle P1, T1, H1, M1^0, A1, \gamma 1\rangle$ and $N2 = \langle P2, T2, H2, M2^0, A2, \gamma 1\rangle$.

The *parallel composition* of these nets, denoted as $N1 \parallel N2$, is an LPN $\langle P, T, H, M^0, A, \gamma\rangle$ obtained in the following way:

1. $P = P1 \cup P2$.
2. $T = T1(A1 \setminus A2) \cup T2(A2 \setminus A1) \cup T12 \cup T1^\bullet \cup T2^\bullet$, where $T12 = T1(A1 \cap A2) \times T2(A1 \cap A2)$.
3. $H = H1(A1 \setminus A2) \cup H2(A2 \setminus A1) \cup H12 \cup H1^\bullet \cup H2^\bullet$, where $H12 = \{(t,p),(p,t) | \ t = (t1,t2) \in T12, p \in P : (p,t1) \in H1 \vee (t1,p) \in H1 \vee (p,t2) \in H2 \vee (t2,p) \in H2\}$.
4. $M^0 = M1^0 \cup M2^0$.
5. $A = A1 \cup A2$.
6. for each t:

$$\gamma(t) = \begin{cases} \gamma 1(t) : & t \in T1(A1 \setminus A2) \\ \gamma 2(t) : & t \in T2(A2 \setminus A1) \\ \gamma 1(t1) : & t = (t1,t2) \in T12 \\ undefined : & otherwise \end{cases}$$

Since we completely identify the labels with the same name in the two LPNs, it is natural to assume that the $\parallel$ operator is symmetric, i.e. $N1 \parallel N2 = N2 \parallel N1$, whereby in the above construction we should have $T12 = T21$ and $H12 = H21$.

It is easy to prove that the $\parallel$ operator is associative, i.e. $(N1 \parallel N2) \parallel N3 = N1 \parallel (N2 \parallel N3)$.

Using the results of [33] we can state the following Proposition about the process generated by the parallel composition of the two LPNs.

Proposition 3.1 *Let two LPNs $N1 = \langle P1, T1, H1, M1^0, A1, \gamma 1\rangle$ and $N2 = \langle P2, T2, H2, M2^0, A2, \gamma 1\rangle$ generate two processes $\Sigma 1 = \langle A1, X1\rangle$ and $\Sigma 2 = \langle A2, X2\rangle$, where $X1 = L_{A1}(N1)$ and $X2 = L_{A2}(N2)$. Then their parallel composition $N = N1 \parallel N2 = \langle P, T, H, M, A^0, \gamma\rangle$, built according to the above rules, generates a process $\Sigma = \langle A, X\rangle$ such that $X = L_A(N)$ and $\Sigma = \Sigma 1 \ w \ \Sigma 2$.*

We can use this proposition, and the fact that weaving is idempotent [39], to semantically justify the "forceful" imposition of idempotence onto the $\parallel$ operator. Thus, despite the formal definition of $\parallel$, we shall assume that $N \parallel N = N$. This pragmatic measure helps us to avoid unnecessary splitting of transitions if two identical nets are composed using the above construction.

Using the above proposition and the associativity of both $\parallel$ and *weave*, we can infer that for any $N_1, ..., N_n : L_A(\|_{i=1..n} N_i) = w_{i=1..n} L_{A_i}(N_i)$, where every A_i is the alphabet of N_i and $A = \bigcup_{i=1..n} A_i$.

This proposition allows redefinition of the CWR in terms of LPNs. If two specifications $\langle A1, \Pi 1\rangle$ and $\langle A2, \Pi 2\rangle$ are respectively implemented by LPNs $N1 = \langle P1, T1, H1, M1^0, B1, \gamma 1\rangle$ and $N2 = \langle P2, T2, H2, M2^0, B2, \gamma 1\rangle$ ($A1 \subseteq B1$ and $A2 \subseteq B2$), then the parallel composition LPN $N = N1 \parallel N2$ is the implementation of $\langle A1 \cup A2, \Pi 1 \wedge \Pi 2\rangle$. This allows efficient manipulations of *finite* objects, such as process specifications and their abstract implementations in

LPN, whilst avoiding the use of their behaviours in terms of trace sets, which are infinite for cyclic processes.

Using LPNs to construct process descriptions offers another important advantage: use of the algebra of LPNs, based on the results of Mazurkiewicz [24] for unlabelled Petri nets. Mazurkiewicz algebra consists of Petri nets produced by parallel composition of Petri nets, with as zero element the empty Petri net $N0 = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ and as generator set the set of all one place nets. A Petri net $N(p, T1, T2, k) = \langle \{p\}, T1 \cup T2, (T1 \times \{p\}) \cup (T2 \times \{p\}), \{(p, k)\} \rangle$ is called a *one place net*. It contains only one place p and two sets of transitions. Its initial marking places $k, k \in \omega$ tokens into p.

We extend this approach to LPNs by defining the empty LPN $N0 = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ and one place LPNs $N(p, T1, T2, k, A, \gamma)$ in the obvious way. The algebra of LPNs can thus be constructed using our parallel composition $\|$ operator.

3.1.5 Constructing LPNs from primitive components

The LPN algebra with its $\|$ operator allows construction of more complex LPNs from elementary components, one-place LPNs, where each such one-place net can implement a specific requirement about the form of causality or choice relationship between individual events. Such a requirement in its general form can be defined using our specification notation, the characteristic predicate defining a set of allowable traces over a given alphabet. This is done as follows.

For a two sets of events, $A = \{a_1, ..., a_m\}$ and $B = \{b_1, ..., b_n\}$, called *causes* and *effects* respectively, and a value k, called *causality slack*, we define a general *causality requirement*, which is a specification pair $\langle A \cup B, \pi^{\rightarrow}(A, B, k) \rangle$, where $\pi^{\rightarrow}(A, B, k) = (\forall \sigma \in (A \cup B)^* : \sum_{i=1}^{n} l(\sigma \lceil B) - \sum_{i=1}^{m} l(\sigma \lceil A) \leq k)$. The $\pi^{\rightarrow}(A, B, k)$ predicate is used to indicate that the number of times that any effect from B can occur without at least one occurrence of a cause from A is bounded by k.

Behavioural specifications often need such primitive paradigms as causality and choice constraint between a limited group of events. Some formalisms, such as guarded commands or CSP [23, 26], use them as special constructs, but they appear to be special cases of the above causality requirement. For example, $\langle \{a, b\}, \pi^{\rightarrow}(\{a\}, \{b\}, 0) \rangle$ defines a simple form of the strong causality, in which event b can only happen if a has occurred. The $\langle \{a, b, c\}, \pi^{\rightarrow}(\{a\}, \{b, c\}, 0) \rangle$ pair defines a simple form of two-way choice, in which neither b or c can happen before a, and if a occurs once, it may cause either b or c to occur but not both. A weak causality link, in which some effect c can be caused by either a or b, is defined by $\langle \{a, b, c\}, \pi^{\rightarrow}(\{a, b\}, \{c\}, 0) \rangle$. A strongly causal link between a and b but with some finite "slack" k, allowing b to occur k times without at least one occurrence of b is defined by $\langle \{a, b\}, \pi^{\rightarrow}(\{a\}, \{b\}, k) \rangle$.

By using one-place LPNs we can easily implement the general causality requirement. For example, the first of the above cases is implemented by the following LPN:

$$N(p, \{t_1\}, \{t_2\}, 0, \{a, b\}, \{(t_1, a), (t_2, b)\})$$

The LPNs for the other examples are obvious.

If we regard a process specification as a conjunction of causality requirements, where each requirement is an appropriate parametrisation of the generic form $\langle A \cup B, \pi^{\rightarrow}(A, B, k) \rangle$, we can show that the CWR, redefined for LPNs, justifies applying a one-to-one conversion of the list of causality requirements into the corresponding LPN implementation of the process.

We illustrate the LPN construction process with a simple example. A one-place buffer is a mechanism which can be defined by the specification $\langle \{a, b\}, 0 \leq l(t \lceil a) - l(t \lceil b) \;\leq\; 1 \rangle$, which

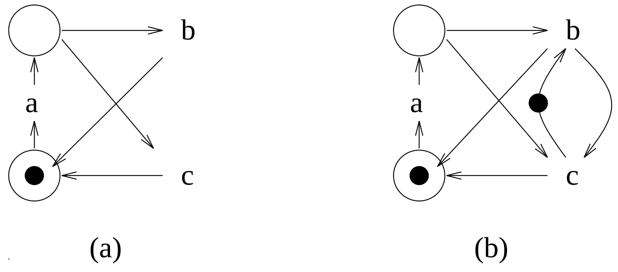

Figure 2: Labelled Petri nets for Selector

is composed from two primitive causality requirements joined by conjunction: $\langle\{a,b\}, l(t\lceil b) - l(t\lceil a) \leq 0\rangle$ and $\langle\{a,b\}, l(t\lceil a) - l(t\lceil b) \leq 1\rangle$.

Here, a and b stand for input and output port names respectively, and the corresponding behavioural actions on these ports are writing data into the buffer and reading data from the buffer. The first requirement states that in a one-place buffer, every output (reading) event occurrence must be preceded by at least one input (writing) event and repetitive reading of the same data is not possible. The second requirement ensures that no overwriting of the as yet unread data is allowed, which implies that the buffer capacity is one data item.

By means of the CWR for LPNs, we can state that the LPN shown in Figure 1(b) is the correct LPN implementation of the above specification. In the same way we can demonstrate that, if we put k tokens instead of one into the same place as in the one-place buffer LPN, we obtain the LPN for a k-place buffer.

Another useful example is an ordered two-way selector. This has one input port and two output ports. Data is always written to the input port (symbol a) and read via one of the output ports (symbols b and c). In its role as a data store, the selector acts just as a one-place buffer, except for "splitting" event b into two mutually exclusive events without changing the slack between input and output. This requirement can be formally represented by the following specification: $\langle\{a\} \cup \{b,c\}, 0 \leq l(t\lceil a) - l(t\lceil b,c) \leq 1\rangle$. The corresponding LPN, obtained as a parallel composition of two one-place LPNs, is shown in Figure 2(a). If we were to build a two-way selector that non-deterministically chooses either b or c, the above specification and its LPN are sufficient. We should however recall that our selector is "ordered", which means that we would like to constrain, with extra causality rules, the freedom to choose between the output ports. This ordering constraint can be represented by the two causality requirements on the order of events b and c, which together model the behaviour of one place buffer: $\langle\{b,c\}, 0 \leq l(t\lceil b) - l(t\lceil c) \leq 1\rangle$. The events on b and c must thus alternate, starting with b. This requirement, joined via the CWR with the previous one, completely defines the behaviour of the ordered two-way selector: $\langle\{a,b,c\}, 0 \leq l(t\lceil a) - l(t\lceil b,c) \leq 1 \wedge 0 \leq l(t\lceil b) - l(t\lceil c) \leq 1\rangle$. The corresponding LPN implementation is shown in Figure 2(b).

3.1.6 Verification of process decomposition

The above discussion of the LPN construction process using primitive one-place fragments, each of which has a precise equivalent in terms of the causality requirement, demonstrates how an LPN satisfying a set of elementary inequalities can be built. The overall predicate formed by the conjunction of these inequalities (we call such a predicate a *normal form* specification of the LPN) remains true for any trace generated by the LPN, as implied by the CWR. On the other hand, CWR implies that any trace $\sigma \in A^*$ which satisfies the normal form of N, must be a member of $L_A(N)$.

The relationship between an LPN and its normal form specification is called a *one-to-one conformity* between the LPN and the predicate-based specification. Our general definition of a specification however does not need to be a normal form of our initial LPN. For example, some transitions (*internal* transitions) and associated places may not be represented by explicit causality conditions but only be required by some implementation choice (e.g., the counter output transitions if we choose to implement the k place buffer with a counter, as we discuss more in detail below). In this situation, which can often occur in practical design, the relationship between the predicate-based specification and its LPN has to be ruled by the top-down specification definition. We refer to this type of conformity as a (top-down) conformity.

The concept of conformity would not be very useful if we were not able to decompose the LPN model into a set of simpler LPNs whose parallel composition would satisfy the original LPN behaviour[4]. The decomposition process should not prevent the designer from trying to refine the events of the original LPN model. The designer would have to be able to check whether a particular decomposition was correct with respect to the initial specification. Checking the final LPN against the original predicate specification using the top-down conformity can be very laborious, since it requires not only checking that every generated trace satisfies the predicate, but also that every trace satisfied by the predicate is implementable in the LPN.

In this sense we can look at another instance of the "implementation satisfies specification" paradigm, formulated within the LPN framework. Here, the parallel composition of more elementary LPNs is regarded as the implementation, while the original LPN is assumed to be the specification.

The implementation correctness check can obviously be done in the following straightforward manner. Provided that the original LPN, say N, is a correct implementation of the predicate-based specification, we have to prove that the projection of the process generated by the implementation LPN (denoted as N') onto the set of labels of N' is equal to the process generated by N. In other words, we must verify the condition $L_{A'}(N')\lceil A = L_A(N)$. Existing techniques, using either the Petri net reachability graph or net unfolding, can be used to solve this problem. The disadvantage of this method is however that it requires construction of the reachability graph of N' (which could be very complex), and doing transformations on it.

It would definitely be preferable to use hierarchical decomposition/verification techniques, in which a net is first presented as a composition of nets of a relatively moderate size. Each subnet of the composition is then decomposed into a set of nets that is not difficult to verify. Each decomposition is then checked independently until the whole implementation LPN N' is proved correct with respect to the original net N. Since the original net is a correct implementation of the predicate-based specification, we can conclude that the final net implementation is correct, too.

Thus the overall abstract synthesis strategy is as follows. The designer first develops a set of fundamental causality constraints and builds, using the concept of one-to-one conformity, the original LPN. The latter is then decomposed by means of a set of simpler LPNs, which are brought into parallel composition to form another LPN. Finally, this LPN is checked against the original LPN.

The designer often requires that the implementation satisfies its specification in a more liberal way than the one we have just discussed. For example, the designer may impose bounds

[4]It will be shown later that the model complexity can be measured in terms of the number of states the event-based model generates. The reason for using such a metric, despite the fact that the decomposed LPN can be descriptively very simple, is that the logic synthesis stage essentially draws upon the state graph representation.

on the behaviour of a data buffer. A lower bound ensures that the buffer implementation will not overflow at peak traffic. An upper bound specifies some reasonable constraints on memory or silicon area.

In terms of trace set containment, this can be stated as follows. Let a pair of predicate-based specifications $S1 = \langle A1, \Pi1\rangle$ and $S2 = \langle A2, \Pi2\rangle$ (assume for convenience that $A1 = A2 = A$) define two processes one-to-one implemented by two LPNs $N1$ and $N2$. We call the $S1$ and its LPN $N1$ a *lower specification bound* and $S2$ and its LPN $N2$ a *upper specification bound* iff $L_A(N1) \subseteq L_A(N2)$. Now any LPN N defined on an alphabet $B, A \subseteq B$, is called a *weak implementation* or *weakly conformant* to its specification iff $L_A(N1) \subseteq L_B(N)\lceil A \subseteq L_A(N2)$. Thus, the verification of a possible implementation requires solving the language containment problem. Such a problem can be solved in an alternative manner, suggested in [33], which makes use of the parallel composition operator. If we have two LPNs N and N' with alphabets A and A' such that $A \subseteq A'$, then $L_A(N) \subseteq L_{A'}(N')\lceil A$ if and only if $L_{A'\cup A}(N' \parallel N)\lceil A = L_A(N)$. The proof of this equivalence follows from the the argument in [39] about the properties the weaving of trace structures. This approach is similar to those used for finite-state machine verification [25]. It should be noted that we do not yet specify inputs and outputs between the circuit and its environment. Their communication is thus undirected and amounts to pure (rendez-vous type) synchronisation, which is conveniently used to define the control flow at the initial stage. The specifications of the environment and the circuit can thus be regarded as equal. With our "two bounds" approach we can always assume that the lower bound is what we can *expect, at most* from the environment, and thus *at least* from the circuit, whereas the upper bound is what we can *afford, at most* in the circuit.

3.2 Discriminators

So far we have only looked at mechanisms with alphabets which are the names of nodes connecting them to the environment. We assumed only the existence of their behavioural interpretation, in which each node was in one-to-one correspondence with the name of an event of the process generated by the mechanism. LPNs complied easily with this behavioural view. Even the composition of two processes did not add much to the structural aspect, as to whether the weaving of two processes, or the parallel composition of LPNs, implied an interconnection of two circuits, or simply the combination of two different behavioural views of the same circuit. Although we used such words as "input port" and "output port", we did not make any assumptions about possible identification of ports of two different mechanisms which would be associated with the same channel, and hence the same event symbol, if the designer intended to interconnect these mechanisms together.

In order to proceed with the synthesis of control circuits from the behavioural specification in terms of LPNs, the designer must at some point address the issue of structure. The decomposition of the model into subcomponent models requires construction of a set of structurally separate mechanisms interconnected through their ports. The overall behaviour of the interconnection, when expressed using the LPN composition, must comply with the initial specification of the circuit as a single component, if such specification is possible.

The refinement of a process into subprocesses whose composition through the $\parallel$ operator forms the given process and have no separate structural components underlying them, will be called *behavioural refinement*. If we refine a process into subprocesses associated with separate structural components, and identify the structural interconnections, we refer to it as *structural refinement*.

In order to perform structural synthesis at this abstraction level, we want to interconnect some of the nodes of two composed mechanisms, thereby identifying corresponding symbols and events. For the sake of convenience, we introduce the concept of a structural object called a *discriminator* (in contrast to that of the behavioural type, which is a process). We associate with a discriminator D a process $\langle A_D, X_D \rangle$ and assume that this process, possibly a parallel composition of other processes, is defined on a single structural component. Thus we regard D as a black box with a set of pins labelled by symbols from A_D. Using such pins we can structurally interconnect D with other discriminators, and thus form discriminators of a higher abstraction level.

Consider a subclass of predicates on traces such that the predicates are defined on the values of parameters $l(\sigma\lceil B)$, where B is a subset of A_D, i.e. they specify a relationship between the numbers of occurrences of the node symbols. We therefore consider a class of mechanisms that "discriminate" between their nodes during their operation, according to some prescribed strategy. D is called a *discriminator* of Π-type if for each $\sigma \in X_D$ the predicate Π is true and it is true only for traces in X_D.

In order to define a discriminator D of Π-type, we specify a pair $\langle A_D, \Pi \rangle$, called the discriminator specification, such that $X_D = \{\sigma \mid \sigma \in A_D^* \wedge \Pi(\sigma)\}$, and we recall that X_D is prefix-closed.

The one-place and k-place buffers and the ordered one-place selector, can be regarded as examples of discriminators over their corresponding sets of input and output ports. The LPN shown in Figure 1(b) defines the behaviour for a one-place buffer $BUF_1(a,b)$. We denote such a discriminator by $BUF_1(a,b)$, implying that it is a two-node mechanism with specification

$$\langle \{a,b\}, 0 \le l(\sigma\lceil a) - l(\sigma\lceil b) \ \le \ 1 \rangle$$

If we construct predicates on $l(\sigma\lceil B)$ for subsets B in A_D, we can obtain various types of discriminator. The following table contains some of these types with their characteristic predicate.

name	**symbol**	**characteristic predicate**
buffer of capacity k	$BUF_k(a,b)$	$\Pi 1: \ 0 \le l(\sigma\lceil a) - l(\sigma\lceil b) \le k$
multi-channel buffer of capacity k	$MBUF_k(A,B)$	$\Pi 2: \ 0 \le l(\sigma\lceil A) - l(\sigma\lceil B) \le k$ $(\forall i: \ 1 \le i \le k:$ $0 \le l(\sigma\lceil a_i) - l(\sigma\lceil b_i) \le 1 \wedge$ $a_i \in A \wedge b_i \in B)$
selector 1 to k	$SEL_k(a,B)$	$\Pi 3: \ 0 \le l(\sigma\lceil a) - l(\sigma\lceil B) \le 1$
multiplexer k to 1	$MLX_k(A,b)$	$\Pi 4: \ 0 \le l(\sigma\lceil A) - l(\sigma\lceil b) \le 1$
ordered selector 1 to k	$OSEL_k(a,B)$	$\Pi 5: \ \Pi 3 \wedge (\forall i,j:$ $1 \le i \le k \wedge 1 \le j \le k \wedge i < j:$ $0 \le l(\sigma\lceil b_i) - l(\sigma\lceil b_j) \le 1 \wedge b_i, b_j \in B)$
ordered multiplexer k to 1	$OMLX_k(A,b)$	$\Pi 6: \ \Pi 4 \wedge (\forall i,j:$ $1 \le i \le k \wedge 1 \le j \le k \wedge i < j:$ $0 \le l(\sigma\lceil a_i) - l(\sigma\lceil a_j) \le 1 \wedge a_i, a_j \in A)$
modulo k counter	$CNT_k(a,b)$	$\Pi 7$: $k(l(\sigma\lceil b)) \le l(\sigma\lceil a) \le k(l(\sigma\lceil b)+1)$

In addition to the qualitative capture of a discriminator provided by its characteristic predicate, we can also suggest some quantitative measures. These are an *excess* , a *slack* and a *shift.* The excess of a pair of node symbols (a, b), $a, b \in A_D$ for the trace σ, $\sigma \in X_D$, is defined by the expression $\mathcal{E}(a, b, \sigma) = l(\sigma\lceil a) - l(\sigma\lceil b)$. The slack of a pair of node symbols (a, b), $a, b \in A_D$ for the trace σ, $\sigma \in X_D$, denoted by $\mathcal{S}(a, b, \sigma)$, is the maximum number of symbols a which occur between two adjacent symbols b in σ. The shift of a pair of node symbols (a, b), $a, b \in A_D$ for the trace σ, $\sigma \in X_D$, denoted by $\mathcal{I}(a, b, \sigma)$, is the number of symbols a which have occurred in σ since the most recent occurrence of b in σ.

In the above types of discriminator the excess of pairs of their symbols is either bounded for all $\sigma \in X_D$ (e.g., in $BUF_k(a, b)$, $\forall\sigma : 0 \leq \mathcal{E}(a, b, \sigma) \leq k$) or is linear in the length of σ (e.g., in $OSEL_k(a, B)$, $\forall\sigma : \mathcal{E}(a, b_i, \sigma) = O(kl(\sigma))$), which means that with respect to ports a and b_i, $OSEL_k(a, B)$ can be viewed as a counter[5]. The slacks of all these types are bounded. We can call such types *linear discriminators.* In the same way we might have suggested other types, for instance, those whose characteristic predicate would be based on some nonlinear function F of the trace length: $F(l(\sigma\lceil a)) \leq l(\sigma\lceil b) \leq F(l(\sigma\lceil a) + 1)$.

It is interesting to note that, compared to the slack, which is fairly constant for any trace of a k-place buffer, the shift is a more dynamic characteristic in the sense that, like the excess $\mathcal{E}(a, b, \sigma)$, it varies with the length of the trace.

In addition to the trace length, these measures can also be used for extending the idea of a discriminator, by defining the specification characteristic predicates on them. We can even extend the very idea of input and output ports as sources of events. It is often convenient to regard them as logical values which characterise certain classes of generated traces.

One such example is a frequency differentiator with a maximum allowed shift k. This mechanism can be defined as follows. It has inputs a and b and outputs x and y. Whenever the shift $\mathcal{I}(a, b, \sigma)$ becomes greater than k, the outputs are $x = 1, y = 0$. Similarly, whenever $\mathcal{I}(b, a, \sigma)$ is greater than k, we have $x = 0, y = 1$. Only if both $\mathcal{I}(a, b, \sigma) \leq k$ and $\mathcal{I}(b, a, \sigma) \leq k$, we have $x = 0, y = 0$ (parity state). Thus, the characteristic predicate can be written as: $\Pi = \forall\sigma \in \{a, b\}^* : (\mathcal{I}(b, a, \sigma) \leq k \wedge x = 0, y = 0) \vee (\mathcal{I}(a, b, \sigma) > k \wedge x = 1, y = 0) \vee (\mathcal{I}(b, a, \sigma) > k \wedge x = 0, y = 1)$.

The LPN implementing this behaviour for $k = 2$ is shown in Figure 3. Places labelled with x and y stand for the markings in which $x = 1, y = 0$ and $x = 0, y = 1$ respectively. Places inside the dotted box correspond to the markings in which $x = y = 0$. Transitions labelled with $x+, x-, y+, y-$ denote the change of the state of the logical outputs of the circuit. We could of course find a predicate that would represent this behaviour in a purely event-oriented notation, using the set of events $\{a, b, x+, x-, y+, y-\}$, but it is much more natural to use the level-oriented notation for x and y in this example.

3.3 Abstract design of FIFO buffers

In this section we demonstrate how the above concepts can be used to design a basic unit of a token ring LAN adapter, a FIFO buffer. This is a module capable of receiving up to k items of data, and storing them until they are read, in the same order, from the output port. A typical interface adaptor consists of two types of units: buffering modules, such as FIFOs, and protocol controllers.

Figure 4 shows the adaptor, whose main function is to provide each local subsystem with

[5] Such a straightforward implementation of the counter is not generic however. It is impractical in most cases ($k > 4$), since its size is linear to the magnitude of k. The size of normal counter designs is however $O(logk)$.

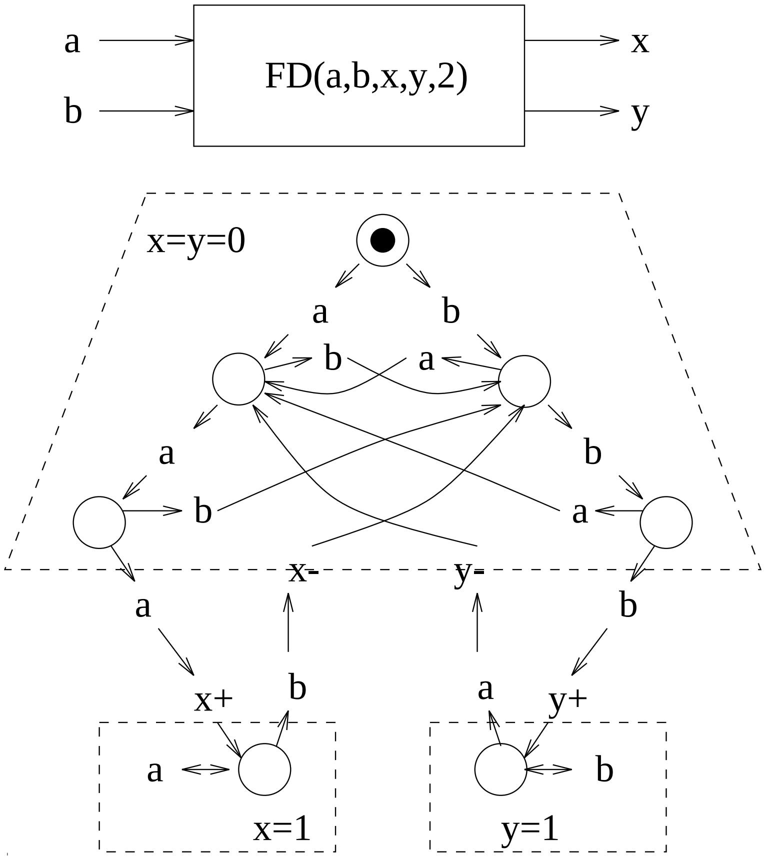

Figure 3: Labelled Petri net for frequency differentiator example

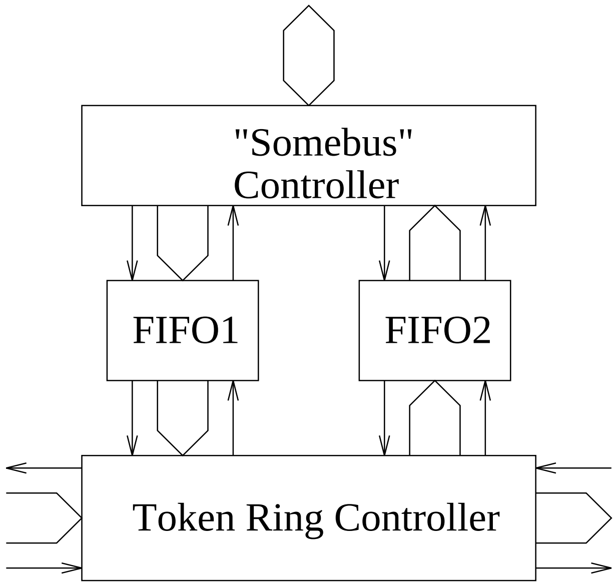

Figure 4: A typical token-ring adaptor

an interface through which it can communicate with other local subsystems. This creates an interconnection service that can be used by the higher level protocol entities. An example of such an adaptor for a fault-tolerant on-board computer was presented in [47].

The structure of the adaptor incorporates a "Somebus" controller, a token-ring controller, and a pair of FIFOs for storing packets containing message bytes. The main function of the "Somebus" controller is to perform the "Somebus" signalling scheme with the local subsystem. A packet in FIFO 1 is either transmitted to the token ring link, or a packet from FIFO 2 is received from the token ring input link and is subsequently delivered to the local subsystem environment.

The reason for incorporating FIFOs into an adaptor is obvious: to maximise the performance of the whole distributed environment, in a manner similar to the VME-controller board [41].

Our design will make full use of our abstraction of control flow. This will enable us to synthesize a control structure (implementing predicate $0 \leq l(\sigma\lceil a) - l(\sigma\lceil b) \leq k$) for the buffer independent of the data path details, even those details relating to the order in which data has to be read from the buffer with respect to the order in which they are written. Thus, our control circuit will be equally usable with other buffer access disciplines, such as LIFO.

3.3.1 Top level specification

We formalize our idea of the FIFO module by introducing a structural model for it. The discriminator $BUF_k(a, b)$ models the FIFO, where a has the following meaning: 'an item of data enters the FIFO through the input port', and b means that 'an item of data is retrieved from the FIFO through the output port'. Note that, since $BUF_k(a, b)$ does not define the order in which items are retrieved from the FIFO, it would be more appropriate to use the more general term "buffer" here. We therefore specify the basic control mechanism of a buffer. This specification is invariant to the discipline of accessing the data path. We thus assume that the data item may be placed in the buffer and taken from it *randomly*.

The FIFO discipline can be specified as follows. Let $d(p_i)$ denote the i-th data value in

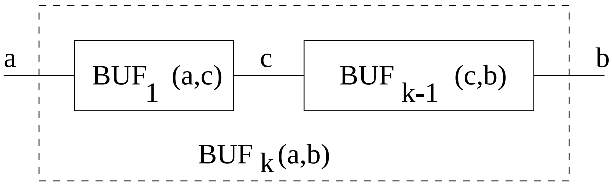

Figure 5: Series (pipeline) buffer decomposition

the ordered sequence passing through the port p. The buffer realises the FIFO discipline if $d(a_0) = d(b_0) \wedge \forall i > 0 : d(a_i) = d(b_i)$ implies $d(a_{i+1}) = d(b_{i+1})$.

We now present two approaches for solving the problem of developing control for the buffer, by decomposing the initial two-port k-place buffer discriminator, with slack $\mathcal{S}(a,b) = k$, into an interconnection of simpler discriminators. The complexity of a discriminator, which can be regarded as a criterion for decomposition, can be measured in terms of the slack between its ports, because this intuitively measures the amount of "memory" that is required between the ports.

3.3.2 First approach

The semantics of a FIFO buffer helps us to arrive at two fundamental ideas about its decomposition. The first idea is to decompose it into a pipeline of buffers of lower capacity, connected in series. Each sub-buffer must have its own storage, and therefore every data item must travel across all sub-buffers before leaving the module. It is easy to prove, by induction on the length k of the buffer, that this organisation ensures the FIFO discipline if each cell (one-place buffer) outputs the same value through port b as it inputs through port a. If we use the discriminator notation for the corresponding LPN, the decomposition of $BUF_k(a,b)$ can be given by $BUF_1(a,c) \parallel BUF_{k-1}(c,b)$ if $k > 1$.

Alternatively, we can build a parallel interconnection of k buffers of capacity 1, which together correspond to a multi-channel buffer of capacity k. We need two additional submodules for organizing the required order between these elementary buffers. The first submodule orders the data items at the input, and corresponds to an ordered selector 1 to k. The second submodule orders the output flow in the same sequence as the first one, and is modeled as an ordered multiplexer k to 1. This decomposition is defined by: $OSEL_k(a,C) \parallel (\parallel_{i=1..k} BUF_1(c_i,d_i)) \parallel OMUX_k(D,b)$ where $C = \{c_1, ..., c_k\}$ and $D = \{d_1, ..., d_k\}$. This organisation also satisfies the FIFO discipline, which can be easily proven using the fact that both selector and multiplexor access the links with the multi-channel buffer in the order of the port subscripts, and that for any $i > 0$ the c_i action precedes the d_i action. Figures 5 and 6 show the corresponding structures of the buffer in the case of series and parallel interconnection, respectively.

Using CWR it can be formally proven that the first structure with the given identification of nodes is, in fact, a discriminator $BUF_k(a,b)$, and the second structure, with its own interconnection of identical nodes of the component discriminators, constitutes a buffer $BUF_{k+2}(a,b)$. Such a proof manipulates characteristic predicates using an algebra of inequalities. From the predicate specifications for the components we can easily see that each primitive component has lower complexity (i.e. slacks) than the original k-place buffer.

The LPN implementations for $BUF_1(c_i,d_i)$ and $OSEL_k(a,C)$ are shown in Figure 7 (a) and (b), respectively. The composite LPNs for both designs can be obtained in a straightforward manner by making identical those transitions whose corresponding nodes are structurally con-

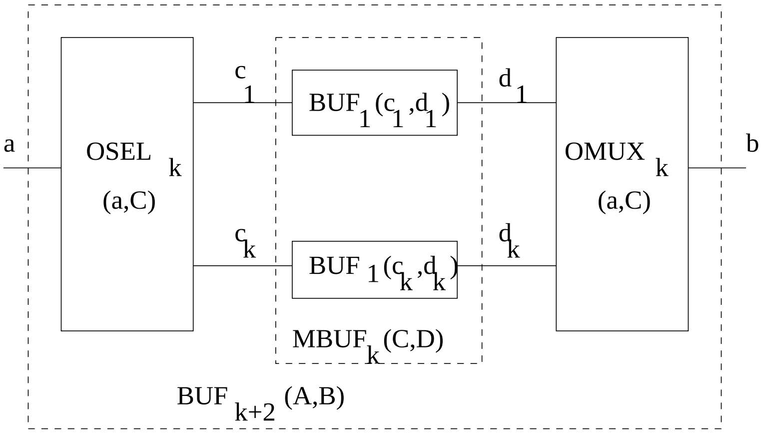

Figure 6: Parallel buffer decomposition

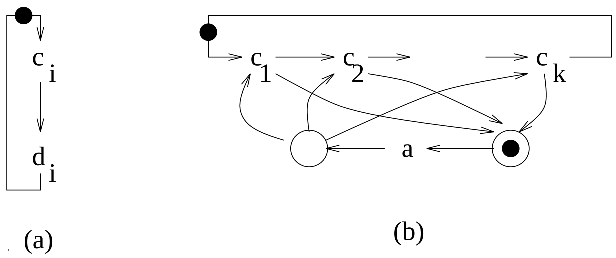

Figure 7: Labelled Petri nets for $BUF_1(c_i, d_i)$ and $OSEL_k(a, C)$

nected in Figure 6. Both implementations can be verified by checking the conformity conditions satisfied by their LPN against the LPN for $BUF_k(a, b)$.

Both solutions have some good properties because of their regularity, and the generic structure of the buffer control circuit. The second solution has an advantage over the first in speed. Although they both have the same throughput, it is easy to see that the propagation delay for one item of data in the first case is proportional to the buffer length. In the second case the delay is constant, and, in the case of the decomposition shown in Figure 6, can be approximated to $3d_1$ where d_1 is the propagation delay for a one-place buffer (for convenience we consider the delay introduced by the single cycle action of selector or multiplexer to be d_1 as well). Although superior in speed, the second solution has a disadvantage in terms of silicon area, which can be crucial if we consider buffers with large capacity. The area of the control circuitry for both these solutions is linear in k, but the second one has a larger multiplicative factor.

3.4 Alternative approach

3.4.1 Counter-based control

How can we limit the size of the control circuit area to make it logarithmic in k? The solution is clearly to be achieved by using a counter as part of the buffer control circuit. The data path in such a buffer will also have to be different from the previous two solutions - it will be based on ordinary memory with built-in write/read address mod k counters. This solution is in line with the FIFO described in [40, 9]. However, in contrast to the approach described in [9], which deals with the post-hoc verification of a design, we are formally synthesizing it at the discriminator level. If we ignore the problem of data path synthesis for now, the main problem is to find an adequate LPN description for a new discriminator - mod k-counter, because it is a part of the control circuit. Furthermore, this counter must be reversible - it must be able to count Up and Down.

The use of a counter to implement the control flow of the k-place buffer is an example of an often used technique: in order to keep the size of the control growing less fast than the size of the data circuitry, one has to add one more data path layer inside the control.

An important detail about implementing a k-place buffer using an ordinary memory and a counter is that both, together with their Write/Read and Count-Up/Count-Down operations, form a critical region, which must be protected from concurrent access by the primary buffer actions Put data and Get data, denoted by a and b respectively. Unlike the first two solutions, where the ordering between the writing into and reading from the buffer location was implemented locally at the level of each primitive 1-place buffer, our solution cannot rely on such an implicit ordering. Thus, although we do not limit the buffer capacity by introducing some sort of mutual exclusion mechanism, we protect our shared resources from potential conflict that may otherwise lead to the problems of non-deterministic execution of an intertwined pair of non-atomic actions [2].

It is important to realise that adding mutual exclusion between a pair of atomic actions does not change their interleaving semantics (it does affect finer forms of semantics, such as true concurrency or step sequences [13]). Therefore, in terms of our trace model of processes and their LPNs, the corresponding behaviours can be regarded as equivalent. It is exactly up to this semantics that the design of a FIFO buffer presented by Sutherland and Dill [40, 9] formally satisfies the specification of a k-place buffer, in which both ports can be considered independent, and thus be activated simultaneously. This means that in a true concurrency framework the design reported in [40, 9] is not a correct implementation of the original specification of a k-place buffer. To prove this inadequacy in such finer semantics, we can assume that if the mutual exclusion device, which resolves potential conflicts of concurrent access for Write and Read operations, is unfair, then we cannot guarantee that our full set of possible interleavings between Write and Read actions is realisable.

The above arguments can be easily illustrated by a comparison of two discriminators. One is the pure $BUF_k(a,b)$ shown in in Figure 8(a), and the other is the parallel composition of LPNs of the same $BUF_k(a,b)$ and a selector $SEL(a,b)$, shown in Figure 8(b). Here, the selector constrains the execution of a and b, by not allowing them to happen concurrently. The corresponding LPNs are shown in Figure 9(a) and (b).

3.4.2 Refining the buffer control structure

The structure shown in Figure 8(c) is a refinement of the previous structure. In order to allow a and b to happen concurrently, we introduce mutual exclusion inside the buffer component. We introduce the arbiter discriminator whose LPN is shown in Figure 9(c). It operates in parallel with two sequencers, SEQ1 and SEQ2. This composition provides control for the unit called Memory Control, by ensuring that the actions on ports a' and b' (access to memory and counter), which are in the critical sections, are mutually exclusive. The interaction with the arbiter requires the following three actions from each side ($i = 1, 2$): “request” (denoted as R_i) to enter the critical section, “grant” (G_i), to permit entry of the critical section, and “done” (D_i) to acknowledge exit from the critical section. The LPN model of this system is shown in Figure 9(d), where the memory control is modelled by the fragment inside the dashed box

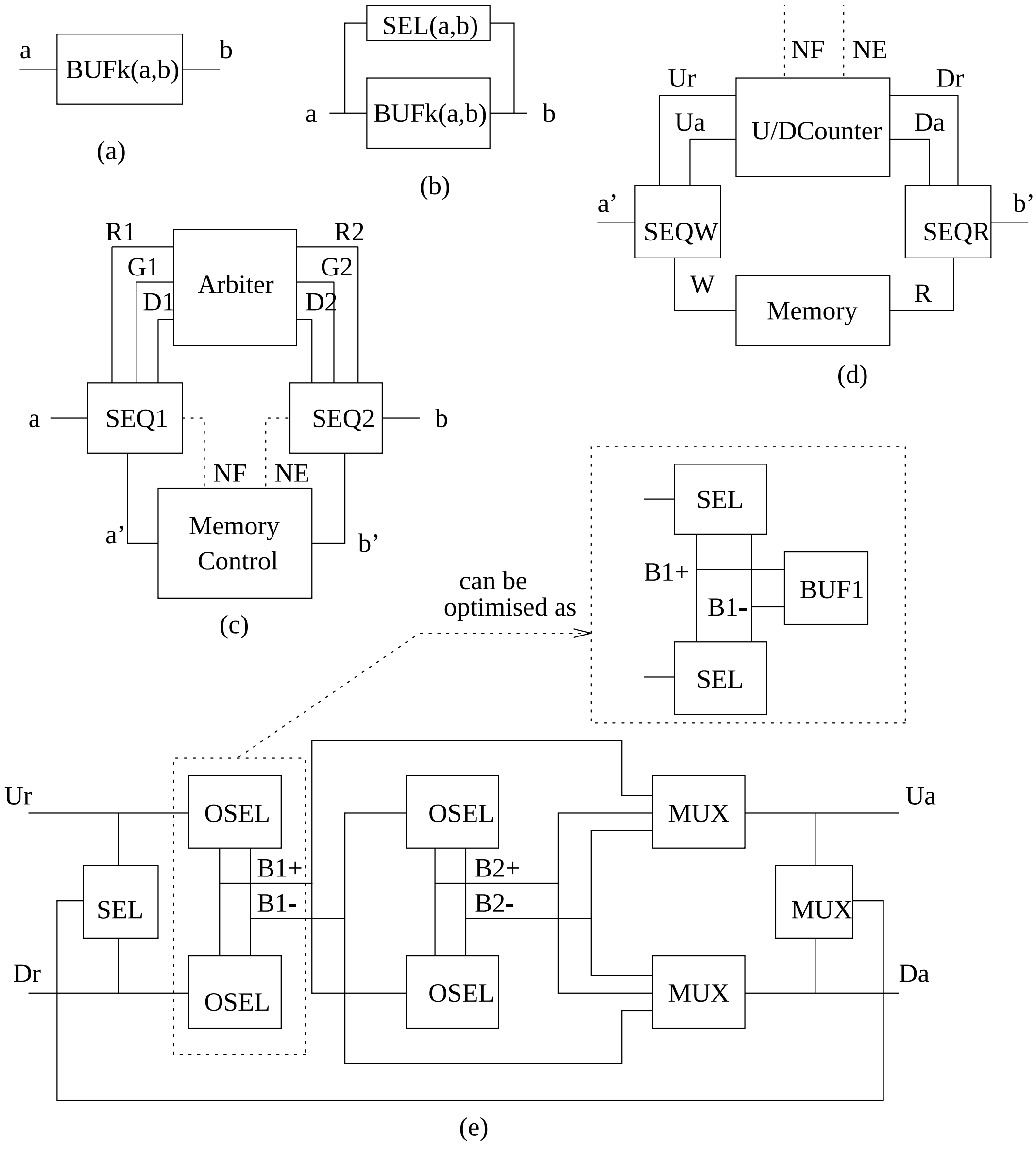

Figure 8: Structural decomposition of buffer

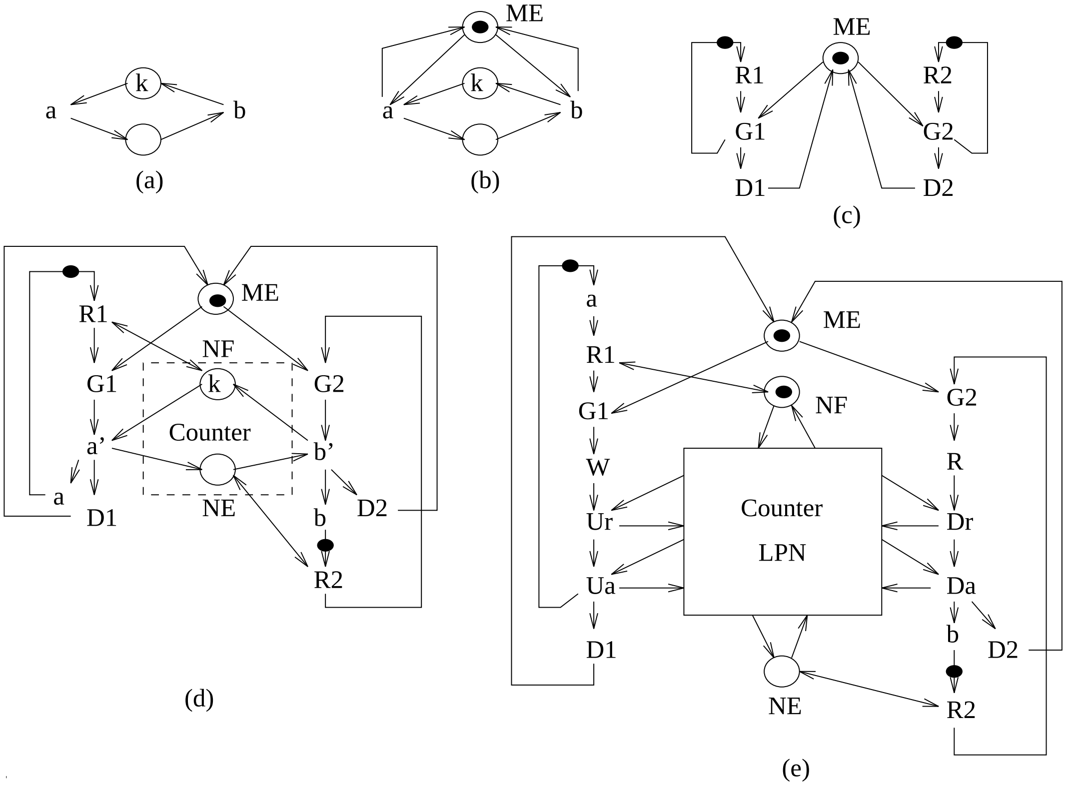

Figure 9: Labelled Petri nets for counter-based buffer design

Counter. This fragment also provides two conditions, Not-Full (NF) and Not-Empty (NE), to the control flow in the sequencers. These conditions can be easily formed by the marking in the places labelled NF and NF. Indeed, if there is at least one token in place NF ($M(NF) > 0$), the buffer is not full. Likewise, the buffer is not empty while there is at least one token in place NE ($M(NE) > 0$). It is easy to prove that the LPN in Figure 9(d) correctly (with respect to the trace semantics) models the k-place buffer. The sequence of $R1, G1, a'$ and $D1$ actions can be compressed into one action a, and this will give us the LPN in Figure 9(b), which is equivalent to the original model of the k-place buffer in Figure 9(a). During such a compression we must take into account the fact that the arc leading from the place to the transition labelled with a', which decrements the marking in NF whenever a' fires, overrides the arc from NF to $R1$, which is purely an enabling arc. Similar reasoning is applied to the place NE and actions b' and $R2$.

Having protected the critical sections of a' and b', we can now refine them to separate the actions on the data path (memory Write (W) and Read (R) operations) from those on the remaining control path (counter increment and decrement). For the counter operations it is convenient to consider separately the actions of request and acknowledgement for both the Up and Down operations. Thus we have the (Ur, Ua) and (Dr, Da) pairs of actions. The structural refinement of the memory control is shown in Figure 8(d), where we have two separate sequencers, SEQW and SEQR, for the Write and Read operations on the buffer. The U/D Counter module, synchronized with the sequencers, is also responsible for producing the NF and NF flags. We abstract from the internal structure and behaviour of the Memory unit, assuming that it works in synchronisation with the control path mechanisms, by having two separate ports W and R. The corresponding LPN is shown in Figure 9(e), where the only part left “hidden” is that of the

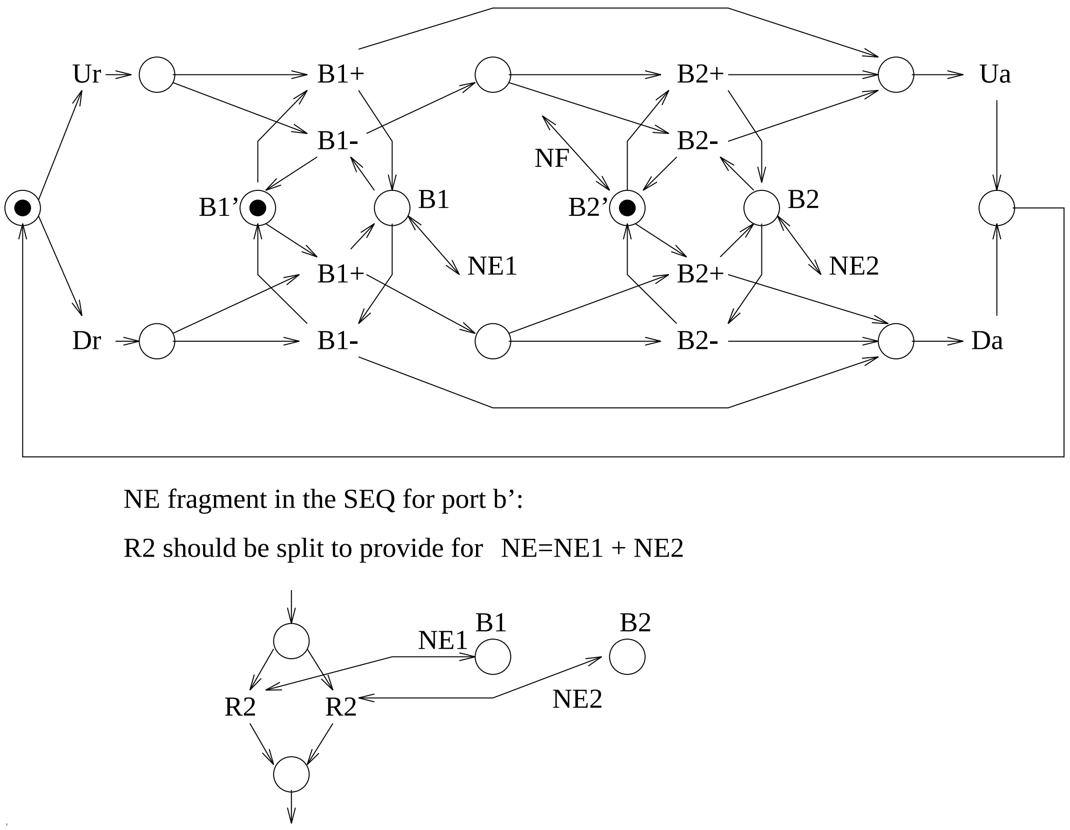

Figure 10: Labelled Petri net model of a counter

Counter LPN.

3.4.3 Constructing the counter model

The structural implementation of the U/D Counter, for the case of the modulo-4 counter, is shown in Figure 8(e). The corresponding LPN, with the help of which one can trace the behaviour of the interconnection, is shown in Figure 10. It should be pointed out that in order to obtain the NF condition we can add an extra modulo-2 counting stage and use the place standing for the 0 value in this stage. We should therefore bear in mind that this simple example of the modulo-4 counter only supports the control of a 2-place buffer. The complement of the most significant bit is used as an NF flag - if this bit is set, the complement place ($B2'$) contains no tokens, and the next attempt to execute $R1$ is not allowed until at least one data item is read from the buffer and the counter is decremented. Unfortunately, we cannot use any single place as an indicator of the NE condition. Such a condition is formed by the logical OR of the values of the counter bits ($B1$ and $B2$). Because Petri nets cannot implement this OR using only one transition, we have to split the action conditioned by the NE flag ($R2$) into two actions, as shown in Figure 10.

We can prove, by induction on the length of the counter (we consider only modulo-n counters with n a power of 2), that the structure built in the same way as the one shown in Figure 8(e), and with its behaviour defined by an LPN similar to the one in Figure 10, satisfies the specification of a modulo-k counter. For the case shown Figures 10 and 11, the fact that the implementation satisfies the specification can easily be shown.

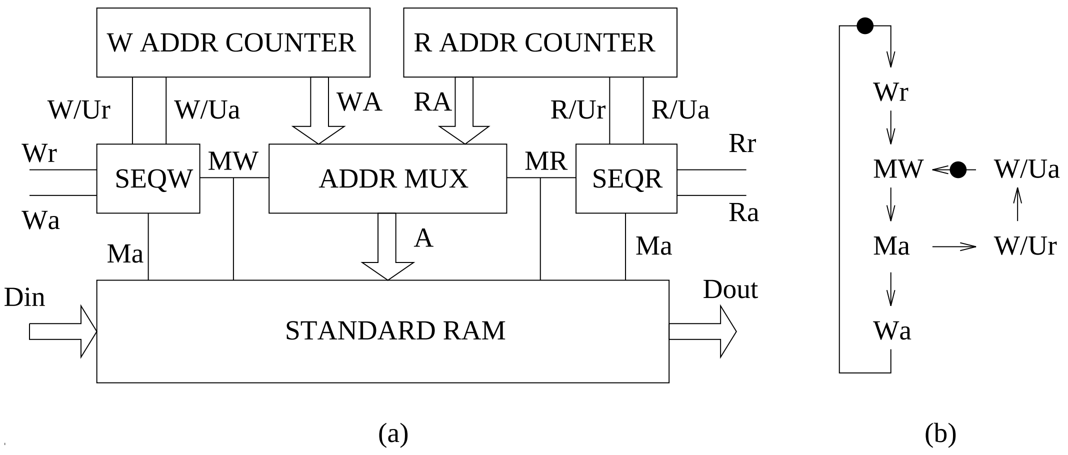

Figure 11: Structural representation of memory unit

3.4.4 Data path refinement

The design of the buffer control has been generic in the sense that we have only implemented the part of the specification defined by $BUF_k(a,b)$. The implementation does not impose any restrictions upon the access to data stored in the memory unit, apart from mutual exclusion between the read and write actions. This is the major distinction of this design methodology compared to those under the first approach (Section 3.3.2).

Now, using the access discipline requirement, we can complete our abstract synthesis of the control by refining the structure of the control "sub-layer" of the memory unit (shown in Figure 8(d) as a black box). This structure is depicted in Figure 11(a). It includes two modulo m ($m \geq k$, where k is size of the buffer) counters producing addresses for writing and reading data to and from the RAM.

Due to the effect of the higher level control in Figures 8(c) and (d) it is impossible to activate both write and read actions simultaneously, or let the R ADDR COUNTER overrun the R ADDR COUNTER. Thus the write address always points (modulo m) ahead of the read address, which ensures correct FIFO order in the buffering of data. ADDR MUX is the address multiplexor. The pair of identical sequencers control the sequence of actions in the structure during the write and read operations. The previously "compressed" actions W and R in Figure 8(d) are refined into the corresponding request-acknowledge pairs, W_r, W_a and R_r, R_a. Figure 11(b) shows the LPN describing one of the sequencers. Due to the separation of the memory address counters for the write and read operations, it is possible to update the address for the next operation concurrently with the acknowledgement to the higher level of control. This does not create any problems, as the critical sections must include only the Up/Down action of the shared counter in Figure 8(d) and the operations on the RAM.

The LIFO organisation requires that the last item pushed into the RAM is the first item to be popped. This enables us to use only one address counter, since the write and read actions are mutually excluded by the buffer control. However, the sequencers for writing and reading must be different. Assume that the address counter normally points to the most recently written memory location. Then, during writing, the address is first incremented and then the memory write operation is executed. During reading, the memory read occurs first, after which the address is decremented. The LPNs for the write and read operations are shown in Figure 12(b) and (c), respectively.

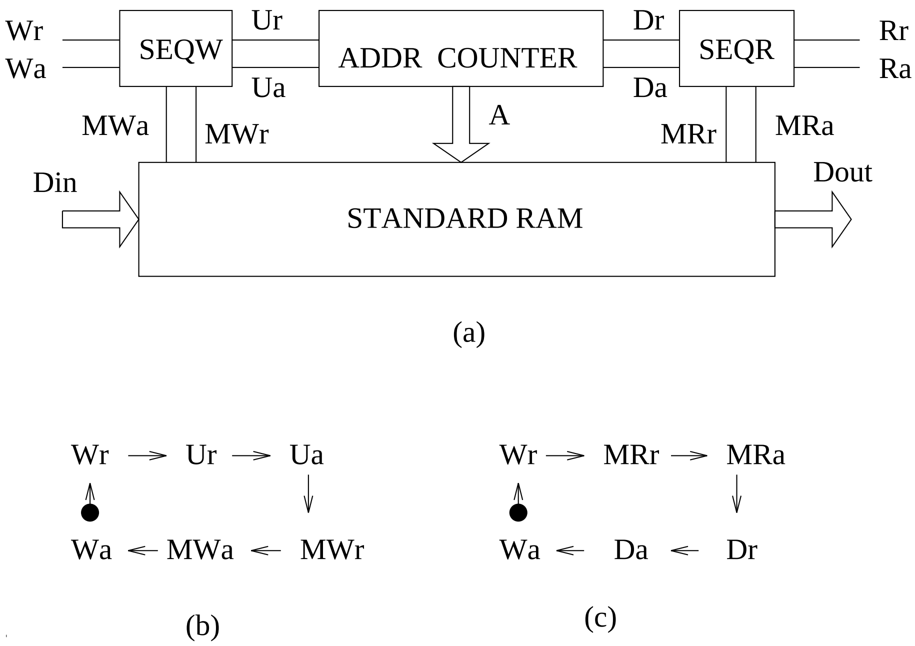

Figure 12: Data path model of a LIFO (stack) memory

4 Logic design

In order to obtain a circuit for each discriminator in the control structure, we need a technique for synthesizing logic. We assume that the initial specification of a component consists of a list of port names, with associated event symbols, and an LPN describing the behaviour. We now outline a method for deriving self-timed circuits for discriminators from their initial behavioural descriptions. Three major steps are required.

The first and most crucial step is to construct a signalling expansion of the abstract behavioural implementation of a discriminator defined on an alphabet of node symbols. As a result we need to obtain a black box with a set of input and output signals. Each port and event in the original specification is associated with a subset of signals and signal transitions. The relationship between original events and signals transitions must be formally and semantically justified. For example, if abstract action a denotes writing to a buffer, the corresponding signal transition can be an assertion of the request signal a_{req}, denoted as $a_{req}+$, in the handshake pair (a_{req}, a_{ack}) associated with port a. Such signal transitions, which are directly related to the original events, are called *critical* signal transitions. The full set of signal transitions can also include some transitions that play an *auxiliary* part, because the designer has the freedom to place them into the specification. This freedom is constrained only by some local ordering relations between critical signal transitions and their auxiliary associates, and can be an important source for speed or area optimisation during logic synthesis. An example of an auxiliary signal transition is the release of the request signal $a_{req}-$. It should be related to the $a_{req}+$ in such a way that they can never be activated simultaneously, and thus should form a sequentially ordered pair. Sometimes, the entire acknowledgement parts $(a_{ack}+, a_{ack}-)$ of a handshake pair associated with just one abstract event can be regarded as auxiliary. The local ordering requirement would be adherence to the totally sequential protocol of actions: $a_{req}+ \rightarrow a_{ack}+ \rightarrow a_{req}- \rightarrow a_{ack}-$.

The initial LPN must therefore be refined in such a way that the new LPN adequately represents the desired behaviour. The conformity between the semantics of the original specification and that of the refined one must be established with respect to the critical signal transitions. Thus the major result of the first step is the refined version of LPN, semantically equivalent to the original LPN.

The second step is concerned with analysing the signalling expansion with respect to correctness and completeness, and with making necessary corrections and modifications while preserving the prescribed order semantics for the initial signal interpretation.

The third step consists of choosing the most effective technique for a self-timed circuit realisation, and converting the signalling description of the module operation into a self-timed circuit.

4.1 Signal interpretation of discriminators

The specific properties of our design objects, discriminators, are as follows. The node symbols correspond to particular events that occur on the attached data paths. These events are themselves decomposable into groups of more elementary events. The latter are related to changes of certain signal values on the connecting lines that correspond to particular nodes of the discriminator. The other aspect of discriminator design is concurrency and essential asynchrony, which is representable even at the abstract synthesis level, but which has the disadvantage that the signalling interpretation of events may raise the amount of parallelism in the model at the elementary level.

We introduce the notion of signalling expansion as follows. Let a discriminator D be specified by a pair $\langle A, P\rangle$ where A is an alphabet of events and corresponding nodes, and P is a characteristic predicate on the set of traces in A^*. According to this specification, D generates the process $\Sigma = \langle A, X\rangle$, where $X = \{\sigma | \sigma \in A \wedge \Pi(t)\}$. Let each symbol in A be assigned to a subset of signals which are binary variables from a finite set Z. Z is naturally associated with a set of allowed signal transitions $*Z = \bigcup_{z \in Z}\{z+, z-\}$, where $z+$ $(z-)$ denotes the transition of z from 0 to 1 (from 1 to 0). Also, $*z$ is used to denote a transition of z without specifying its direction.

The signalling expansion is defined by two aspects, structural and behavioural. The structural aspect is defined by function $\delta : A \rightarrow 2^Z$, which assigns to each abstract port a set of binary signals (i.e., some circuit wires). Transitions on such signals define the behavioural aspect of the expansion, by means of another mapping $\rho : A \rightarrow 2^{*\delta(a)}$. For each event $a \in A$, ρ finds a subset of critical signal transitions (among the set of transitions of signals in $\delta(a)$). The set defined by $\rho(a)$ is called the *critical transition set* of event a. Note that ρ may assign to each event a number of critical transition sets, so that different occurrences of port a can be associated with different transitions.

A simple example showing why $\rho(a)$ for some abstract event a may contain more than one *set* of transitions, is the so called 2-phase signalling [40], which will be considered in the next section. In this signalling expansion *one* abstract action a, such that $\delta(a) = \{x\}$ for some signal x, is associated with *two* critical transition sets $\{x+\}$ and $\{x-\}$. It is important for 2-phase signalling the signal changes $x+$ and $x-$ are assumed to be *semantically equivalent*.

The process $*\Sigma = \langle *Z, *X\rangle$ (where $*X \subseteq (*Z)^*$) is a prefix-closed set of allowed signal changes, and is called a *signalling expansion* of the process Σ.

The signalling expansion of process Σ to $*\Sigma$ must satisfy a number of correctness and completeness rules. These rules are:

1. Preserve the global order semantics. If in an abstract process Σ events a and b are ordered, then the critical signal transitions of the form $*z_a \in \rho(a)$ and $*z_b \in \rho(b)$ must be put in

$*\Sigma$ in the same order as a and b in Σ. Formally, for a given pair of events a and b and a pair of critical transitions $*z_a$ and $*z_b$, the projection of $*\Sigma$ on $\{*z_a, *z_b\}$ is equal to the projection of Σ on $\{a, b\}$ up to the trivial renaming of the symbols through ρ^{-1}.

2. Preserve the local order semantics. If an abstract event a is expanded into a set of transitions $*\delta(a)$, which are ordered by a local signalling protocol defined by a *partial order* on the corresponding signals $*z \in *\delta(a)$, then this protocol has to be compatible with the traces in $*\Sigma$.

3. Guarantee a consistent ordering for the transitions of each signal (signs must alternate, no two transitions of the same signal can be concurrently enabled).

4. Provide the completeness of the signal level specification. This rule reflects the need for a description of the circuit which has a sufficient number of logic variables in Z to allow the derivation of the boolean functions for the non-input signals. The problem of completeness is thus concerned with the problem of the unique assignment of the states of the expansion process to boolean vectors, from which switching functions are derived. In order to meet this requirement, the analysis of the process $*\Sigma$ has to be made using the Complete State Coding property checking(see the next section), and subsequent correction by insertion of additional internal variables.

4.2 Specialised signalling expansion types

In order to avoid the problem of over-generalisation, we specialise the above framework to the two most useful types of expansion.

The first type, which is called the *handshake expansion* [23] of port actions, associates a port a of the process Σ with a pair of signals, a *request* signal a_{req} and an *acknowledgement* signal a_{ack}. The request signal is assumed to be an input and the acknowledgement signal an output, if the port a has been semantically identified in Σ with some action that is *initiated* by the environment and *received* by the process. For example, if a stands for the writing of a data item into a buffer, this action is started by the environment, which asserts the request signal, and is received by the process. When the write action is complete, the process asserts the acknowledgement signal, which is then received by the environment. Conversely, the request signal of an action initiated by the process is an output, and its acknowledge signal is an input.

The second type, called *(simple) signal casting* associates a port a of the process Σ with a single signal z_a, which can either be input or output. The indication of whether z_a is input or output is given on the basis of the semantic interpretation of the port, for which the environment is assumed to be the source or the destination, respectively. This signal expansion type is convenient if we use the port a as an internal signal within a decomposition of the control circuit.

These types reflect the hardware-oriented nature of the interaction between the processes through their shared ports. In hardware, a pair of circuits interact directly, through a set of wires or signals. If these wires are control wires, they are always *directed* in the sense that one circuit is always the sender and the other circuit is the receiver.

It should be noted that in our abstract implementation of the circuit by an LPN we may often need to interpret signals as input and outputs. Some handshake interaction may need to be defined explicitly at the LPN level. For example, in the structure and LPNs of the k-place buffer we used explicit handshake ports to interact with the Up/Down Counter and Memory unit. The handshake refinement at the abstract synthesis level was necessary for the interaction between the sequencer and the counter discriminators, which is of the procedure call type. Such an interaction must ensure that all the actions of the counter during the Up (or Down) operation are executed before the sequencer can proceed further. Thus, if the handshake has been explicitly refined in the abstract synthesis, we usually do not need to expand it further and hence just use the signal casting type.

Another way in which the signalling types may differ is the number of phases that are assumed to be significant in the two-phase cycle of each binary signal [40]. Signalling on port a is called *4-phase signalling* if only one of the transitions of z, either z_a+ or z_a-, is assumed to be *significant* with respect to the action on a. If both transitions are significant and semantically equivalent in representing the occurrence of the same action on a, the signalling is called *2-phase signalling*.

The four types of signalling expansion (all combinations of handshake/casting and 4-phase/2phase) help to make the notions of critical and auxiliary transitions explicit.

For the handshake expansion of a port a which is an initiating port with 4-phase signalling, the critical set includes the pair of asserting actions $a_{req}+$ and $a_{ack}+$. If a is a receiving port, the critical set includes $a_{ack}-$ and $a_{req}+$. The $a_{ack}-$ indicates readiness to receive the request. The global ordering constraint will thus require that both these transitions must always precede any critical transitions that refine actions following the a action in the corresponding traces. For an initiating port with 2-phase signalling, we have to consider two occurrences of a, one which is associated with the critical set including the pair of asserting actions and the other one with the pair of releasing actions. Similar changes take place for a 2-phase receiving port.

For the 4-phase signal casting, only the assertion z_a+ is critical, and it has to be in the same ordering relations with the critical transitions of other actions as a relates to their abstract prototypes. For the 2-phase signal casting, each transition of z is associated with the occurrence of a.

The local order semantics of the two expansion types demands the following:

Handshake. Preserve the 'request-acknowledge' matching. For every trace in $*\Sigma$, the transitions of each handshake pair are totally sequentially ordered as $a_{req}+ \rightarrow a_{ack}+ \rightarrow a_{req}- \rightarrow a_{ack}- \rightarrow a_{req}+ \rightarrow ...$, with the starting change being $a_{req}+$ for the initiating port and $a_{ack}-$ for the receiving port.

Signal casting. For every signal $z \in Z$, the transitions of z are in the same total order in all traces in $*\Sigma$, with the starting change being $z+$.

4.3 Signal Transition Graphs

The Signal Transition Graph is a special case of the Labelled Petri Net model, and is used to describe signalling expansions of LPNs. The major advantage of using Signal Transitions Graphs at the logic synthesis stage is that this model has proved to be the most efficient in defining causality and parallelism at the binary level. Signal Transition Graphs derived from LPN models can be analysed by the same methods and tools as LPNs. This avoids the problem of finding an intermediate notation to prove the semantic relationship between the abstract and logic synthesis models. There are several techniques and tools for synthesis of asynchronous circuits from Signal Transition Graphs [8, 20, 38]. Another important advantage is that such a relatively low level model can be used as a separate specification notation for objects defined directly at the signal level, for example, for specifying bus signalling protocols and controller circuits.

It should be stressed that Signal Transition Graphs represent a narrower class of processes than those that can be generally defined with LPNs. But this narrowness only concerns their alphabet of labels. We do not impose any restrictions upon the structure of the underlying Petri nets, so that all the causality paradigms achievable at the abstract level are preserved at the signal transition level. Furthermore, since it is possible to insert auxiliary signal transitions, we can optimise a design by changing its Signal Transition Graph.

We define a Signal Transition Graph (STG) as an LPN $\mathcal{N} = \langle P, T, H, M^0, *Z, \gamma \rangle$. It is thus a Petri net whose transitions are labelled, through function γ, by the transitions of Boolean variables in Z.

We now show how the four major rules that the signalling expansion has to satisfy are interpreted in terms of the properties of the STG. The first rule is the global order semantic correctness.

4.3.1 The STG expansion

For a given LPN $N = \langle P, T, H, M^0, A, \gamma \rangle$ the STG $\mathcal{N} = \langle P', T', H', M'^0, *Z, \gamma' \rangle$ is called the *STG expansion* of N if it is built from N by expanding the transitions of the N through the mappings $\delta : A \to 2^Z$ and $\rho : A \to 2^{*\delta(a)}$ in such a way that for every $a \in A$ there is a precise definition of the type of the expansion (handshake, signal casting, 4-phase or 2-phase) with the definition of the local signal transition protocol for a, and for every signal $z \in Z$ there is a port that is mapped to a subset of Z involving z. This definition guarantees that every labelled transition in the STG expansion of the net N has some port prototype, and hence some labelled transition associated with it in N.

Having built the STG expansion, we must ensure that this expansion satisfies the original LPN model. Since such an STG is also an LPN, whose transition labels are related to those of the original LPN, the conformity check can be done using the same mechanism used for LPN decomposition.

We call the STG expansion $\mathcal{N} = \langle P', T', H', M'^0, *Z, \gamma' \rangle$ of N a *correct expansion* iff it satisfies the following conditions:

1. For every transition $t \in T$ in N labelled with a port symbol $a \in A$ there is a set of transitions in $U \subset T'$ labelled with the names from one of the critical sets $\rho(a)$. U must be:
   - connected (via some places in P'),
   - closed (no transitions outside U can belong to all paths between pairs of transitions in U),
   - acyclic (there are no loops in the refinement),
   - free from conflict (the occurrence of a transition in U cannot disable another transition in U),
   - if $z+ \in U$, then $z- \notin U$, and vice-versa.
2. an LPN $N' = \langle P'', T'', H'', M''^0, A, \gamma'' \rangle$ can be built from the STG $\mathcal{N}$ by compressing the U fragments, consisting of the transitions labelled with the signal changes from the same critical transition sets into one transition labelled with $a = \rho^{-1}(\gamma'(U))$, where $\gamma'(U)$ stands for a set of signal changes which label the transitions in U. All the remaining (auxiliary) transitions in the STG are not labelled with the symbols of A.

3. N' is conformant to N, i.e. it is its correct LPN implementation. We allow either strict conformance or a more flexible approach with two specification bounds, as defined earlier.

In many cases the designer may not need to verify the STG expansion for conformance, if the STG was built by refining the transitions of the original LPN in accordance with the above rules. The expansion will simply be correct by construction. It is however possible that in some practical situations, the designer would want to modify the STG, for the purpose of optimisation. An independent test for conformance would always enable the designer to verify that the modifications do not bring the behaviour outside the specification bounds.

Let us now turn to the other rules of the signalling expansion.

4.3.2 STG consistency and completeness

Assume that the STG expansion correctly represents the abstract model of the behaviour of the circuit. This expansion must also be correct and complete. The intermediate representation from which one can derive a logic implementation is the state graph, generated by the STG $\mathcal{N} = \langle P, T, H, M^0, *Z, \gamma \rangle$. The mechanism by which the STG generates a state graph is the same as the one which produces the reachability graph for the underlying Petri net. So we require that each marking M, reachable from the initial marking M^0, has a *consistent* labelling with a vector of signal values v^M, where bit v_z^M is the value of signal z in marking M and for each arc in the reachability graph (M, t, M'):

- if $\gamma(t) = z+$, then $v_z^M = 0$ and $v_z^{M'} = 1$,
- if $\gamma(t) = z-$, then $v_z^M = 1$ and $v_z^{M'} = 0$,
- otherwise $v_z^M = v_z^{M'}$.

For any state transition of the above type we say that z is *excited* in marking M if z can change its value. If there is no transition from M in which z can change its value, z is called *stable* in M. We denote the excited variables in state vectors with an asterisk (*). For example, if the state is labelled with the values of three signals z_1, z_2 and z_3 as 1*01, this means that variable z_1 is excited in this state.

We now identify the notion of a marking of the STG with the notion of a state of the circuit described by the STG, assuming that a state is a marking M with consistent labelling v^M.

Note that this labelling associates the M^0 with the initial state of the circuit. Thus the state graph can be built from the STG by assigning to each marking, starting from the initial marking, a binary code. We call an STG *signal-consistent* if each marking in its state graph has consistent labelling.

It is easy to see that for a signal-consistent STG in every labelled trace $\sigma \in L_{*Z}(\mathcal{N})$ the signs of the transitions of each signal $z \in Z$ alternate, i.e. depending on the initial state either $S(z+, z-, \sigma) = 1$ or $S(z-, z+, \sigma) = 1$. This implies that for every signal z no marking can be reached in which two transitions of the opposite signs are enabled. A signal-consistent STG satisfies the local order requirement with respect to the signal casting expansion.

For a given subset of signals, the subset of their signal values in a state is called a *sub-state*. We call an STG *handshake-consistent* if for each handshake pair (z_{req}, z_{ack}), initially set in sub-state 00, the state graph allows only the following transitions between the handshake sub-states: $00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00 \ldots$.

Signal-consistency is a necessary and sufficient condition for an STG to generate a fully labelled state graph[35, 8, 20]. This is perfectly acceptable if the STG is used as initial specification language. However, since we are considering using the STG as an intermediate model, we should be "responsible" for ensuring other types of consistency, according to the requirements of the correct signalling expansion. Thus, the property of general consistency can be specified by the designer. So if any "non-standard" types of signalling expansion are used one can define their consistency (against the abstract behavioural specification) with respect to each such type.

For example, consider two abstract *mutually exclusive* actions, Read and Write, that can be performed in a bus control circuit. Mutually exclusive here means that while a Read action is being performed, no Write action can be activated, and vice-versa. Now, assume that the signalling expansion δ assigns the signal set { *Read_req, Ack* } to Read and the set { *Write_req, Ack* } to Write. Assume also that the behavioural part of the expansions provides some critical transition sets to Read and Write. Let such sets be { *Read_req+* } and { *Write_req+*}, respectively. Now, what will happen to the consistency requirement? It will be specific in the following way. Signals *Read_req* and *Write_req* must never be simultaneously active, i.e. there should be no state in which *Read_req* and *Write_req* are both equal to 1. In other words, the STG will be said to be consistent with respect to the actions Read and Write if it allows only the following trace fragments: *Read_req+* $\rightarrow$ *Ack+* $\rightarrow$ *Read_req-* $\rightarrow$ *Ack-* or *Write_req+* $\rightarrow$ *Ack+* $\rightarrow$ *Write_req-* $\rightarrow$ *Ack-*. Thus, although the sequence *Read_req+* $\rightarrow$ *Write_req+* $\rightarrow$ *Ack+* $\rightarrow$ *Read_req-* $\rightarrow$ *Write_req-* $\rightarrow$ *Ack-* would be perfectly acceptable from the viewpoint of handshake-consistency as defined above, it does not satisfy one further consistency property, that is non-overlapping Read and Write actions.

An STG is defined as *valid* if it is consistent and its underlying Petri net is bounded. Recall that Petri net boundedness guarantees that the state graph is finite.

Intuitively, our aim is to implement the STG as a circuit with one signal for each output signal in Z, where the Boolean function computed by each gate maps each binary value in the state graph to the corresponding implied value for that signal. The implied value for signal z in state M (labelled as v^M) is defined as the complement of v_z^M if z is excited in M, and v_z^M if it is stable. So for example if $v^M = 0^*00$ for signal ordering $z_1 z_2 z_3$, then the implied value of z_1 is 1 and the implied value of z_2 and z_3 is 0. It has been known since the early work on STGs [35, 6] that the existence of a state graph with consistent labelling does not immediately provide the possibility of deriving the implied values for all non-input signals uniquely on the set of binary vectors for the set Z. The problem is the potential existence of two or more semantically different states with the same binary encoding.

It was proved in [28] that the necessary and sufficient condition for implementability of a valid STG is the Complete State Coding (CSC) property. We say that an STG has the CSC property if all markings with the same binary label have the same set of enabled output transitions. It is easy to see that if the STG does not have the CSC property, then a pair of states with the same label will have a different implied value for some output signal z, and there is no Boolean function of the set of STG signals that can characterise z.

A number of techniques have been proposed for checking the satisfiability of an STG to the CSC requirement, and transforming the STG in order to satisfy this property. The most general techniques [20, 16, 45] operate on the state graph, and therefore require high complexity procedures for state assignment. Less general but more efficient techniques operate directly on the STG model, and use various forms of relations between transitions and signals, such as coupledness or lock relations [50, 44, 22]. These techniques work on subclasses of Petri nets such as live-safe free-choice nets. A recently proposed [32] approach, which combines solving the CSC problem with a logic derivation procedure, applies to live-safe free-choice nets and uses the Petri net markings to assist the process of state encoding and Boolean function cover derivation.

The CSC property is characteristic of whether the STG specification is complete in terms of sufficient number of signals to distinguish the circuit states. Thus, if the circuit does not have the CSC property, extra signals should be added to the STG or state graph. Addition of new internal signals must not violate the order of the target signal transitions, but can also be used for optimisation purposes.

In some cases [16] even the most general techniques for the CSC problem fail to insert new signals into the STG without some auxiliary transformations of the STG, such as unfolding. The finite-period unfolding of an STG has a number of instances for each Petri net transition, and decreases the slack between the underlying Petri net transitions while preserving the original trace sets with respect to the signal transition labels. The fact that the slack value can be used as a criterion for the "hidden complexity" of the STG has not been duly recognised yet. Indeed, it is easy to imagine that if a particular signal switches several times while other signals remain unchanged, this could be an indication that a CSC problem exists. It would thus be possible to identify whether a given STG specification is likely to cause the CSC problem, at the abstract synthesis stage. We can use the heuristic that if some event has a slack with all other events of more than 1, this means that the corresponding signalling expansion will have a CSC problem. The measure of slack between a pair of events is indicative of the "buffering characteristic" of the control flow. Hence there is a close relationship between the motivation for the decomposition of a k-place buffer into more simple (in terms of slack) components and solving the CSC problem.

As a result of this observation we suggest that a transformation of the specification has to be done at the level of LPN, where the original LPN model must be decomposed in such a way that for every abstract event *a the slack with other events is minimised*. There must be at least one other abstract event b with which a has a slack equal to 1.

Although using such an heuristic may not give an optimal solution in all cases, the approach seems much more efficient in terms of the overall synthesis process. It matches the concept of decomposition well, and, as the following example shows, leads to a better solution than the unfolding technique proposed in [16].

4.3.3 Two-place buffer example

Consider the abstract specification of 2-place buffer, which is modelled by the unsafe, 2-bounded LPN shown in Figure 13(a). The corresponding STG expansion is shown in Figure 13(b). The port events a and b are modelled by signals a and b using the signal casting method, with $a+$ and $b+$ as critical transitions. It is easy to see that this STG has a CSC problem: signal a can change its value twice before b switches from 0 to 1. The state graph constructed on the reachability graph contains pairs of binary states that are indistinguishable using only two signals. The specification needs extra internal signals to implement enough internal memory to distinguish such markings, in order to derive the logical implementation correctly.

None of the existing STG state assignment techniques and software tools has been able to resolve this example without changing the original order between signals. For example, the method described in [16], which is based on graph colouring, requires unfolding the original STG into two periods before being applied. Such an unfolding is shown in Figure 13(c). Here all the transitions are duplicated, which helps to reduce the slack between the Petri net transitions underlying the signal changes of a and b from 2 to 1. It appears that this method is capable of solving the CSC problem for an STG in which signals are "connected" to each other by the

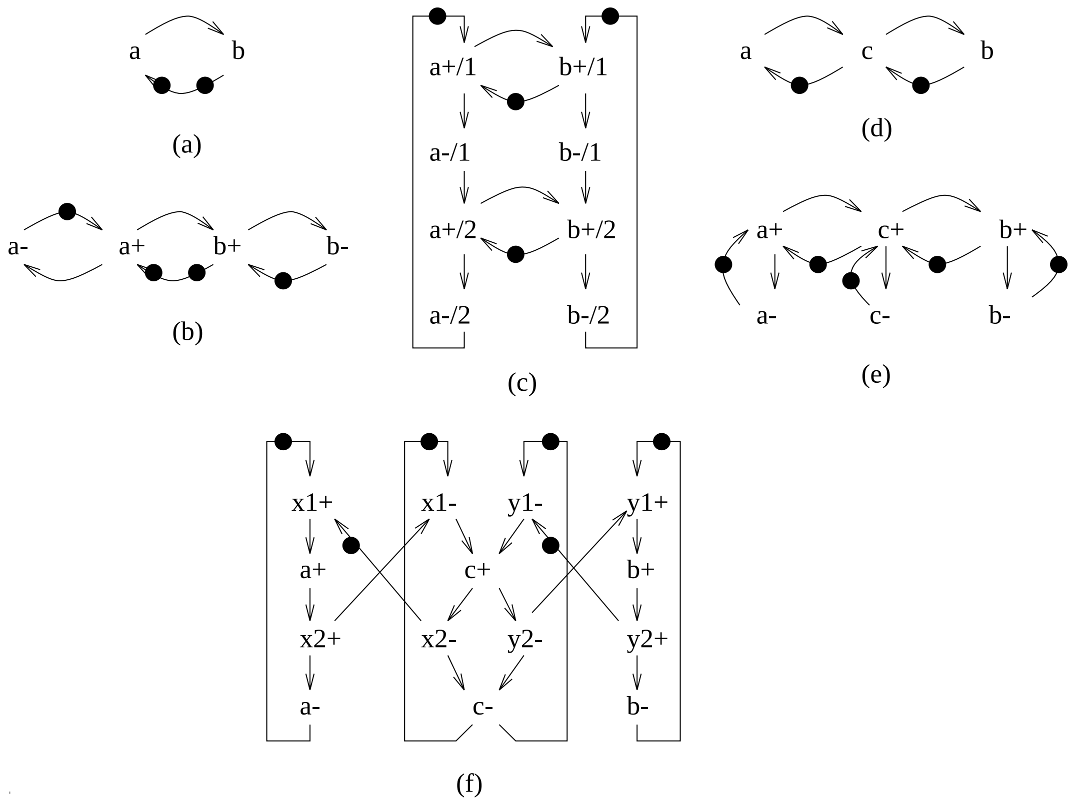

Figure 13: Two-place buffer example

link with slack equal to 1. The disadvantage is that the reachability graph of such a new STG is twice the size of the original graph. The duplication of transitions of the same variable is another complication.

We propose to use the decomposition technique at an earlier, abstract stage, which is aimed at reducing the slack between events in the LPN. Recall that the three decompositions presented earlier for the k-place buffer simplify the model in such a way that for every event a there is another event b with the slack between a and b equal to 1. For such a small value of buffering capacity, the least overhead would be achieved through using the standard series pipeline decomposition of the original LPN in Figure 13(a) into a linked pair of 1-place buffers. The modified LPN is shown in Figure 13(d), where event c is used as an internal buffering event. This event plays, at the abstract level, the part of an extra *explicit* memory which helps reduce the slack and make the STG expansion implementable.

The new STG expansion, in which the slack between the Petri net transitions the changes of a and c and c and b is equal to 1, is shown in Figure 13(e). This STG is simpler than the unfolding in Figure 13(c), and allows use of existing CSC techniques. One of the possible STG versions with CSC, which can be used for deriving a logical implementation, is shown in Figure 13(f).

4.3.4 Logic implementation

Existing methods for STG-based synthesis of speed-independent control circuits require that the STG be *deterministic* with respect to the internal or output signals. In other words, non-determinism can only be used as a form of abstraction to describe more compactly the behaviour of the environment in which the circuit operates. Therefore the STG can only have conflicts (in which one STG transition disables another STG transition) among transitions labelled with input signals. Such an STG is called *output-persistent* in [48] because the underlying LPN is persistent with respect to such transitions. If there is a conflict between transitions labelled with output signals, the circuit derived from the state graph will not be hazard-free. This is because of the inertia of a signal transition, which when enabled and then disabled without changing its value, may produce a glitch at the gate output. To model the behaviour of a circuit in which the output signal transitions can be disabled safely[6], we need a more general model of the circuit implementation than the one defined just by an interconnection of logical gates.

It has been shown that the so called Asynchronous Control Structure, which allows use of multi-output components, can be an adequate structural model for such circuits with output nondeterminism, and yet be defined within the framework of hazard-free behaviour. A classic example of such a structure is a circuit with mutual exclusion elements as primitive components. Freedom from hazards is ensured at the lower implementation level by using special (analogue) interconnections of transistors (see for example the implementation of a mutex element in [37]). We also have to rely on certain assumptions that such interconnections behave as specified at their logical interface.

Although some useful circuits have been designed from STGs with output non-persistency (e.g., design of a low latency arbiter [51]) the complete theory of synthesis of hazard-free circuits with internal non-determinism, based on the foundations of [48], is still under development. In this paper, using the example of a k-place buffer controller, we informally show how circuits with internal non-determinism can be built. We use two different circuit implementation techniques. One is STG-based synthesis, in which the circuit is derived from the state graph under the assumption of 4-phase signalling, so the control flow semantics of the $z+$ transition may not be the same as that of the $z-$ transition. We also apply Sutherland's 2-phase, or transition, signalling, in which $z+$ and $z-$ are semantically equivalent, and the designer may gain performance by utilising both changes of the same control signal. Using our terminology, both transitions of z are critical to represent one action of a port with which we associate signal z.

4.4 Logic synthesis for k-place buffer control

Synthesis from the LPN shown in Figure 9(e) proceeds through a signalling expansion, resulting in a handshake expansion for memory ports W and R, as well as for a and b. The latter are expanded into request and acknowledgement signals between the buffer and its environment: R_{in}, A_{in} for a and R_{out}, A_{out} for b. The remaining events, $R_1, R_2, G_1, G_2, D_1, D_2, U_r, U_a, D_r, D_a$ are expanded through signal casting. The overall signalling expansion is made under the 2-phase signalling discipline, in which all the signals first change from 0 to 1, and then from 1 to 0, with both phases being significant with respect to the control flow. The main advantage of 2-phase signalling is speed, as was noted in [40]. We also benefit from a similar signalling scheme in the internal interfacing with the RGD arbiter, Memory unit and Up/Down Counter.

In the structural model of the buffer controller shown in Figure 8(c) and (d), we explicitly use sequencers to control activation of the other components. We can simplify the design by interconnecting the modules directly, without explicit sequencers, so that they operate in accordance with the LPN specification shown in Figure 9(e).

The combination of the 2-phase and 4-phase signalling schemes takes place during the effect

[6]It has been shown that such circuits cannot be modeled satisfactorily using Boolean logic gates [46].

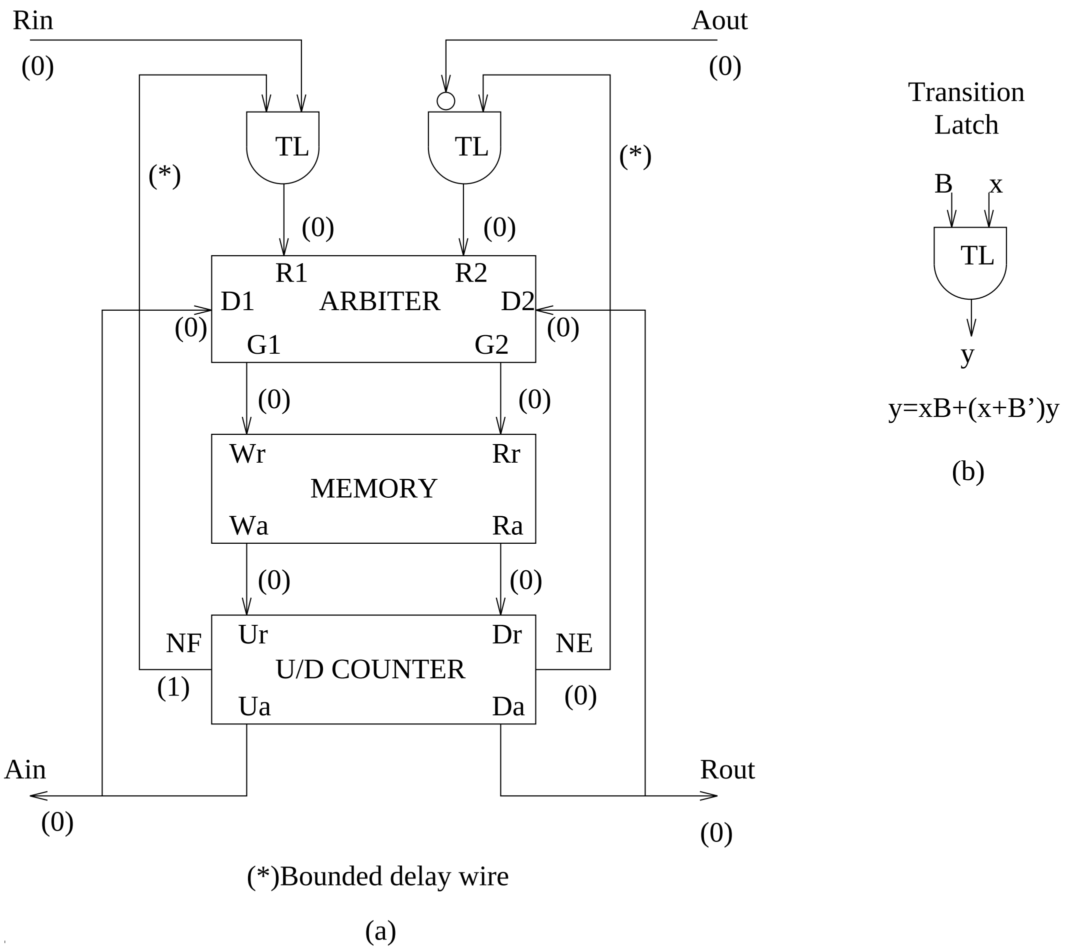

Figure 14: Circuit for counter-based buffer control

of the conditions Not-Full (NF) and Not-Empty(NE) on the control flow. This is where the main difference lies between our design and the FIFO design from [40, 9], and where simplification over the latter is achieved.

The block diagram of the implementation is shown in Figure 14(a). Most parts of this diagram are intuitively clear, as they correspond to the LPN in Figure 9(e). The only units not yet introduced are the Transition Latches (TLs) (Figure 14(b)). These are the kernel of the combination of the two signalling schemes.

Informally, a TL behaves in such a way that it transmits the two-phase signals from its event-based input x to its event-based output y only if $B = 1$. The sources of the Boolean inputs B are the Boolean flags NF and NE. The corresponding conditions in Sutherland's circuit use 2-phase signalling, and their synchronisation could be done using Muller C-elements. The testers, which are used in [40] to generate dynamic flags from the data-path value of the Up/Down Counter, are completely eliminated in our design, at the cost of introducing some bounded wire delay constraints.

The behaviour of a TL element is more complicated (otherwise it would have been just an AND gate !). We need to take into account the dynamic conditions in which both inputs x and B can switch. The best way is to use STGs to specify the TL element. Figure 15 shows such an STG and the state graph it generates, and since this state graph is *semi-modular* (non-input

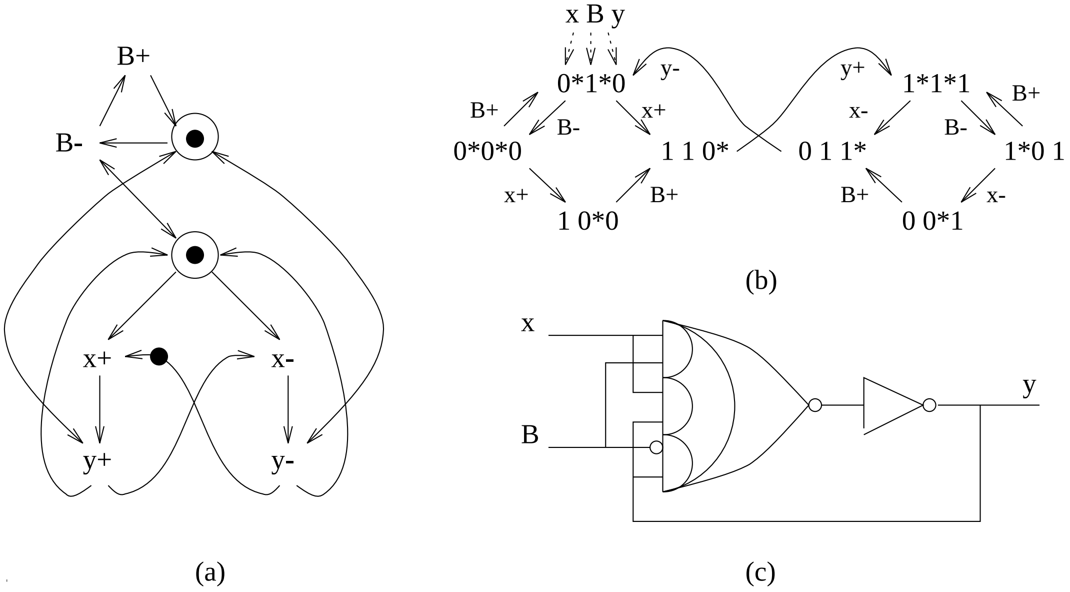

Figure 15: Transition latch synthesis

signal transitions are never disabled [48]) and has CSC with respect to the only non-input signal y, we can derive a logic implementation in the usual way [6, 20].

Note that transition $B+$ is caused by the actions in the part of the control circuit associated with data output (i.e. during a Down operation of the counter). It can thus be concurrent with the changes of x (which are in this case the changes of R_{in}). On the other hand, transition $B-$ (from Not-Full to Full) is controlled by the same (Input or Write) part of the control circuit and it takes place within the critical section protected by the arbiter. We also assume that the change of the NF condition takes place *before* the acknowledgement U_a is generated by the Counter. Therefore, when the next change of x (R_{in}) arrives B (NF) can be safely assumed to be stable. Similar assumptions can be made about the other TL element. As a result, we can introduce a causality condition in the STG such that in the corresponding state graph, when the circuit is in the states 011^* and 110^*, there must be no change of B until the output y has changed, which guarantees semi-modularity.

Due to the presence of both B and its inversion on the inputs of the TL we must make sure that the delay of this inversion is small enough to avoid possible hazards on y. It is easily seen that the circuit is correct and speed-independent under the indicated assumptions about delays. We assume that the two wires labelled with (*) in Figure 14(a) have less delay than the time between the departure of the output signal (A_{in} or R_{out}) of a handshake and the arrival of the input signal (R_{in} or A_{out}).

It is almost inevitable that the combination of the two signalling strategies affects the pure delay-insensitivity. But if our assumptions about wires delays are reasonable, it is not difficult to see that the overall design is much simpler than the one from [40, 9]. Furthermore the hazard-freedom is guaranteed by speed-independence.

The other components in Figure 14(a) are designed as follows (note that no circuits for these components were presented in [40, 9]). The same RGD arbiter can be used as in [40]. Although the internal structure of this arbiter has not been published anywhere, some authors use it as a black box [10]. One possible implementation based on standard building blocks such as C-element, Merge, Mutex and Toggle is shown in Figure 16. The idea of memory design based on

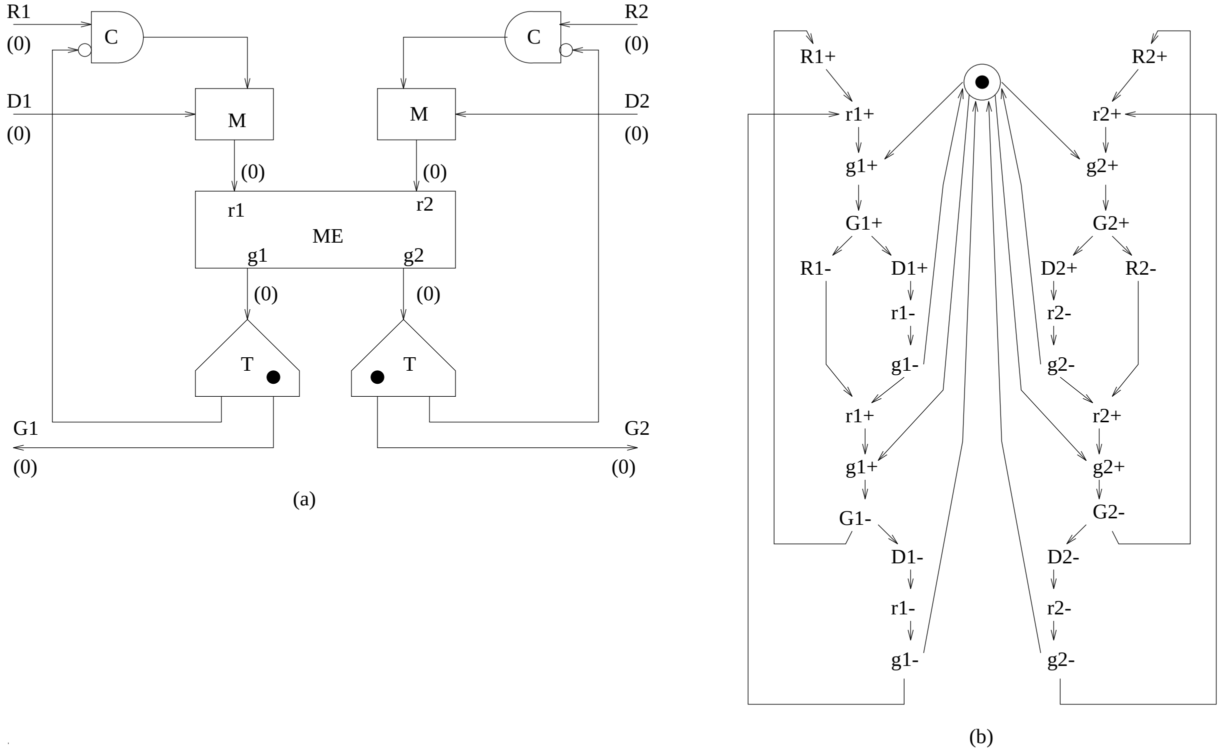

Figure 16: One possible implementation of RGD arbiter

standard RAM was already shown and should not present problems in further refinement.

All the examples of mod n counters, built using Transition Signalling [11, 4] are just Up-Counters, and not reversible counters as our design requires. It is possible to implement our LPN specification of an Up/Down Counter using similar constructs consisting of Toggles and Merges. An example is shown in Figure 17. The difference in the interconnection between the Ack (U_a) and Carry (U_c) outputs of the Up and Down parts ensures correct operation of each stage. We generate the Carry signal in the Up mode if the stage Bit transitions from 1 to 0, whereas in the Down mode the Carry is produced when Bit goes from 0 to 1.

The implementation of the one-bit stage shown in Figure 17(d) is satisfactory in principle, but only under the assumptions about the boundedness of delays in the Merges producing the Bit values.

A different design can be obtained by converting the LPN into an STG with four-phase signalling. This is purely a syntactic transformation, which does not affect the actual operational idea of transition signalling. The STG for one stage of the counter is shown in Figure 18.

This is the set of resulting equations for the counter:

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

These equations can be implemented either as the so-called complex gates [8], with inter-

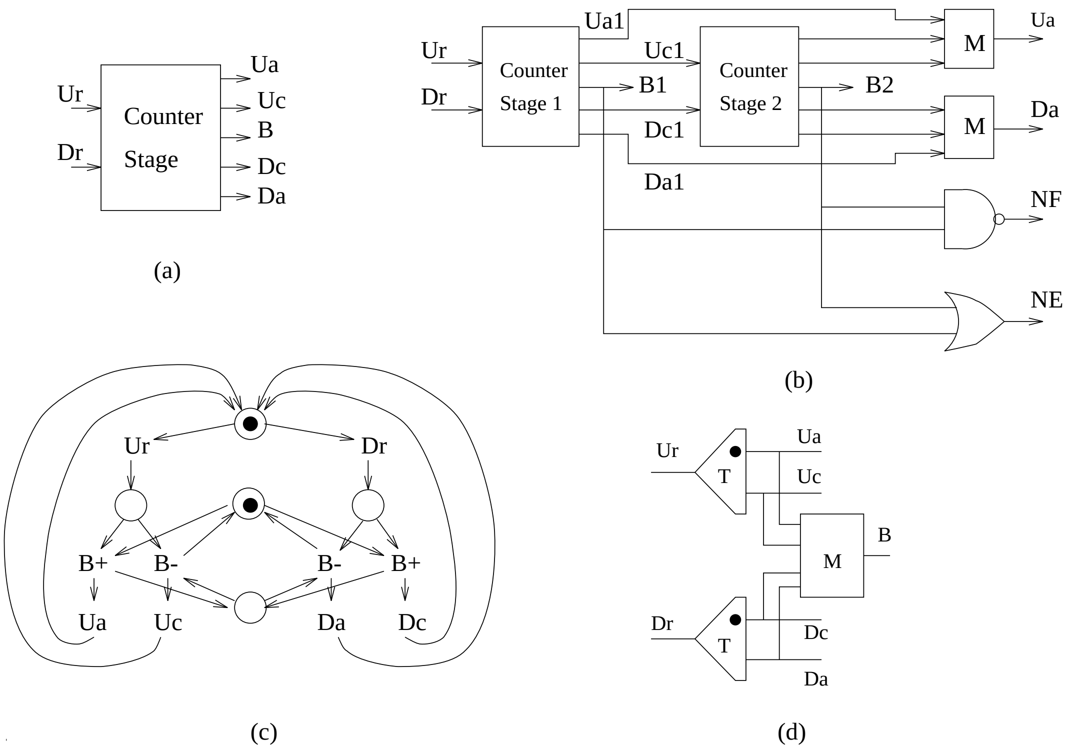

Figure 17: Synthesis of the counter circuit

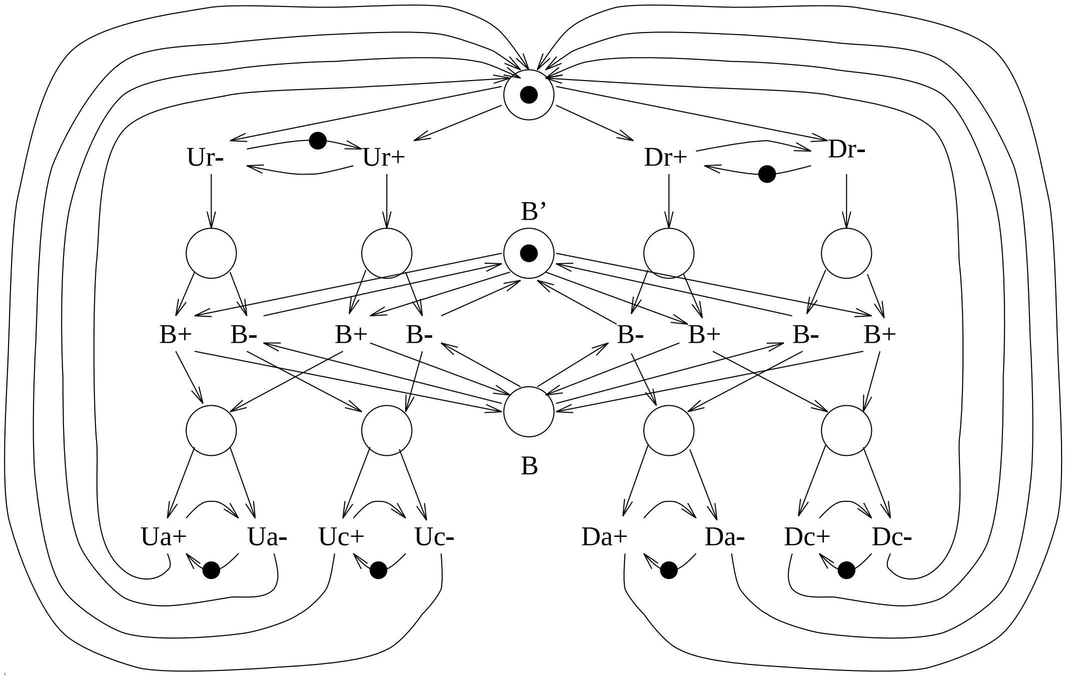

Figure 18: Signal transition graph for a counter stage

nal feedbacks, or as a row of SR-latches with separate two-layer sum-of-products logic for the excitation functions S and R that can be easily obtained from the above equations. For example,

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

In the latter case, one cannot however guarantee that the implementation is speed-independent with respect to the delays of the AND and OR gates involved in the implementation of the S and R functions. Special hazard analysis and elimination techniques, described in [20], can be employed but this will make the final circuit operate under a bounded delay model.

5 Conclusion

The main distinctive aspect of this methodology in comparison with those presented elsewhere [8, 15], is that it provides a unified and rigorous solution to the problem of synthesizing logical implementations for interface hardware, especially for control circuits, from their abstract specifications.

The methodology covers both the abstract and the logic synthesis phases, and offers the following major advantages:

- Better abstraction of the control flow of the synthesized circuit; for example, it is possible to model and design a control circuit for a k-place buffer that can be used for any type of buffer (FIFO, LIFO or RAM buffer).

- Useful heuristics can be derived at a higher level of abstraction; for example, by using the concept of synchronisation slack, it is possible to avoid or resolve the Complete State Coding problem for the control circuit at the abstract specification level.

Although our prime example, a FIFO buffer control, is rather simple, it is possible to apply the same methodology to other types of discriminators, such as a frequency differentiator, a low latency arbiter, and so on.

Other types of control circuits for a LAN adaptor may require signal transition labelling of Petri net models at the initial specification level. This would be the case for the circuit of the "Somebus" controller. Bus protocols are typically defined in terms of signal transitions and do not need an extra level of action abstraction. On the other hand, the specification of the token-ring controller may involve an extra level of control, such as a register transfer control, and this would require using some other modelling technique. Nevertheless, as has been shown in [47], one can again use some form of interpreted Petri nets as a specification formalism, in order to preserve semantic uniformity between the levels of abstraction.

References

[1] P.A. Beerel and T.H. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings of ICCAD'92*, Santa Clara, CA, November 1992.

[2] M. Ben Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall International, London, 1990.

[3] K. van Berkel. *Handshake circuits: an intermediary between communicating processes and VLSI.* PhD thesis, Technical University of Eindhoven, 1992.

[4] K. van Berkel. VLSI programming of a modulo-N counter with constant response time and constant power. In *Asynchronous Design Methodologies: IFIP Transactions A-28*, pages 1–11. North-Holland, Amsterdam, 1993. Proceedings of the IFIP WG 10.5 Working Conference, Manchester, UK.

[5] J.A. Brzozowski and C.-J.H. Seger. Advances in asynchronous circuit theory. *Bulletin of the EATCS*, (42), October 1990. (Part 2 in No.43, Feb. 1991).

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

[7] T.-A. Chu. Automatic synthesis and verification of hazard-free control circuits from asynchronous finite state machine specifications. In *Proceedings of ICCD'92*, Cambridge, MA, October 1992.

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

[9] D.L. Dill, S.M. Nowick, and R.F. Sproull. Automatic verification of speed-independent circuits with Petri net specifications. In *Proceedings of ICCD'89*, Cambridge, MA, October 1989.

[10] J.C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract.* CWI, Amsterdam, 1989.

[11] G. Goossens and M.B. Josephs (Editors). Acid-wg/exact workshop on asynchronous controllers and interfacing. Technical Report EXACT/D.2/IMEC/m3/D1, IMEC Laboratory, September 1992.

[12] D.A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3-4), 1954.

[13] R. Janicki and M. Koutny. On equivalent execution semantics of concurrent systems. In *Lecture Notes in Computer Science, Vol. 266.* Springer-Verlag, 1987.

[14] M. Josephs and J.T. Udding. An algebra for delay-insensitive circuits. Technical Report WUCS-89-54, Dept of CS, Washington University, St Louis, 1989.

[15] M. Josephs and J.T. Udding. Delay-insensitive circuits. In *Lecture Notes in Computer Science, Vol. 458.* Springer-Verlag, 1990.

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

[17] M.A. Kishinevsky, A.Y. Kondratyev, and A.R. Taubin. Formal method for self-timed design. In *Proc. EDAC'91*, pages 197–201, 1991.

[18] A Kondratyev, M. Kishinevsky, and A. Yakovlev. Hazard-free implementation of speed-independent circuits. Manuscript in preparation, 1993.

[19] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the Design Automation Conference*, June 1991.

[20] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, Boston, 1993.

[21] L. Lavagno and A. Yakovlev. Synthesis of speed-independent circuits with conflict resolution elements. Manuscript in preparation, 1993.

[22] K.-J. Lin and C.-S. Lin. On the verification of state-coding in STGs. In *Proceedings of ICCAD'92*, Santa Clara, CA, November 1992.

[23] A.J. Martin. Synthesis of asynchronous vlsi circuits. In J. Staunstrup (ed.), editor, *Formal Methods for VLSI Design*, chapter 6. North Holland, Amsterdam, 1990. IFIP WG 10.5 Lecture Notes.

[24] A. Mazurkiewicz. Concurrency, modularity and synchronization. In *Lecture Notes in Computer Science, Vol. 379*. Springer-Verlag, 1989.

[25] M.C. McFarland. Formal verification of sequential hardware: a tutorial. *IEEE Trans. CAD of integrated circuits and systems*, 12(5), May 1993.

[26] T.H.-Y. Meng, D.G. Messerschmitt, and R.W. Brodersen. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Trans. on CAD*, 8:1185–1205, November 1989.

[27] R.E. Miller. *Switching Theory*, volume 2: Sequential Circuits and Machines, chapter 10. Wiley-Interscience, New York, 1965.

[28] C.W. Moon, P.R. Stephan, and R.K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *International Conference on CAD*, 1991.

[29] D. Muller and W. Bartky. A theory of asynchronous circuits. In *Annals of Computation Laboratory*, pages 204–243. Harvard University, 1959.

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

[31] S.M. Nowick and D.L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proceedings of ICCAD'91*, Santa Clara, CA, November 1991.

[32] E. Pastor and J. Cortadella. Polynomial algorithms for the synthesis of hazard-free circuits from signal transition graphs. In *Proceedings of ICCAD'89*, Santa Clara, CA, November 1993.

[33] I. Reicher and M. Yoeli. Net-based modeling of communicating parallel processes with applications to VLSI design. Technical Report 532, Technion, Haifa, 1988.

[34] M. Rem. Concurrent computations and VLSI circuits. In *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 399–437. Springer-Verlag, Berlin, 1985.

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

[36] C. L. Seitz. *System timing. In: Mead, C.A. and L.A. Conway, Introduction to VLSI Systems*, pages p. 218–262. Addison-Wesley, 1980.

[37] C.L. Seitz. Ideas about arbiters. *Lambda*, 1(1):10–14, 1980.

[38] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, R.K. Brayton H. Savoj, P.R. Stephan, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California at Berkeley, 1992.

[39] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1985.

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

[41] Breaking the speed barrier on the VME-bus. *Electronics*, 26:58–60, 1986.

[42] J.T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1:197–204, 1986.

[43] S.P. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, New York, 1969.

[44] P. Vanbekbergen et al. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *Proceedings of Int. Conf. on CAD (ICCAD'90)*, pages 184–187, November 1990.

[45] P Vanbekbergen, B. Lin, G. Goossens, and H. De Man. Generalized state assignment theory for transformations on signal transition graphs. In *Proceedings of ICCAD'92*, Santa Clara, CA, November 1992.

[46] V. Varshavsky, M. Kishinevsky, V. Marakhovsky, V. Peschansky, L. Rosenblum, A. Taubin, and B. Tzirlin. *Self-Timed Control of Concurrent Processes*. Kluwer AP, Dordrecht, 1990.

[47] V.I. Varshavsky, V.Y. Volodarsky, V.B. Marakhovsky, L.Y. Rosenblum, Y.S. Tatarinov, and A.V. Yakovlev. Structural organisation and information interchange protocols for a fault-tolerant self-synchronous ring baseband channel. *Automatic Control and Computer Science (translation from Avtomatika i Vychislitelnaya Teknika)*, 22(Part 4):44–51, 1988. see also: Vol. 22, Part 6, p. 59-67, and Vol. 23 (1989), Part 1, p. 53-58.

[48] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified STG model for asynchronous control circuit synthesis. In *Proceedings of ICCAD'92*, Santa Clara, CA, November 1992.

[49] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified STG model for asynchronous control circuit synthesis. *Formal Methods in System Design*, 1993. Accepted for publication.

[50] A. Yakovlev and A. Petrov. Petri nets and parallel bus controller design. In *Proc. 11th Int. Conf. on Applications and Theory of Petri Nets, Paris, June 1990*, pages 244–264, 1990.

[51] A Yakovlev, A. Petrov, and L. Lavagno. High speed asynchronous arbiter. Technical Report 427, University of Newcastle upon Tyne Computing Science, May 1993.

[52] A.V. Yakovlev. Analysis of concurrent systems through lattices. Unpublished manuscript, July 1991.

[53] A.V. Yakovlev. Synthesis of hazard-free asynchronous circuits from generalised signal-transition graphs. In *Proceedings of the Sixth International Conference on VLSI Design*, Bombay, India, January 1993.