Concurrency Models for Designing Interface Logic in Distributed Systems

A.V. Yakovlev
Concurrency Models for Designing Interface Logic in Distributed Systems

A.V. Yakovlev

Abstract

Asynchronous digital interface circuits exhibit a high degree of concurrency. Self-timed implementation is the most appropriate design discipline for them. We analyse a number of concurrency models with respect to their capacity to describe certain types of self-timed, or delay-insensitive, modules and systems. We introduce two major stages in the design process: abstract synthesis and logic design. They involve formal manipulation of both structural representations (discriminators) and behavioural descriptions by labelled Petri nets and signal graphs. In greater detail we examine the signal graphs, which are subject to formal treatment and mechanical translation to delay-insensitive circuits. Two examples of designing logic for interface adaptors effectively illustrate the approach.

Series Editor: M.J. Elphick

© 1989 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

YAKOVLEV, Alexandre V.

Concurrency models for designing interface logic in distributed systems.
[By] A.V. yakovlev

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1989.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 285)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

Asynchronous digital interface circuits exhibit a high degree of concurrency. Self-timed implementation is the most appropriate design discipline for them. We analyse a number of concurrency models with respect to their capacity to describe certain types of self-timed, or delay-insensitive, modules and systems. We introduce two major stages in the design process: abstract synthesis and logic design. They involve formal manipulation of both structural representations (discriminators) and behavioural descriptions by labelled Petri nets and signal graphs. In a greater detail we examine the signal graphs, which are subject to formal treatment and mechanical translation to delay-insensitive circuits. Two examples of designing logic for interface adaptors effectively illustrate the approach.

About the author

Mr. Yakovlev was a visiting researcher at the Computing Laboratory from 1984 to 1985.

Suggested keywords

BUS CONTROLLER  SELF-TIMED CIRCUITS
CONCURRENCY MODELS  SIGNAL GRAPHS
DISCRIMINATOR  SIGNALLING EXPANSION
FIFO BUFFER  SPECIFICATION
IMPLEMENTATION  TOKEN-RING LAN ADAPTOR
LABELLED PETRI NET  TRACE THEORY
PROCESS

Suggested classmarks (primary classmark underlined)
Dewey (18th):  621.38173  001.64404
U.D.C.:  621.3.049.771.14  519.687
CONCURRENCY MODELS FOR DESIGNING INTERFACE LOGIC IN DISTRIBUTED SYSTEMS

A.V. Yakovlev
Leningrad Electrical Engineering Institute
Computing Science Department
Leningrad 197022
U.S.S.R

Abstract

Asynchronous digital interface circuits exhibit a high degree of concurrency. Self-timed implementation is the most appropriate design discipline for them. We analyse a number of concurrency models with respect to their capacity to describe certain types of self-timed, or delay-insensitive, modules and systems. We introduce two major stages in the design process: abstract synthesis and logic design. They involve formal manipulation of both structural representations (discriminators) and behavioural descriptions by labelled Petri nets and signal graphs. In a greater detail we examine the signal graphs, which are subject to formal treatment and mechanical translation to delay-insensitive circuits. Two examples of designing logic for interface adaptors effectively illustrate the approach.

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 graphs, implementation.

1 Introduction

Modern technologies allow the construction of VLSI circuits whose internal behaviour exhibits a high degree of parallelism. In order to ensure that these circuits operate correctly under the presence of such undesired phenomena as electronic arbitration and metastability, higher values of wire vs gate delay ratios leading to signal discrepancies, parametric instabilities of gates etc., their design is performed in delay-insensitive or self-timed fashion [1,2]. The most widely used examples of concurrent hardware are regular structures like pipeline and wavefront arrays which are easily decomposed in a sequential, parallel or recursive way. On the other hand, such objects as asynchronous interface controllers, which are a lot less regular but can be equally concurrent, are far from being subjected to formal treatment, as they have been designed mainly by engineers who tend to use timing diagrams or flow charts. The necessity of formal treatment of self-timed design issues is obvious due to the complexity of the construction of delay-insensitive circuits, which has been a major reason of impediment to that approach among the wider hardware engineering community. The ultimate goal of our research is the mechanization of the design process to such a degree that it will be easily incorporated into the overall CAD environment for developing

*In 1984/85 on leave to Computing Laboratory, University of Newcastle upon Tyne, as a visiting researcher.
distributed systems, for example, for translating a physical layer protocol specification into a collection of self-timed modules.

The paper aims at the following major goal: to demonstrate the techniques of using some formal models of concurrency for constructing basic units of a typical interface adaptor structure. These techniques accommodate a stepwise design procedure that may involve the following steps. The first step is concerned with architectural design whose main result is a composition of functionally independent units specified in an abstract manner using such notions as symbolic events, traces of events and characteristic predicates on traces. Each unit is characterized as a certain type of discriminator. During the second step, the following issue is tackled. If the complexity of subsequent translation into a circuit is high enough then the unit is further split until either a standard (library) self-timed circuit element for each sub-unit can be found, or until it is possible within a reasonable limit of complexity to create an internal dynamic description for each new sub-unit discriminator. The third step converts the internal abstract dynamic (behavioural) specification into its logic version by a so-called signalling expansion.

During the fourth step, the signalling expansion is verified with respect to its correctness and completeness thereby providing the final model of the discriminator circuit operation from which the last, fifth step derives the system of Boolean equations characterizing the result of the design process.

First, we present a brief review of a number of concurrency models that support the VLSI design process, as known from the literature. Models are assessed for their capacity to describe certain types of self-timed modules and systems. Then we introduce a general structure for a typical interface adaptor for a LAN token ring architecture from which we deduce that two different design strategies are to be applied to the FIFO buffer unit and bus controller unit. This difference relates only the abstract synthesis stage.

In a greater detail we then examine the notion of a discriminator (which is the major structural issue), and the most appropriate behavioural formalism at the logic design stage (which is a signal graph model). Signal graphs can be an equally suitable formalism for hardware designers as timing diagrams but are also subject to formal treatment and mechanical translation to delay-insensitive circuits.

We end the paper with two design examples of using our formal techniques for the construction of self-timed circuits for a FIFO buffer of capacity 1 and a master controller for bus data read operations.

2 Models for Self-timed Hardware Design

A self-timed system is often defined as a collection of self-timed modules, or elements, which communicate through asynchronous protocols [3]. Such a system does not require a global clock. All system-level events are ordered in time by the causal relationships among the module actions. The order established by the designer must be further preserved in a final circuit, thereby guaranteeing the correct operation independently of element and wire delays.

The evolution of logic design methods shows that the most commonly used method of asynchronous circuit design has for many years been the Huffman state machine [4]. However, this and other methods based on asynchronous automata are increasingly inadequate for VLSI implementation [5]. First, they are not independent of element and wire delays. Second, they are not susceptible to a stepwise refinement of parallelism within circuits. Third, they require solving the state assignment problem, which can be algorithmically complex even for rather low-dimension circuits. The pioneering works of D.E. Muller on speed-independent logic [6] suggests some nice formalisms. For example, the Muller state
diagram, an oriented graph defined on a full state set (a set of binary N-tuples), can properly represent the concurrent switching processes of circuit subcomponents by using the excitation mechanism.

The existing formal models can be split into four groups: (i) graphical notations, state or event oriented, (ii) symbolic notations, (iii) models based on high level programming languages, and (iv) combined models.

The major representatives of the type (i), apart from finite-state machines, are signal Petri nets and graphs [7,8], transition diagrams and parallel flow charts [3]. The main principles of building such models are the same: a set of vertices (events) is mapped onto the set of known actions performed on control and data variables. Local pairwise relationships correspond to one or another class of causal ordering of events in a system. The verification procedures for such models are relatively straightforward and the direct translation of correct specification into a self-timed circuit is normally done by the correctness-by-construction principle. However these models are good mainly for control circuits. The most impressive formalism of the type (ii) is a trace theory and its program notation [9]. The latter includes some kind of regular expression augmented with a comma operator ('weave' composition) and a number of structural primitives allowing the description of both parallelism and modularity of a designed component. This approach can be useful for obtaining more complex behaviour from its simpler forms by means of composition techniques. The models of the type (iii) are mostly functional programming languages. They are subject to the kind of silicon compilation where each operator is translated into a self-timed template which can be properly parametrized, and thereafter yield a self-timed circuit cell. At the moment this approach [10] is limited to objects with fairly primitive control flow but with a wide range of data typing capabilities [11]. The members of the group (iv) are characterized by the combination of tools of first three groups. For example, the initial specification is done with a high level programming notation, then it is converted to an 'object code' which can be a data-driven Petri net [12]. Then the 'object code' is verified and further translated to a self-timed circuit where each circuit cell corresponds to some particular fragment of the Petri net. The number of such fragments and corresponding cells is very limited. This approach is still rather problem oriented, e.g. for signal processing circuits, but nevertheless seems attractive.

The analysis of the above (and many other) tools shows that the usefulness of a formal model for the self-timed circuit design is determined by a large number of criteria. For example, it is affected by the structure type (regular vs non-regular, or data flow vs control flow), the degree of parallelism and data dependence, the necessity of data typing, the degree of delay independence (delay-insensitivity on a transistor, gate or module level). Henceforth, we consider control circuits with generally non-regular structure, limited data dependence and delay-insensitivity with respect to the gate level. The prototype for our basic logic design formalism, signal graphs, is a timing diagram whose application area will be quite obvious to hardware designers.

3 An example: a token ring LAN adaptor

A typical interface adaptor may consist of two types of unit: buffering modules, say FIFOs, and signalling protocol controllers. Fig. 1 shows an example of the token ring LAN adaptor whose main function is to provide each local subsystem with interface medium through which the subsystem can communicate with other local subsystems, thereby creating an interconnection service that can be used by the higher level protocol entities.

The structure of the adaptor incorporates two controllers: 'Somebus' controller and token-ring controller, and a pair of FIFOs for storing packets containing message bytes.
The main function of the 'Somebus' controller is to perform a standard 'Somebus' signalling scheme of the local subsystem in the right order to the packet (byte by byte) in FIFO 1 which can be further transmitted to the token ring link by the token ring controller, and to retrieve the packet (byte by byte) from FIFO 2, if it has been received from the token ring input link, with subsequent delivery of the packet to the local subsystem environment.

The reason for incorporating FIFOs into an adaptor is obvious: to keep the performance of the whole distributed environment at its highest rate, with much the same effect as has been recently reported for the VME-controller board [13].

We want to design the above modules in a self-timed fashion. The design procedures for FIFOs and controllers are fairly different and will be outlined for some pieces of logic in Section 6.

4 Abstract synthesis

4.1 Theoretical background

We consider an abstract model of mechanism D with a finite set of nodes which are labelled with distinct symbols from an alphabet \( A = \{a_1, a_2, \ldots, a_n\} \). To each labelled node we relate an event whose occurrence manifests itself by adding corresponding symbol(s) to the sequence of symbols of previous events.

The formalism that we use for the abstract symbolic specification of mechanisms is trace theory [9]. We are especially concerned with such notions as processes and specifications.

4.1.1 Processes

A finite-length string of symbols is called a trace. The set of all traces with symbols of alphabet \( A \) is denoted by \( A^* \).

The process is a pair \( T = (< A, X >) \), where \( A \) is an alphabet and \( X \) is a non-empty prefix-closed set of traces, \( X \subseteq A^* \), i.e. \( X \neq \emptyset \), \( X = \text{pref}(X) \) where \( \text{pref}(X) \) denotes a set \( X \) extended with all prefixes of traces in \( X \) including the empty trace \( \epsilon \).

The key operations on processes are projection and weaving.

The projection of trace \( t \) on alphabet \( A \), denoted by \( t[A] \), is obtained by removing from \( t \) all symbols that are not in \( A \). The projection of process \( T = (< B, X >) \) on \( A \), denoted by \( T[A] \), is defined by \(< B \cap A, \{t[A] | t \in X\} > \).

The weave of processes \( T = (< A, X >) \) and \( U = (< B, Y >) \), denoted by \( T \bowtie U \), is a process defined by

\[ < A \cup B, \{t \in (A \cup B)^* | t[A] \in X \land t[B] \in Y \} > \]

The projection is interpreted as an abstraction (or reduction) of a process with respect to some part of its components, and the weaving presents the most natural form of the composition of a pair of processes which yields a new process.

4.1.2 Specification

There are many ways in which a process may be specified. One may use enumeration of all allowed traces that may be generated on its events. This approach is perhaps the most natural to the starting point of the synthesis procedure when one does not know the internal dependencies between events, and simply gives the behaviour which shows how the latter is seen from the outside world.

However, such an enumeration can be very complex when the process is highly concurrent and, hence, such a specification would be impractical.
The most compact (though equally strict) way to formalize our initial knowledge of the process is to define a characteristic predicate specifying what traces do belong to the trace set of the process, thereby avoiding the enumeration of traces themselves.

The specification of a process \( T \) is a pair \( \langle A, P \rangle \), where \( A \) is an alphabet and \( P \) is a predicate on \( A^* \) such that \( P(s) \) holds. \( S = \langle A, P \rangle \) specifies the process

\[
\langle A, \{ t \mid t \in A^* \land (\forall s : s \leq t : P(s)) \} \rangle
\]

In order to manipulate with processes and their compositions in terms of their specifications we also mention the so-called 'Conjunction-Weave Rule' (CWR) stating that if \( \langle A, P \rangle \) and \( \langle B, Q \rangle \) are specifications of processes \( T \) and \( U \), respectively, then the process \( T \circ U \) (weaving of \( T \) and \( U \)) is specified by \( \langle A \cup B, \forall t : P(t[A] \land Q(t[B]) \rangle \) or more briefly \( \langle A \cup B, P \land Q \rangle \).

4.1.3 Abstract implementation by labelled Petri nets

We now introduce the concept of abstract implementation of a process. Such an implementation gives the internal causal relationship between the process events performed on the process border. The set of traces of a given process can be generated by some underlying formal dynamic model, for example, by a labelled Petri net.

The labelled Petri net (LPN) is a triple \( \langle PN, A, \gamma \rangle \), where \( PN \) is a Petri net \( \langle P, T, H, M \rangle \), where \( P \) is a set of places, \( T \) is a set of transitions, \( H \) is an incidence function on \( T \times P \) and \( P \times T \), and \( M \) is an initial marking, \( A \) is an alphabet of events and \( \gamma : T \rightarrow A \) is a partial function that labels transitions from the set \( T \) by symbols in \( A \).

The LPN shown in Fig. 2(a) generates the process

\[
\{ a, b, c \}, \{ e, a, b, ab, ba, abc, bac, \cdots \}
\]

which can be specified by \( \langle \{ a, b, c \}, P1 \land P2 \rangle \), where \( P1(0 \leq l(t[a]) - l(t[c]) \leq 1) \) and \( P2(0 \leq l(t[b]) - l(t[c]) \leq 1) \), where the notation \( l(t[a]) \) is used for a length of a trace \( t[\{a\}] \) i.e. the number of occurrences of \( a \) in the trace \( t \).

It is clear that the above process can be obtained using CWR by weaving two processes with specifications of the form \( \langle \{ a, c \}, P1 \rangle \) and \( \langle \{ b, c \}, P2 \rangle \).

In terms of LPNs the weaving corresponds to identifying those symbols, and hence, their transitions, which are in common of two processes composed. So we could obtain our composite implementation shown in Fig. 2(a) by weaving two LPNs given in Fig. 2(b) and (c), assuming that the transition \( c \) is identical to both LPNs.

4.2 Discriminators

Looking back to our mechanism \( D \) with nodes one-to-one labelled by symbols we suggest that each symbol denotes a particular event on a corresponding node. In order to perform structural synthesis at this abstraction level we would like to be able to interconnect some of the nodes of two composed mechanisms, thereby identifying corresponding symbols and events.

For the sake of formal convenience, we introduce the concept of a structural object, called discriminator (in opposition to that of the behavioural type which is a process).

Let \( D \) realise a process \( \langle A_D, X_D \rangle \). We restrict ourselves to a subclass of predicates on traces in such a manner that the predicates are defined on the values of parameters \( l(t[B]) \), where \( B \) is a subset of \( A_D \), i.e. they specify an allowed relationship between the numbers of occurrences of node symbols. Therefore, we consider a class of mechanisms
that ‘discriminate’ between their nodes during their operation according to some prescribed strategy.

D is called a discriminator of P-type if for each \( x \in X_D \) the predicate \( P \) is true and it is true only for traces in \( X_D \).

Therefore, formally defining a discriminator \( D \) of P-type, we specify a pair \( < A_D, P > \) (called the discriminator specification) such that \( X_D = \{ x \mid x \in A_D^* \land P(x) \} \) (recall that \( X_D \) is prefix-closed).

The above example of a process, whose LPN is shown in Fig. 2(b), gives the behavioural representation for the discriminator called a buffer with capacity 1.

We denote such a discriminator by \( BUF_1(a, b) \) implying that it is a two-node mechanism whose specification has the form

\[
< \{a, b\}, 0 \leq l(t[a]) - l(t[b]) \leq 1 >
\]

In addition to the qualitative view of a discriminator provided by its characteristic predicate we can also suggest some quantitative measures. These are an excess and a shift. The excess of a pair of node symbols \( \varepsilon(a, b), a, b \in A_D \) for the trace \( t, t \in X_D \), is defined by the expression \( \varepsilon(t[a], t[b]) = l(t[a]) - l(t[b]) \). The shift of a pair of node symbols \( (a, b), a, b \in A_D \) for the trace \( t, t \in X_D \), denoted by \( S(t[a], t[b]) \) is the maximum number of symbols \( a \) which occur between two adjacent symbols \( b \) in the trace \( t \).

Constructing predicates on \( l(t[B]) \) for subsets \( B \) in \( A_D \) we can obtain various types of discriminator. The following table contains some of these types with their characteristic predicates.

<table>
<thead>
<tr>
<th>name</th>
<th>symbol</th>
<th>characteristic predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer of capacity k</td>
<td>( BUF_k(a, b) )</td>
<td>( P_1 : 0 \leq l(t[a]) - l(t[b]) \leq k )</td>
</tr>
<tr>
<td>multi-channel buffer of capacity k</td>
<td>( MBUF_k(A, B) )</td>
<td>( P_2 : 0 \leq l(t[A]) - l(t[B]) \leq k )</td>
</tr>
<tr>
<td>selector 1 to k</td>
<td>( SEL_k(a, B) )</td>
<td>( P_3 : 0 \leq l(t[a]) - l(t[B]) \leq 1 )</td>
</tr>
<tr>
<td>multiplexer k to 1</td>
<td>( MLX_k(A, b) )</td>
<td>( P_4 : 0 \leq l(t[A]) - l(t[b]) \leq 1 )</td>
</tr>
<tr>
<td>ordered selector 1 to k</td>
<td>( OSEL_k(a, B) )</td>
<td>( P_5 : P_3 \land (\forall i, j : 1 \leq i \leq k \land 1 \leq j \leq k \land i &lt; j) )</td>
</tr>
<tr>
<td>ordered multiplexer k to 1</td>
<td>( OMLX_k(A, b) )</td>
<td>( P_6 : P_4 \land (\forall i, j : 1 \leq i \leq k \land 1 \leq j \leq k \land i &lt; j) )</td>
</tr>
</tbody>
</table>

It should be pointed out that the above types of discriminator have the excesses of pairs of their symbols such that they either are bounded for all \( t \in X_D \) as, for example, in \( BUF_k(a, b) : t \leq \varepsilon(a, b, t) \leq k \) or relate to the length of \( t \) in linear order as, for example, in \( OSEL_k(a, B) : \forall t : \varepsilon(a, b, t) = 0(k, l(t)) \). The shifts of all these types are bounded. Such a class of types can be named linear discriminators.

Henceforth, we will use only those types of discriminator that are shown in the above table. Generally, we might have suggested other types of a wider class, for instance, such that have a characteristic predicate based on some function \( F \):
\[ F(l(t[a])) \leq l(t[b]) \leq F(l(t[a]) + 1). \]

4.3 Abstract design of FIFO buffers

In this section we demonstrate how the above formal concept can be used in designing a basic unit of the token ring LAN adapter, such as a FIFO buffer.

Suppose we need to design a module which is capable of receiving up to k items of data at the input port and storing them until they are read in the same order from the output port.

4.3.1 Top level specification

We formalize our notion of such a module by introducing a structural model for it. The discriminator \( BU_F_k(a, b) \) models the FIFO if the node \( a \) has the following meaning: ‘an item of data enters from the input port’, and the node \( b \) means that ‘an item of data is retrieved from FIFO through the output port’.

4.3.2 Decomposing the buffer specification

The other important issue of the FIFO definition is that the order of items at the input must be the same as the order of items at the output port. The semantics of such an ordered buffer helps us to arrive at two fundamental ideas of how to decompose our buffer.

The first idea is to arrange a pipeline of buffers of a lower capacity connecting them in series. It implies that each subbuffer has its own storage, and therefore every data item entering the whole buffer must travel across all subbuffers in series before it leaves the module.

The other approach, called parallel composition, uses a collection of \( k \) buffers of capacity 1 which together correspond to a multi-channel buffer of capacity \( k \) and 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 ordered multiplexer \( k \) to 1.

Fig. 3 shows the corresponding structures of the buffer in the case of series (pipeline) composition (a) and parallel composition (b).

Using CWR it can be formally shown that the first structure with the given identification of nodes is, in fact, a discriminator \( BU_F_k(a, b) \) and the second structure, with its own interconnection of identical nodes of the component discriminators, constitutes a buffer \( BU_F_{k+2}(a, b) \). We hint that such a proof manipulates characteristic predicates using an algebra of inequalities.

4.3.3 Abstract behavioural implementation

The behavioural implementation of the buffer can be obtained using the LPN notation. Our design strategy demands that such an implementation should be constructed only after the structural decomposition, in order to bridle the complexity. That is why we suggest that LPNs have to be constructed for separate component discriminators. Two examples of LPN implementations are shown in Fig. 4 for \( BU_F_1(c_1, d_1) \) and \( OSEL_k(a, C) \). The composite LPN can be obtained in a straightforward manner by making identical such transitions whose corresponding nodes are structurally connected in Fig. 3.
5 Logic design

In order to obtain a circuit for each component discriminator we need to choose the technique for converting an abstract implementation of a component, say, by LPN into a system of Boolean functions which define our final object.

In this section, some ways of deriving self-timed circuits for discriminators from their initial behavioural descriptions are outlined. These ways require three major steps.

The first and most crucial step is to construct a signalling expansion (for example, by means of signal graphs [7]) of the abstract behavioural implementation of a discriminator defined on an alphabet of node symbols.

The second step is concerned with analysing the signalling expansion with respect to correctness and completeness, and making necessary corrections and modifications while preserving the prescribed order semantics for the initial signal interpretation, thereby providing the final model of the discriminator circuit operation.

The third step consists of choosing the most effective technique for a self-timed circuit realisation among the existing ones [2] and converting the signalling description of operation into the self-timed circuit of the module.

5.1 Signal interpretation of discriminators

The specific aspects of our design objects, discriminators, are as follows. The node symbols correspond to particular events that occur at the attached data paths. These events are themselves decomposable into groups of more elementary (or atomic) events. The latter are related to changes of some signal values at the interconnection 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 the signalling interpretation of events may raise the amount of parallelism in the model at the elementary level events.

We introduce the notion of signalling expansion as follows. Let a discriminator D be specified by the pair \(< A, P > \) 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 \(T = < A, X >\), where \(X = \{ t : t \in A \land P(t) \}\). Let each symbol in \(A\) be assigned to a subset of signals which are binary variables from a finite set \(SA\). For \(SA\) a finite set of allowed changes of signals, denoted by \(SC\), is defined as \(\cup_{sa \in SA} \{ +sa, -sa \}\), where \(+sa(=sa)\) denotes the transition of \(sa\) from 0 to 1 (from 1 to 0). The function of signalling expansion \(\delta\) is defined by \(\delta : A \rightarrow 2^{SC}\).

The process \(ST = < SC, SX >\), where \(SX \subseteq SC^*\) is a prefix-closed set of allowed signal changes, called a signalling expansion of the process \(T\).

The signalling expansion of process \(T\) to \(ST\) must satisfy certain rules in order to comply with principles of the self-timed approach. These rules are:

1. Preserve the global order semantics. If in an abstract process \(T\) events \(a\) and \(b\) are ordered, then corresponding signal changes of the form \(+sa\) and \(+sb\) must be ordered in \(ST\) in the same order as in \(T\).

2. Preserve the local order semantics. If an abstract event \(a\) is meant to be expanded into a signalling sequence defined on corresponding signals \(+sa\) and \(-sa\), say, in the form \(+sa_1, -sa_1, \ldots, +sa_k, -sa_k\), where the upper index shows the serial number of the signal transition within possibly a multiple-transition operating cycle, then the given order has to be preserved in \(ST\).
3. Preserve 'request-acknowledge' matching. Every data port which corresponds to a particular node of discriminator, say, a has to be expanded into at least one pair of input-output signals sa (input) and oa (output) for which the local signalling order is defined such that for each pair of transitions \((+sa, -sa)\), where \(sa = ia\) (or \(sa = oa\)), there must be exactly one transition \(+sa'\) or \(-sa'\), where \(sa' = oa\) (or \(sa = ia\)).

4. Provide the completeness of signal specification. This rule reflects the need for a non-contradictory description of the circuit, i.e. for a sufficient number of logic variables in \(SA\). In order to meet this rule the analysis of the process \(ST\) has to be made using, for example, the signal graph model (see the next section), and the subsequent correction by inserting additional variables.

5.2 Analysing signalling expansion by signal graphs

In this section we introduce signal graphs, an attractive formal model for analysing signalling expansions of the behavioural description of discriminators. These can also be used as a separate specification tool for objects defined directly at the signal level, for example, for specifying bus signalling protocols and corresponding controller circuits.

It should be pointed out that signal graphs represent a more narrow class of processes than that that can be generally defined with LPNs. But the narrowness is concerned only with their inability to specify alternatives in processes. However when we need to define the highly concurrent behaviour of a mechanism, they provide the most compact form of description, and what is more important, the polynomially complex analysis procedures.

We define a signal graph as a subclass of signal Petri nets. It has been shown elsewhere [7] that a so called normal signal graph generates distributive state transition diagrams. Here we examine the normalcy at a greater detail and present the technique for checking them for an arbitrary signal graph with respect to practical logic design examples.

Some knowledge of subclasses of Petri nets, particularly, marked graphs is presumed here. Marked graphs (MG) generate distributive marking diagrams (MD). The latter are oriented graphs whose vertices are reachable markings and whose arcs are labelled with firing transitions. The term 'distributivity' is related to the corresponding lattices which can be defined on a set of vectors of transition firing numbers with respect to a given initial marking.

In order to define a signal graph, a set of binary variables (signals) \(S = \{s_1, s_2, \cdots, s_n\}\) is introduced. We denote transitions of signal \(si\) from 0 to 1 by \(+si\) and from 1 to 0 by \(-si\), and either of these by \(d_{si}\) where \(d \in \{+, -\}\).

A signal graph (SG) is a triple \(<MG, SC, \tau>\) where \(MG\) is a live and safe marked graph, defined by \(MG = <V, E, M>\) where \(V\) is a finite set of vertices, \(E \subseteq V \times V\) is a set of arcs, \(M : E \rightarrow B\) is an initial marking function with \(B = \{0, 1\}\); \(SC = \cup_{i\{d_{si}\}}\) is a set of allowed transitions of signal variables; \(\tau : V \rightarrow SC\) is a labelling function. Therefore, SG is an MG in which vertices are named with the signal transitions.

We call a labelling function conflict-free if for each reachable marking and variable \(si\) there is at most one excited vertex labelled with the transition \(d_{si}\).

An SG with the conflict-free labelling is called coherent.

We call a labelling function sign-balanced if for each sequence of signal transitions with respect to initial marking between any two transitions of the same type \(d_{si}\) there exists at least one transition of the opposite type \(d_{si}\).

An SG with the sign-balanced labelling is called consistent.

It is clear that the class of consistent SGs is contained within the class of coherent SGs. The consistency implies a necessary level of correctness of a specification given by SG that is expressed in following statement.
Statement 1. A consistent SG generates a state transition diagram. □

A state transition diagram (STD) is an oriented graph whose vertices are labelled with full states of a specified circuit, i.e. they are binary n-tuples of values of si, and arcs are labelled with corresponding transitions dsi. The values that can change between a given state and another one connected to each other by an arc are marked with the asterisk (*) token. A variable whose value in n-tuple is marked with * is called excited in a given state. In this paper we omit the description of algorithms for converting a consistent SG to STD and vice versa. We just hint that such a conversion may use the ordinary procedure of building the MD by depth-first search where each marking has corresponding states in the STD.

A consistent SG, however, may generate STD with multiple states, i.e. states labelled with equal n-tuples of signal values. Such an STD is called contradictory. Informally, the contradiction of this kind means that the system is yet underspecified and some components are still hidden from the designer's knowledge. For example, when an SG specifies an interface protocol these components may be interpreted as an internal memory of the controller circuit which is absolutely necessary for the realization of the latter.

We incorporate a higher level of correctness into the hierarchy of SG classes through the notion of a normal SG which guarantees the completeness of the specification.

An SG is called normal if it is consistent and for each allowed sequence of markings has no proper subset of variables S' ⊂ S which can proceed through the full cycle of their values while other variables (from S \ S') stay unchanged. An STD of a normal SG is non-contradictory and distributive.

It is suitable to check consistency and normality using the relations of precedence and concurrency on the set of signal transitions SC. The formalization of these relations requires the introduction of a concept of a history, or so-called unfolding, which is an infinite and acyclic object generated by SG. Each occurrence of a transition in SG generates a unique vertex in unfolding with the same label, augmented with a unique number of the occurrence. A set of vertices having such an index i in the unfolding with corresponding transition arcs is called rth period of unfolding.

Pair of occurrences, viz. ith and jth, of transitions dsi and dsk belong to a precedence relation, denoted by dsi i → dsk j, if there is a path in unfolding between dsi i and dsk j. If not (dsi i → dsk j) then dsi i and dsk j are in concurrency relation, which is denoted by dsi i || dsk j. When considering the SG transitions dsi and dsk the concurrency relation is defined between them if there exists any pair of their occurrences (dsi i, dsk j) belonging to the concurrency relation on the occurrence set of the unfolding. Similarly, two transitions in SG dsi and dsk in the precedence relation i f for any i dsi i → dsk j → dsi i+1. It can be shown that the unfolding can be floored to its first two periods, and corresponding relations can be computed on this finite object with polynomial complexity with respect to the number of vertices in the SG.

In terms of the above relations the fact that the labelling is conflict-free means that for each variable si all its transitions dsi, dsi are ordered according to the precedence relation. Hence, each transition dsi can be assigned with an index w such that dsi w → dsi w+1. Then we can determine whether the labelling is sign-balanced by simply checking the condition for each si:

\[ dsi_w \rightarrow \overline{dsi}_w \rightarrow dsi_{w+1} \]

For the analysis of normality we restrict ourselves to ensuring that all variables si_w are distinct (each transition dsi_w labels only one vertex in SG). This restriction does not seem crucial and we use it only for the sake of clarity.

Let us have a consistent SG. Then the variables si and sj are called:
directly strongly connected \((CON)\) iff there exists the following precedence: \(d_{si} \rightarrow d_{sj} \rightarrow d_{si} \rightarrow d_{sj}\), with respect to given initial marking;

strongly connected iff \((si, sj) \in CON^*\), where \(CON^*\) is the transitive and symmetric closure of \(CON\);

weakly connected of rank \(r, r \geq 0\) \((CONW^r)\), iff there exists a precedence of the form: \(d_{si} \rightarrow d_{sj} \rightarrow d_{si} \rightarrow d_{sk}\), with respect to initial marking, and \((sj, sk) \in LINK^r\), where

\[
LINK^r = \begin{cases} 
CON^*, \text{ if } r = 0 \\
(CONW^{r-1} \cup LINK^{r-1})^*, \text{ if } r > 0
\end{cases}
\]

connected if \((si, sj) \in LINK^{r_{max}+1}\) (the maximal rank of weak connectedness is determined from the condition \(CONW^{r_{max}+1} = \emptyset\)).

For brevity we denote the connectedness relation with \(L\). This relation partitions the set \(S\) onto the disjoint classes of connectedness.

**Statement 2.** A consistent SG is normal if all its variables belong to one connectedness class. \(\square\)

Since each iteration step of constructing the relation \(LINK^r\) using \(LINK^{r-1}\) has the complexity of \(O(n^5)\), where \(n\) is the number of vertices in SG, and \(rmax\) has complexity \(O(n)\), the complexity of the normality check is of \(O(n^4)\).

**Example.** Consider the SG shown in Fig. 5. The notation \(+i, -i\) is more convenient than that of \(+si\) and \(-si\). Variable 2 switches twice \((+2.1, -2.1\) and \(+2.2, -2.2)\). The STD generated by the SG is shown in Fig. 6. It contains multiple states, for example, \(000000\) and \(000000\), and, hence, is not distributive (contradictory). Analysing the SG we establish that the SG is live, safe and consistent (for each variable there exists a synchrocyle, i.e. a simple circuit with single token, which defines the linear order for all the changes of a given variable, and these changes are sign-balanced).

The triangular matrix below contains the results of computing the connectedness relation. The matrix entries, digits, correspond to following relations:

\(CON - 1, LINK^0 = CON^* - 1 \cup 2, CON^0 - 3, LINK^1 - 1 \cup 2 \cup 3 \cup 4;\) the maximal rank of weak connectedness is 0 because \(CONW^1 = \emptyset\).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2.1</th>
<th>2.2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2.1</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The connectedness is determined by \(L = LINK^1\). The matrix entries marked with 0s are not in the relation \(L\). Generally, for the weak connectedness of arbitrary rank \(r\) entries in \(LINK^r \setminus (CONW^{r-1} \cup LINK^{r-1})\) and \(CONW^r\) are denoted by numbers \(2r+1\) and \(2r+3\), respectively.

Therefore, the given SG is not normal: \(L\) splits the set \(S\) into two non-empty classes: \(K^1 = \{1, 2.1, 2.2, 3, 4\}\) and \(K^2 = \{5\}\). In order to normalize the SG we have to connect variable 5 with at least one of the variables from \(K^1\). It should be instructive that the normalization must not change the previous ordering between the signal transitions. Then
such a coupling is performed through the insertion of additional variable(s) into the SG specification. We can, for example, insert +6 between +5 and −5, and −6 between +3 and +2.2. This transformation results in several new entries in connectedness, namely (6,5), (6,3), (6,4) ∈ CON. Hence, both 5 and 6 enter the common class K = K1 ∪ K2 ∪ {6}. We also note that the above transformation has not changed the initial ordering of signal changes.

6 Two examples of self-timed logic design

In the above section we have shown how the normalcy of an SG can be established. The normalcy is a sufficient condition for the existence of a distributive STD and, hence, of a delay-insensitive circuit [3,14]. The circuit can be derived from the normal SG by means of obtaining the Boolean functions (BFs) for variables $s_i$ using a truth table (TT) which can be built from the STD corresponding to the SG. However, the chain SG - STD - TT - BFs is involved with exponentially complex steps. Therefore, we look for an alternative technique for direct conversion of SG to BFs. Such bridling of design complexity is concerned, first of all, with laying some restrictions upon the complexity of the connectedness hierarchy.

In this paper we do by no means solve the problem of obtaining the general mechanism of deriving functions directly from a SG. We rather illustrate by two characteristic examples our design approach, applying it to both a FIFO elementary subunit $BUF_1(a,b)$ and a master controller for bus data read operations.

6.1 Designing logic for a buffer of capacity 1

We demonstrate how the three steps of logic design described in the preamble to section 5 are followed for a simple discriminator $BUF_1(a,b)$ whose initial abstract synthesis has been presented in section 4.

We presume here that events $a$ and $b$ have the following semantics: $a$ is 'a data value is received from the input port', and $b$ is 'a data value is written into the register and thereby applied to the output port'. We partition the structure of such a buffer cell into the pair of subcells: a data cell (DC) and a control cell (CC) as shown in Fig. 7(a). The self-timed signalling convention at both the input and output ports is defined by corresponding signal graphs shown in Fig. 7(b) and (c), respectively.

It should be noted that the implementation of DC provides an indication of the defined values at the two-rail input $DI = \{di,di'\}$ and at the two-rail output $DO = \{do,do'\}$. According to the two-rail coding discipline [1,2], for 'one' and 'zero' data values the combinations 01 and 10 are used, respectively. The all zero spacer (00) is used for representing the 'data undefined' value.

Fig. 8 shows the signalling expansion defined by $S(a) = \{di,di',ao\}$ and $S(b) = \{do,do',bi\}$ for which the SG contains the labels $+DI(+DO)$ and $-DI(-DO)$ denoting the transitions on the two-rail port from the spacer to the defined state and from the defined state to the spacer, respectively. The signalling expansion satisfies the first three rules of section 5.1. The fourth rule is not satisfied because this SG is not normal: variables constitute two separate connectedness classes $\{DI,ao\}$ and $\{DO,bi\}$.

It makes sense to design DC and CC separately. Corresponding signal graphs for DC and CC are shown in Fig. 9(a) and (b), respectively. We notice that the SG in Fig. 9(a) gives the definition of the operation of a latch element (D-flop with cutting off the inputs) with the data outputs $do, do'$ and the data inputs $dt, dt'$, the hold signal input $bo$ and the pair of indication outputs $ao$ and $bi1$. $ao$ indicates the defined and the spacer at the inputs $DI$, whereas $bi1$ does the same function for $DO$.  

12
Consider in more detail the synthesis of CC logic. The signalling expansion of Fig. 9(b) in which \( S(a) = \{ ai, ao \} \) and \( S(b) = \{ bo, bi, b1 \} \) does also satisfy the first three rules of section 5.1. However the set \( SA = S(a) \cup S(b) \) is partitioned in two separate connectedness classes which must be linked by some additional variables in order to derive a logic circuit. It can be shown that the addition of only one extra variable to the specification while preserving the established order on the set \( SA \) will not suffice for making all variables connected. After adding two variables \( d1 \) and \( d2 \) such that \((d1, bo) \in CON, (d2, ai) \in CON \) and \((d2, d1), (d2, bo) \in CONW^0\), we obtain the resulting SG shown in Fig. 10 which is normal. From this SG we derive Boolean functions for those variables that are realized in CC: \( ao, bo, d1, d2 \). These functions defining the delay-insensitive circuit are expressed in the following form:

\[
\begin{align*}
d1 &= \overline{d2} \, ai \, b1 \, \overline{b1} + (\overline{b1} + b1) \, d1 \\
d2 &= d1 \, \overline{b1} \, \overline{b1} + ai \, \overline{d2} \\
ao &= d1 \, d2 + d2 \, b1 \\
bo &= d1 \, d2 + d1 \, b1
\end{align*}
\]

The first two functions define self-dependent variables \( d1 \) and \( d2 \) in the form \( x = Sx + Rx \, x \), where \( Sx \) is an excitation function for the transition of \( x \) from 0 to 1, and \( Rx \) is one for the transition of \( x \) from 1 to 0. Both \( Sx \) and \( Rx \) are independent of \( x \), and they must satisfy the invariant \( Sx \cap Rx = 0 \). The other two functions for \( ao \) and \( bo \) are not self-dependent, that is, if they are defined in the above canonical form then their \( Rx \) is identical to 1. For them the orthogonality invariant is no longer held, because the feedback is always inhibited.

The above system can be realized using a collection of AND-OR-NOT gates and inverters. Fig. 11 shows such a realization where the implementation of a box element is given in Fig. 11(b). This circuit behaves in accordance with its specification defined in Fig. 9(b) with respect to the initial variable set. It is delay-insensitive with respect to gate and wire delays excluding the delays of feedback wires. The latter are presumed negligible since the box elements are apparently accommodated within equichronic regions.

6.2 Designing logic for a bus data read master controller

In this section we consider how the ‘Somebus’ controller logic circuit can be obtained through using the formal approach of applying signal graphs as a specification tool for backplane bus protocol.

The design objects like interface controllers for such backplane protocols as Unibus, Q-bus, Futurebus etc (with asynchronous data transfer) show that their operation discipline defines not only causal relationships between binary signals but also ordering of events concerning with changing the values on parallel bundles of wires which are subject of the skew phenomenon. The most widely used techniques for compensating these skews exploit a somewhat quasi-asynchronous approach: to insert a scaled built-in delay. There are, however, other possible methods using, for example, the self-synchronizing code \([3,14]\). Here we apply signal graphs for specifying signalling disciplines of the first, more standard, approach.

We supplement the model of SG given in section 5.2 with some other types of vertices which model the effect of inserting a scaled delay of a certain value. Vertices labelled with \( T(K) \) model the delay of \( K \) units of time. The meaning of such labellings as \( T(\geq K) \), \( T(\leq K) \) and \( T(K1 \leq, \leq K2) \) is obvious without any comments. Such a modification of SG is called a timed SG (TSG) \([7]\).
It is quite common that the backplane protocols are defined with timing diagrams. Fig. 12 shows an example of such a definition for the data read operation protocol of Unibus. It uses signals \( AC \) (modelling state of address and control line bundles), \( D \) (state of data line bundle), \( MSYN \) and \( SSYN \) (states of master and slave synchronization lines respectively). The approach proposed here exploits TSG and significantly simplifies the description technique as well as the protocol understanding. The corresponding TSG is shown in Fig. 13. For the sake of clarity, we distinguish the vertices denoting the master's actions (boxes) and the slave's actions (ovals). The time delays of 150ns and 75ns are introduced by vertices \( T(\geq 150) \) and \( T(\geq 75) \) due to the Unibus requirement concerned with skew compensations.

The procedure for translating the given interface protocol into a self-timed master logical circuit is performed as follows. First of all, a structurization of the controller must be done. The circuit has two structural links: one is with the bus medium and the other is with a higher interaction level as shown in Fig. 14. We assume that the internal structure incorporates two operational units, registers, one of which, the register \( Rg1 \), is used for latching \( AC \) state value, and the other, the register \( Rg2 \), is used for latching the received data \( D \). The control part produces the controls for both registers, receives the completion indication signals from them, produces control signals \( MSYN \) and \( ACK \) and receives control signals \( SSYN \) and \( READ \). The signal discipline on the pair of \( READ \) and \( ACK \) signals is usual handshake.

Then we undertake the following transformation of the TSG to SG defining the operational behaviour of the control part. In doing so we first exclude all timed vertices, bearing in mind that the required timing order of vertices incident to them will further be realised by special circuitry. Then we exclude vertices labelled with \( +D \) and \( -D \) because they do not carry any valuable information necessary for the implementation of the master controller. After that, having found that the obtained SG is not normal (an arc directly leading from \( -AC \) to \( +AC \) shows that \( AC \) forms a separate connectedness class) we insert an additional variable, say \( d \), for which a pair of transitions \( +d \) and \( -d \) finds its place in the SG with taking into account the useful semantics of such a variable as a signal for strobing data from the \( D \) bus into the latch \( Rg2 \). Particularly, it is clear that the strobe \( +d \) is possible only after the delay of 75ns after the receipt of \( +SSYN \).

As a result of the above transformations we obtain an SG which is normal. By the technique, similar to that described in section 6.1, we derive the system of BFs in the form

\[
\begin{align*}
AC &= d + MSYN AC \\
MSYN &= d \ AC SSYN + (d + SSYN) MSYN \\
d &= SSYN MSYN + AC d
\end{align*}
\]

There is an important detail in this system. It seems that for \( MSYN \) in \( R_{MSYN} \) only the immediate predecessors variables could have been included, viz. \( d \), which is also used in the function \( S_{MSYN} \) providing by that the required orthogonality \( S_{MSYN} R_{MSYN} = 0 \). But in order to make the whole circuit delay-independent with respect to all gates we must also bear in mind that the variable \( SSYN \) enters the system in both direct and negated forms which implies using the inverter with a finite delay. Hence, we must provide the indication of the output of this inverter at the input of the gate realising \( MSYN \) in both switching changes. Namely, for indicating the change of this signal from 1 to 0 we insert \( SSYN \) into \( R_{MSYN} \). It is also possible, since \((MSYN, SSYN) \in CON\). Such a technique can be effectively applied in other cases when negated variables are used in BFs and the
ordering logic of signal changes is not violated by such a strengthening of $Sx$ or $Rx$ for some $x$.

The above system provides a selection for the actual control circuit. We need some additions to be made to take into account the previous exclusions from the initial TSG.

First, we must provide the interaction on the pair $READ$ and $ACK$. The SG shown in Fig. 15 allows to use the completion signal of $Rg2$ as the signal $ACK$. This signal originates as $d$, which passes through the $Rg2$ playing the role of a simple delay for $d$.

Second, the circuit must have built-in delays of 150ns and 75ns that can be realised by standard RC networks.

Third, the bus must be driven by corresponding bus drivers, and the signals from the bus are to be applied first to Schmidt-flop elements. However, as normally done in multiplexing, the bus driving signals must be merged from all separate function subcontrollers (write and interrupt operations) into single signals before they can be applied to the bus drivers.

The final implementation of the control part is matched with $Rg1$ and $Rg2$ using the 'request-acknowledge' handshake. This imposes some requirements on the realization of these latches, which must provide the completion signals for both operational phases: writing phase and idle phase. The examples of such registers are given at length in [14]. It is easy to be convinced that the circuit is insensitive to gate and wire delays and, hence, possesses all positive attributes of the self-timed implementations - the operation speed is determined by real element delays, rather than by the clock interval, and the self-checking margin because the circuit halts in case of a stuck-at fault [14].

7 Conclusion

The main distinctive issue of the above methodology comparing it with those given elsewhere [8, 15] is that it provides the unified and rigorous solution to the problem of synthesizing logical implementations for interface hardware, especially for the control circuits which are insensitive to delays of elements and wires.

The methodology covers both an abstract stage of the design and a logic of signal synthesis phase. Of course, there are still certain limitations relating to classes of objects to be designed (the abstract synthesis is performed only for the discriminator-like modules) as well as processes involved (concurrency without alternatives), however that does not seem a principal obstacle to make the necessary extensions in order to accommodate more powerful objects and processes which is the subject for further research. It should however be pointed out that the increase of description power of both structure and model classes must always be traded-off versus the inevitable complexity growth for executing the analysis procedures. Therefore, the technique of checking the signalling expansion's correctness and completeness by means of effective concepts of normalcy and connectedness facilitates some rather simple ways to the correction of the specification while preserving the initial order semantics despite obvious limitations of the signal graph formalism.

Further study may be attempted, for example, in establishing some constructive restrictions on classes of connectedness for finding how they affect the Boolean functions derivation rules.

Acknowledgements

The author would like to thank Mr. Albert Koelmans and his colleagues at the Computing Laboratory for their support and attention.
References


Fig. 1. A typical token ring LAN adaptor

Fig. 2. An example of a labelled Petri net implementation of a process
Fig. 3. Decomposition of buffer
(a) series   (b) parallel
Fig. 4. Labelled Petri net for BUF \((c_i, d_i)\) and OSEL \((a, c)\)

\[ 1 \quad i \quad i \quad k \]

Fig. 5. An example of a signal graph
Fig. 6. State-transition diagram generated by signal graph
Fig. 7. (a) the structure of a buffer cell  
(b) input signalling conventions  
(c) output signalling conventions
Fig. 8. Signalling expansion of BUF₁ behaviour

Fig. 9. Signal graphs
(a) data cell
(b) control cell
Fig. 10. Normalised signal graph for control cell

Fig. 11. Realisation of control cell. (a) structured representation (b) example for box 1
Fig. 12 Timing diagram of data read operation

Fig. 13. Timed signal graph of data read protocol
Fig. 14. Structure of the data read controller

Fig. 15. Signal graph defining the operation of the master controller circuit