Revised version (January 1991) of the paper presented at Int Conference on application and Theory of Petri Nets, Paris, June 1990. pp 244-264

PETRI NETS AND ASYNCHRONOUS BUS CONTROLLER DESIGN

Alexandre Yakovlev[1] and Alexei Petrov[2]

Computing Science Department
Leningrad Electrical Engineering Institute
Leningrad 197022 USSR

Summary

Petri nets, a convenient formal tool for modelling the dynamic behaviour of asynchronous structures, have been used for both designing digital hardware [1,2,3] and specifying/verifying parallel bus interface protocols [4]. Since the latter are supposed to be implemented in digital hardware controllers, it seems topical to establish a strict semantical relationship between these two applications of Petri nets, thus aiming at a unified theoretical framework that will enable a mechanical translation of Petri net protocol specifications into logical circuits for protocol *controllers*. A characteristic feature of bus interface controllers is that they represent a class of hardware modules whose behaviour is essentially *reactive*, as opposed to another class of hardware which can be called *transformational*. In this dichotomy we follow the ideas outlined in [5]. Therefore, in the design of such objects as bus controllers, the use of Petri nets as a specification language for reactive circuit behaviour is laudable, provided that the design process effectively separates the reactive parts of the controller from its purely transformational parts, whose paradigmatics is much more adequate to the techniques used in combinatorial function modelling rather than to event-oriented notations.

This paper tackles the problem of filling the above-mentioned semantical gap by:
- studying the underlying semantics of Petri net specifications of reactive hardware behaviour and parallel bus protocols;
- analysing the control flow semantics of Petri net specifications defined in terms of the so-called signal graphs, which are a signal-labelled version of marked graphs (a subclass of Petri nets that is capable to represent partially ordered actions of the underlying asynchronous structure);
- demonstrating the example of deriving a *self-timed* implementation [6] of controller circuits from a Petri net specification of the VME-bus data transfer protocol.

The major theoretical results of the paper are aimed to be a methodology for analysis of signal graph specifications that could then be, in a rather straightforward way, tailored to an automated design process.

[1] In 1990/91 with: Department of Computer Studies, Polytechnic of Wales, Pontypridd, Mid Glamorgan CF37 1DL, U.K.

[2] In 1991 with: Swedish Institute of Technology, 8 Apollogatan, Helsinki 00100, Finland

1 Petri Nets and Parallel Bus Protocol Specification

1.1 Transactions, Cycles, Actions

In order to show how Petri nets can be put into work for the modelling of the lower layers of an asynchronous bus protocol, we follow the notation and meaning of the layered model of bus protocols used in [4]. This model suggests that each layer produces a sequence of requests to the lower layer, thus being the user of the latter. The layers which are typical for many asynchronous busses (a synchronous bus, where actions are ordered with respect to some clocking mechanism, can in fact be treated as a special case of asynchronous bus with the clock signals just being viewed as independently generated signals) are: *transaction*, *cycle* and *action* layers.

The transaction layer usually involves cycles of the following types: *arbitration*, *addressing*, *data transfer* and *interrupt handling*. Each of these types uses its own set of bus lines, thus forming its own *domain* of discrete variables, or signals, that are subject to changes, in their values, in order to represent particular messages between the interacting modules on the bus. For example, the data transfer cycle may utilize such lines as Data lines, Command line(s), Synchronization lines and Data Transfer Error line(s). In addition to the bus signal domain, each type of cycle implies the existence of its own operational structure domain which supports the implementation of the protocol cycle inside the bus controller. For the above data transfer example, the inside operational part includes such units as data transceivers, error and command decoders, internal registers for storing the various components of the data path.

At the cycle layer, a particular cycle is carried out. A data transfer cycle typically involves transferring a single data item between a pair of modules. The module which initiates the cycle is often called the *master* and the module executing the master's command is the *slave*. We emphasize here on the fact that our interest is focused on *asynchronous* bus protocols, for which the cycle layer uses the request-acknowledgement pairs of synchronization signals, and no explicit timing or clock signals are present.

Each cycle consists of a sequence of actions which are performed on the variables belonging to the cycle domain. In asynchronous busses such actions are temporally ordered by causal relationships. Practically, an action is a transition of a finite-state signal associated with a single bus line, a set of functionally grouped bus lines or some auxiliary internal status variables.

1.2 From Timing Diagrams to Petri Nets

The most common way of describing protocols at the cycle layer is using *timing diagrams* where the falling and rising edges of waveforms stand for the corresponding protocol actions performed on the cycle domain variables. These diagrams are often a starting point for the process of digital interface circuit design. Unfortunately, such diagrams cannot be efficiently used in specifying hardware modules which exhibit concurrent behaviour. Therefore we suggest that the protocol actions, as well as the internal actions of the bus controller, such as strobing a data item from the data lines into a register, sending a request to the target device etc., need to be associated with transitions of a Petri net built for a given protocol cycle. Topologically, for the data transfer cycles, this net consists of the three parts: master subnet, slave subnet and bus subnet (see Section 6.1).

There are two main reasons why Petri nets can be successfully used in specifying both protocols and the corresponding logic controller behaviours. First, they provide a clear formal way of protocol verification due to their natural purpose of defining concurrent and causally ordered activities. Second, they enable the designer to use some formal techniques for circuit implementation of protocols.

The problems related to the verification of bus protocols by extended Petri nets have been discussed in [4]. Thus, here, our major attention is focused on the problem of designing a self-timed logical implementation for the bus controller from a given Petri net specification.

The initial specification of the bus controller can be obtained from the protocol specification in the following way (for practical "visualization" the reader may refer to Sections 6.1 and 6.2, skipping, at this point, all intermediate material). First, we need to single out a topological part of the whole protocol which is associated with the behaviour of the module we are designing a controller for. Second, we define a top level structural description of the controller, which includes specification of the signal domain at the link with the target device, the opposite side of the bus interface, and at the link with the top-level-decomposed data-path units, like data transceivers and address/mode decoders, which build up the internal operational (and, hence, transformational) hardware of the module. Third, we redefine the behaviour of the controller with respect to the above structurization and its actions on the newly introduced variables. For example, if the bus controller is an adaptor between the bus and a local memory device, we combine the controller's actions of the bus cycle sequencing with its actions on the internal link with the memory unit, and, perhaps, with some additional buffering registers, so as to achieve that the projection of this combination onto the bus cycle variables is compatible with the original cycle specification (i.e., it preserves the original causal ordering).

An important issue, at the latter stage, is finding an efficient decomposition of the controller structure where the reactive part of the controller is explicitly singled out, so that the above combination of signalling specifications is performed on the signals from and to this reactive unit. Although some of these signals may be physically different from the top-level controller interface connections, they can represent such border connections internally, by being just a result of their combinatorial transformations. These transformations of course require introducing some circuit units in the controller's structure. From the "reactive viewpoint", such units can be modelled just as additional delays that are treated either in a strict *delay-insensitive* way (the units are provided with a double-rail information encoding facilities plus necessary completion indicators or they may consist only of simple logical gates operating in delay-insensitive external conditions [6]) or in the so-called pseudo-delay-insensitive way (considering each of them as a particular scaled delay element to compensate the skew of their output values with respect to the concurrent control flow(s) coming directly into the reactive unit).

Although the authors of [4] refer to [7] as a possible way to construct a circuit from an initial Petri net description, we claim that the direct use of the method given in [7] in the bus controller design application would yield the circuit solutions which are rather slow and area-inefficient. They introduce an extra level of control flow in the controller structure and, hence, produce an overly structured design, which may seriously suppress all positive margins of the asynchronous design style.

In order to discuss specific problems related to the implementation of Petri nets in logical circuits for bus controllers, we should analyze the semantics of Petri nets labelled with the actions associated with the changes of values of finite-state components. In the sequel, we shall use term "component", as a substitute for "variable", "signal" etc., which will imply both the interface bus signals, auxiliary internal signals and some finite-state objects of the internal operational part of the controller.

2 Signal Petri Nets and Their Semantics

2.1 Signal Petri Nets

We introduce the notion of signal Petri nets through the commonly used definition of an ordinary Petri net.

A (marked) *Petri net* (PN) is a quadruple $H = <P,T,F,M^o>$ where P and T are finite non-empty sets of places and transitions, respectively, $F \subset P \times T \cup T \times P$ is the flow relation between places and transitions, and $M^o:P \to N$(N is the set of non-negative integers) is the initial marking function.

We assume usual graphical representation of a PN and its firing rule definition. Also, wherever it may be of notational convenience, we use an alternative representation of a marking - as a multiset of places containing a corresponding number of tokens.

The behavioural semantics of a PN can be represented in the two major ways. The *interleaving* semantics is defined by the set of execution (firing) sequences obtained from the reachability graph, or *marking diagram*, built starting from M^0. The *partial order*, or *causal*, semantics can be defined on the set of occurrence nets, each of which corresponds to a single-run PN execution, with partially ordered events standing for the PN transition firings.

The interleaving semantics captures the sequential simulation of concurrent events, while the causal semantics captures concurrency in its natural form, as mutually independent actions, thus giving an explicit form of causality.

It has been shown elsewhere [8] that the partial order semantics is more powerful than the interleaving one for the general class of processes, and their descriptive power is equal only for the class of so-called distributive (also called stable, conservative or AND-confluent) processes [9].

A *signal Petri net* (SPN) is a PN whose transitions are labelled, through an appropriate *labelling function*, by the actions pertained to the changes of states of the discrete components of an asynchronous structure (generally speaking, some actions may however not change the state of a component, for example, the action of reading a value from the component).

Thus, let $X = \{x_1, x_2, ..., x_n\}$ be a set of such components. Each component x_i has a finite non-empty set of states that it may assume during operation, $S(x_i) = \{x_i.1, x_i.2, ..., x_i.k_i\}$. For example, in logical circuits, where the components are binary variables, we have $S(x_i) = \{0,1\}$. Let, for each x_i in X, be given a local operational semantics, the so-called *behavioural cliche* of the component defined, say, in the form of a transition graph on the set of vertices standing for elements $S(x_i)$. In such a graph, the set of arcs defines some finite set of allowed actions changing the component states. Denote the whole set of such actions as DX. For a logical circuit, $DX = \cup_i \{+x_i, -x_i\}$ where $+x_i$ denotes the transition of x_i from 0 to 1, and $-x_i$ the transition from 1 to 0. An alternative graphical representation can also be defined for the component behavioural cliches, describing them as SPNs with local behavioural semantics.

Thus, formally, an SPN is a triple $H^X = <H,DX,f>$ where H is a PN $<P,T,F,M^0>$, DX is the set of allowed actions on the discrete components in X, and $f{:}T\text{->}DX$ is the labelling function.

An example of an SPN is shown in Fig.1,a where the transitions *t1* through *t6* are labelled with the actions on binary variables *x1,x2,x3*.

Since an SPN is the interpretation of a PN, it has the same "unattributed" semantics, called *N-semantics*, as the underlying PN, i.e. it generates both a set of execution sequences and a set of partial orders. Furthermore, we may also study a signal-attributed version of the N-semantics, called *S-semantics*, where the events are assigned to the particular changes of the values of discrete structure components.

If we take the interleaving N-semantics of an SPN in terms of the marking reachability graph for the underlying PN, we can also speak about such corresponding S-semantics where the markings of the PN stand for the global states of components in X. Such semantics can be apparently represented as a directed graph, called the *state diagram*, whose vertices are labelled by the elements of Cartesian product $S(x_1)\times S(x_2)\times ...\times S(x_n)$, and arcs are labelled by the transitions of the form $dx_i \in DX$. Although our intuition suggests that an SPN marking is in direct correspondence with a state of the discrete structure

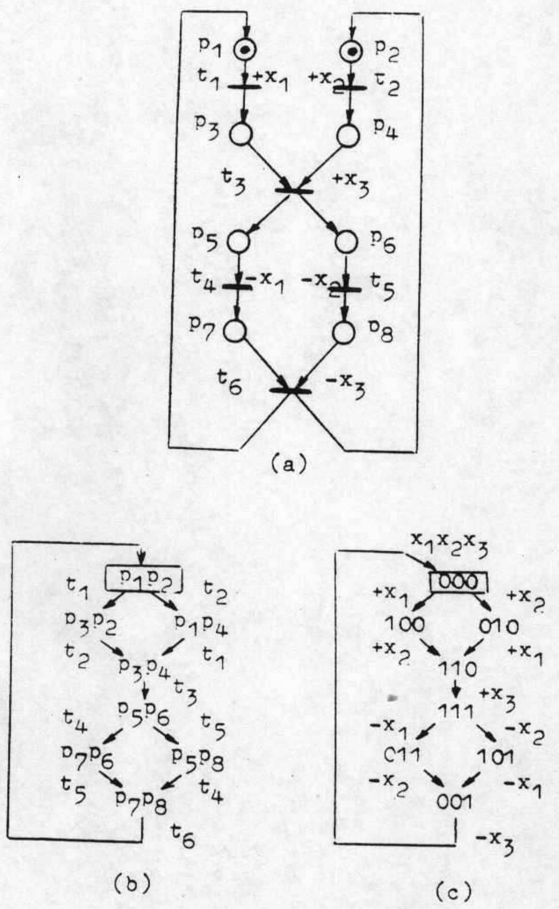


Figure 1. Example of a signal Petri net (a) and its marking (b) and state (c) daigrams

(see Fig.1,b and c), the isomorphism between N-semantics and S-semantics does not generally hold. Indeed, it is not the case that for any SPN, even safe and persistent, we can build a state diagram of the corresponding structure behaviour and, moreover, that such a state diagram, if built, would be complete and consistent for deriving logical functions of the structure implementation.

2.2 Semantic Compatibility

In order to establish consistency of the S-semantics for an SPN, we have to check compatibility between the global partial order defined by this SPN's S-semantics and the local behavioural cliches of constituent components. This should guarantee that even live and safe global specification is realizable for a given set of components. We generally distinguish two forms of compatibility, *weak* and *strong*.

The weak form is quite similar to a weak form of liveness, when a process defining the order between actions performed on the components is accomplishable, - the process goal state is said to be reachable if there exists at least one allowed execution sequence, in terms of the interleaving semantics, such that it is compatible with the allowed sequences of each component's cliche.

As an illustration of the weak compatibility we consider an example, inspired by [10], in which a component called the *Register* assumes certain local semantics of the *Read* and *Write* operations while the global SPN specification defines some schedule, or procedure, of register utilization. The register can store any value from a finite set $X(Reg) = \{x_1, x_2, \dots, x_n\}$. The allowed set of actions is $DX(Reg) = \bigcup_i \{w.i, r.i\}$ where $w.i$ and $r.i$ stand for the "write value x_i into the register" and "read value x_i from the register" actions.

We assume the register to have the following operational cliche defined in terms of the regular expression

$$(w.1;(r.1)^* \mid w.2;(r.2)^* \mid \dots \mid w.n;(r.n)^*)^*$$

where ";", "|" and "*" have their usual meaning of concatenation, selection and iteration operators, respectively. This cliche implies that the register value may be read as many times as needed, zero times inclusive, only if this value has been previously written into the register (but not overwritten by some new value). The SPN specification of the above cliche, for $n = 3$, is presented in Fig.2,a.

Two potential global schedules are defined by the concurrent SPN specifications shown in Fig.2,b and c. In order to check whether these two schedules are weakly compatible with the register cliche we can refer to trace theory [11], in which an attractive synchronization operator between two trace structures may be effectively used for the purpose of such checking.

If $X = \langle aX, tX \rangle$ is a trace structure where aX is a finite alphabet of actions and tX is a set of traces, strings of symbols in aX, i.e. $tX \in (aX)^*$, then the *synchronization* of two trace structures X and Y is the trace structure defined by

$$X \,\&\, Y = \langle aX \cup aY, \{t \mid t \in (aX \cup aY)^* : t/aX \in tX, t/aY \in tY\} \rangle$$

where t/A stands for the projection of trace t on alphabet A.

Let $S1$ and $S2$ denote the trace structures generated by the schedules in Fig.2,b and c, respectively. Since we are interested in checking the fact that a schedule is fully executed, we include in $S1$ and $S2$ only those sequences of actions on the register that begin in the initial marking, p_b, and end in the goal marking, p_g. The trace structure R stands for the register and contains all prefix-closed traces satisfying the register's cliche.

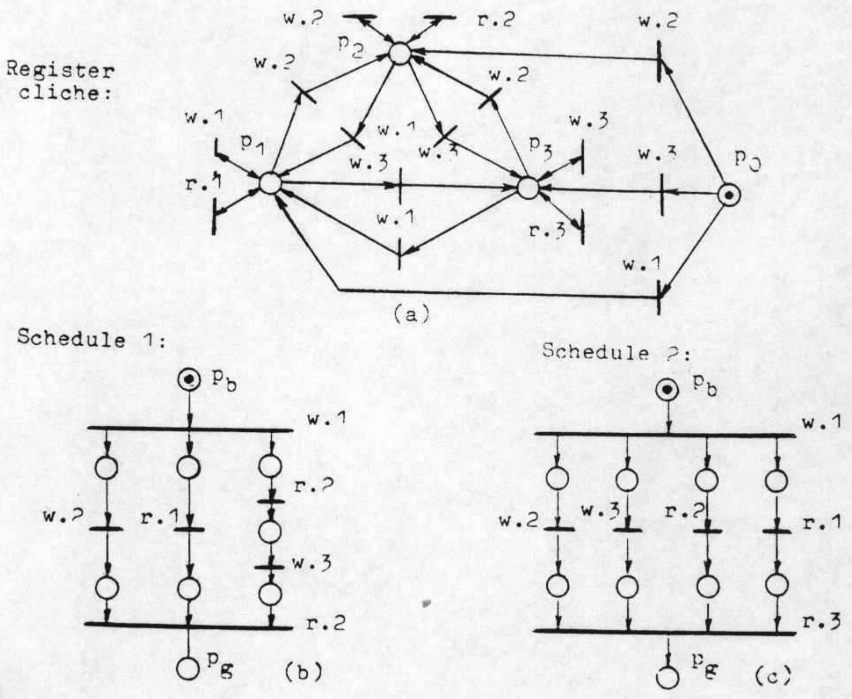


Figure 2. Illustration of weak compatibility

It is obvious in this example that

$S1 \& R = <aS1, \Phi >$ and
$S2 \& R = <aS2, \{w.1\ r.1\ w.2\ r.2\ w.3\ r.3\}>$,

from which we conclude that the first schedule is not, while the second is, weakly compatible with the register behaviour. We should point out, again, that the above analysis has been done in terms of the interleaving semantics of trace structures.

The strong form of compatibility, called *compliance*, requires that the global semantics must be strictly compatible with each component cliche,so that the projection of the global behaviour on the set of the component's state changes (or, generally, actions) is identical to the local cliche, in terms of causal semantics, i.e. using explicit sequentiality and concurrency relations. For example, in the above example with the register, since the latter is a sequential object, the global behaviour will be compliant if and only if it ensures that all the actions on this component are linearily ordered, and such a sequences of actions is weakly compatible, through the above synchronization operator, with the register cliche.

An important issue about the analysis of semantics of PN specifications is that, due to its more compact and intrinsic representation of concurrency, the partial order semantics is more convenient than the interleaving one. In addition, the class of PNs used for specifying protocols and controllers on the cycle layer involves PNs that are free from choices and represent a purely sequential-concurrent paradigm, which is semantically adequate to the relations depicted within the partial order semantics of PNs. This issue allows us to restrict ourselves with a subclass of Petri nets called marked graphs and present the following semantic analysis on the model for which we can take advantage of using, in practical applications, rather easy, i.e. polynomially hard, analysis procedures.

3. Signal Graphs and Their Semantics

A *marked graph* PN can be represented as a directed (monochromatic) graph whose vertices correspond to PN transitions and arcs to PN places. Therefore, it can be called a marked graph (MG) [12]. Bearing in mind such a correspondence, the firing rule for MGs is the same as for PNs. The *liveness* and *safeness* conditions for MG were studied in [12].

An interesting property of a live-safe MG whose simple cycles contain exactly one token each is that such an MG has a unique equivalence class of all its live-safe markings. Therefore, the behaviour of this MG is fully determined by its structure. Such a class of MGs is called *simple* MGs.

Similar to SPNs we consider the concept of a *signal graph* (SG), which is a triple $G^x = <G,DX,f>$ where $G = <V,E,M^o>$ is a MG graph with set of vertices V, set of arcs $E \subset V \times V$ and initial marking $M^o : V \to \{0,1,...\}$; DX is the set of allowed changes of states of variables in X (here, we restrict ourselves only with those specifications where the actions on finite-state components always change the state of the components); $F: V \to DX$ is a labelling function. We also accept that the underlying MG is live-safe and simple. It can easily be proved that such an MG, with respect to its initial marking, presents a partial order on the set of its vertices. Hence, the analysis of the partial order S-semantics of SPNs can be effectively reduced to the analysis of semantics of live-safe SGs.

To find a more fine relationship between N- and S-semantics of SGs we need a more careful study of the labelling function f. For the sake of clarity and due to the application details, it is sufficient to restrict ourselves with discrete components that have only linearily ordered local cliches.

Thus for an SG we introduce the concept of a compliant labelling function.

A labelling function f is called *compliant* for an SG $G^x = <G,DX,f>$ iff for each component the component state changes are linearily ordered and this ordering is compatible with the behavioural cliche of the component.

If X consists only of binary variables, then a compliant labelling guarantees that for each reachable marking and each variable x_i there is at most one enabled vertex labelled with the change of x_i, and for each sequence of variable transitions, between any two changes of the same sign dx_i there exists exactly one change of the opposite sign, $\wedge dx_i$.

An SG with a compliant labelling is called *coherent*.

Statement 1

A signal graph generates a consistent S-semantics iff it is coherent.

An important consequence of the above statement is that it also provides a condition for the correct implementation of a discrete structure. We distinguish two possible ways (in some sense, design semantics) for such an implementation which we call the *explicit control view* (ECV) and *implicit control view* (ICV).

The ECV makes use of the coherent SG (this can also be generalized for SPNs) specification as a model that can be directly translated into the circuit of a central control mechanism (if our target structure is a bus controller, this would be the bus controller controller), which conducts the behaviour of components in X by activating a corresponding state change, for example, through a request-acknowledge handshake in the component logic.

For a physical circuit realization of ECV-strategy, one may use one of the two major principles of PN circuit modelling [13]:

(i) modelling the places, i.e. associating a flip-flop cell with each PN's place and using the intercell signals, which model the labelled transitions, as signals for the above-mentioned request-acknowledgement links with the target structure's components; or

(ii) modelling the transitions, i.e. associating each transition of the PN with a special logical element in the so-called event-signalling control circuit (see also [14]).

Both of these two methods involve the design of an explicit control circuit which isomorphically models the PN's "token game" semantics, by changing the values of its actuator gates or flip-flops in accordance with movement of tokens through the PN, and has the same topological structure as the PN (or SG).

The ICV of an SG specification regards the SG as a behavioural scenario of directly interacting components that change their states by computing, at their inputs, their own transition functions, guards, without any additional control level. Using this strategy would result in a better design in terms of speed and area on the chip.

The results of [15] demonstrate that, within certain operation condition assumptions, there is a possibility to derive self-timed circuits in the ECV-strategy even from non-coherent specifications, using the so-called auto-correct implementation principle.

Checking the coherence of an SG, which opens the way for the ICV-strategy of circuit design, is done through the computation of the relations of *precedence* and *concurrency*, denoted respectively as => and || and built on set DX. The intuitive meaning of the => and || relations is as follows:

$dx_i \Rightarrow dx_j$ means that there exists a simple cycle in the SG such that dx_i precedes dx_j on this cycle with respect to the position of a token on the cycle (recall that every simple cycle in an SG contains exactly one token); this relation guarantees that dx_i and dx_j cannot be enabled concurrently;

$dx_i \parallel dx_j$ means that there is no simple cycle on which dx_i and dx_j reside, so there is a possibility for them to be enabled concurrently.

One should also observe that || is the negation of another relation,<=>, that is simply a reflexive closure of =>, i.e. <=> = (=> U <=), which of course means "there is a simple cycle on which dx_i and dx_j reside". It is important to note that neither of these relations is transitive: although we may have $dx_i \Rightarrow dx_j \Rightarrow dx_k$, it is possible that $dx_i \parallel dx_j$ because dx_i and dx_j can be ordered in one cycle, while dx_j and dx_k in another.

Since, from the algorithmic viewpoint, the definition of => through the concept of cycles may lead to inefficiency in computations, the formalisation of these relations stipulates using a slightly different approach. We define a concept of an operation history, called ***unfolding***, which is an infinite and acyclic graph generated by the original SG. The unfolding is similar to the occurrence net for a PN. Each occurrence of a vertex dx_i in the SG yields a unique vertex in the unfolding, having the same label dx_i augmented with a unique index of occurrence,$dx_i(k)$. A set of vertices with index k in the unfolding, together with the corresponding arcs, is called the k-th *period* of unfolding.

It can be shown that, for practical analysis purposes, the unfolding can be floored to its first two periods, and the => and || relations can be computed on such a finite representation pattern (a characteristic image, or a fixed-point, of the whole partial order history): namely, $dx_i \Rightarrow dx_j$ iff $dx_i(1) \rightarrow dx_j(1)$ and $dx_j(1) \rightarrow dx_i(2)$, where -> means the fact of existence of a simple path in a two-period graph between corresponding occurrence vertices. Based on the properties of unfolding, one can easily prove the following property.

Property 1.

If any four transitions t_1, t_2, t_3, and t_4 , in a live-safe and simple SG, satisfy the following precedence relationship:

$$t_1 \Rightarrow t_2 \Rightarrow t_3 \Rightarrow t_4 \Leftarrow t_1,$$

then there exists a simple cycle where these transitions are ordered, with respect to the initial position of a token, exactly in the order of their subscripts.

The complexity of this computation is $O(n^3)$ where n is the number of vertices in the SG. Similar observations for a model called change charts in [16] were made in [17].

Since the ICV implementation of a discrete structure of a bus controller is more attractive, the further analysis of the S-semantics of a coherent SG is necessary.

4 The ICV Design Strategy and SG Specification Completeness

Although a coherent SG generates the consistent S-semantics, this may be insufficient for designing a circuit within the ICV-strategy. The problem is that a coherent SG may generate a state diagram which has the so-called multiple states, states labelled with equal n-tuples of component states. The state diagram in this case is called *contradictory*. Informally, the contradiction, which exhibits itself as a form of indeterminism, implies that the system is under-specified in the ICV terms, and some components are "hidden" from the designer's knowledge. For example, for the case of using SGs in specifying an interface signalling protocol, these components may be interpreted as a control flow memory, internal for the controller, which is needed for the final implementation stage but has been unnecessary for the protocol specification purposes. Hence, in the ICV terms this effect is seen as incompleteness of the initial specification, and a certain number of additional variables need to be introduced into the systems, thereby ensuring that all global states in the state diagram are unique. For that we define the notion of a normal SG.

An SG is called *normal* iff it is coherent and for each allowed sequence of vertex firings it has no proper subset of components $X' \subset X$ which may proceed through their full cycles of changes of their values while the other, in $X \setminus X'$, remain unchanged.

It can be shown that the state diagram of a normal SG is non-contradictory and distributive. The distributivity is defined within a lattice-theoretical characterisation of S-semantics [16].

A more interesting and practically efficient way of characterization of SG normalicy can be achieved through a causal approach. We consider an operational relation between components, which is called the coupledness relation[1], originally proposed in [19] and later used in [20].

Again, for clarity, we assume the following, however not so crucual, restrictions. First, we consider the case of binary discrete structure, i.e. $S(x_i) = \{0,1\}$, and dx_i and $\hat{}dx_i$ denoting the two mutually opposite state changes of x_i. Second, we consider only such SGs for which the labelling function f is a direct one-to-one mapping of V into DX, i.e. each transition of a particular dx_i labels only one vertex in the SG.

Let a coherent SG be given. Then variables x_i and x_j are called:

- *directly strongly coupled*, $(x_i,x_j),(x_j,x_i) \in \underline{\text{dsc}}$, iff there exists the following precedence combination

  $dx_i \Rightarrow dx_j \Rightarrow \hat{}dx_i \Rightarrow \hat{}dx_j \Leftarrow dx_i$,

  with respect to a given initial marking (recall Property 1);

- *strongly coupled*, $(x_i,x_j) \in \underline{\text{sc}}$, iff $(x_i,x_j) \in \underline{\text{dsc}}^*$ where * stands for the transitive closure;

- *weakly 1-coupled of rank r*, $r >= 0$, $(x_i,x_j) \in \underline{\text{wc1}(r)}$, iff there exists the following precedence combination

  $dx_i \Rightarrow dx_j \Rightarrow \hat{}dx_i \Rightarrow dx_k \Leftarrow dx_i$,

  with respect to a given initial marking, and $(x_j,x_k) \in \underline{\text{link1}(r)}$ where

$$\underline{\text{link1}(r)} = \begin{cases} \underline{\text{sc}}, \text{ if } r = 0 \\ ; \\ (\underline{\text{link1}(r-1)} \cup \underline{\text{wc1}(r-1)})^*, \text{ if } r > 0 \end{cases}$$

[1] We have recently become aware of the work of P. Vanbekbergen et. al. (see, for example, [18]), who, to our great surprise, use a very similar technique based on the lock relation between variables and interleaving relation between transitions. Their interleaving condition helped us to correct our previous coupledness hierarchy introducing the case of 2-coupledness.

- *1-coupled*, $(x_i, x_j) \in \underline{cp1}$ iff $(x_i, x_j) \in \underline{\text{link1}(r_{max1}+1)}$ (the maximum rank, r_{max1}, of the weak 1-coupledness is determined by the following termination condition $\underline{\text{wc1}(r_{max1}+1)} = f$);

- *weakly 2-coupled of rank r, r>=0*, $(x_i, x_j) \in \underline{\text{wc2}(r)}$,
  iff there exists the following precedence combination
  $dx_i => dx_j => dx_l => dx_k <= dx_i$,
  with respect to a given initial marking,
  and $(x_j, x_k), (x_i, x_l) \in \underline{\text{link2}(r)}$ where

$$\underline{\text{link2}(r)} = \begin{cases} \underline{\text{cp1}}, \text{ if } r = 0 \\ ; \\ (\underline{\text{link2}(r-1)} \cup \underline{\text{wc2}(r-1)})^*, \text{ if } r > 0 \end{cases}$$

- *2-coupled (or, simply, coupled)*, $(x_i, x_j) \in \underline{cp}$, iff $(x_i, x_j) \in \underline{\text{link2}(r_{max2}+1)}$ (the maximum rank, r_{max2}, of the weak 2-coupledness is determined by the following termination condition $\underline{\text{wc2}(r_{max2}+1)} = f$);

The coupled relation partitions the set of variables X into disjoint classes of coupledness, which help us claiming the following.

Statement 2

A coherent signal graph is normal iff all its components belong to the single coupledness class.

An example of a normal SG is shown in Fig.3. It illustrates all of the above coupledness paradigms.

Using the primary relation, precedence (=>), we can check normalicy for an SG with polynomial complexity.

At its worst case, the checking procedure, which involves construction of coupledness classes, may require checking the precedence conditions at all the levels of coupledness hierarchy:

first, checking the <u>dsc</u> condition and adding new members to the <u>sc</u> classes, which involves searching through pairs of variables (and, thus, has time complexity of $O(n^2)$);

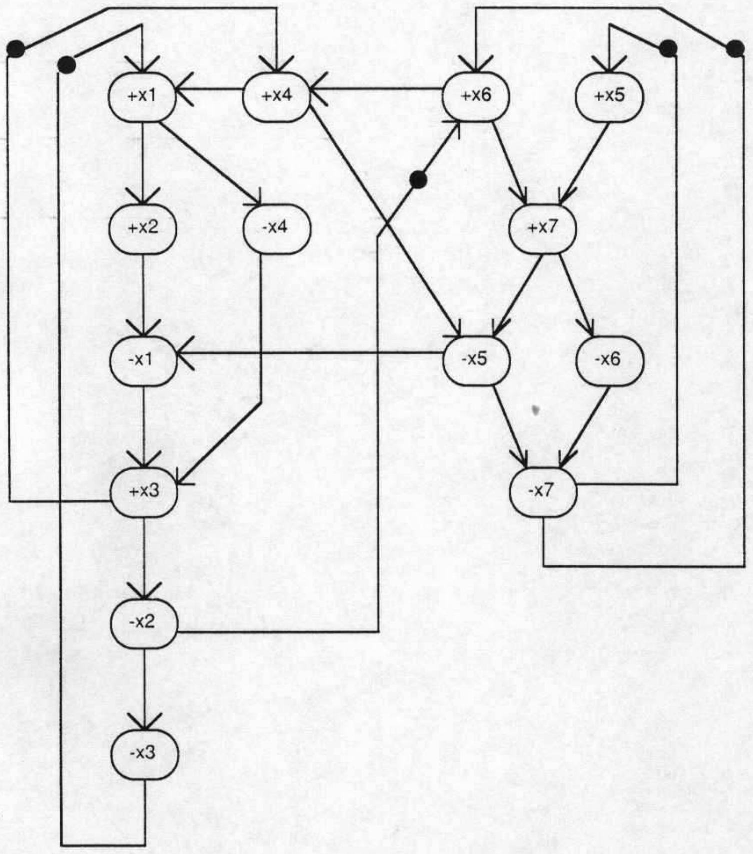
second, checking the <u>wc1(r)</u> condition and adding new members to the <u>cp1</u> classes, which involves searching through triples of variables and moving up the rank value (thus, requiring $O(n^4)$ steps);

third, checking the <u>wc2(r)</u> condition and adding new members to the <u>cp</u> classes, which involves looking through combinations of four variables and moving up rank value ($O(n^5)$ steps). the

(Note that in the second and third stages we have $r_{max} =< n$.)

Hence, the worst case cost of normalicy check, based on the preconstructed precedence relation, is $O(n^5)$.

The main advantage of the relation-based technique for analysing the specification completeness is that it does not need to work with the exponentially hard (with respect to n) interleaving semantics of a global state diagram.



dsc = {(x1,x2),(x2,x3),(x5,x6),(x6,x7)}

sc-classes: {x1,x2,x3}, {x4}, {x5,x6,x7}

wc1(1) = {(x1,x4)}

c1-classes: {x1,x2,x3,x4}, {x5,x6,x7}

wc2(1) = {(x4,x6)}

cp-classes: {x1,x2,x3,x4,x5,x6,x7}

Figure 3. Example of a normal signal graph

It is also interesting to compare the above coupledness relation hierarchy technique with the method of analysis of signal-transition graphs from [3], which are similar to our SGs. In order to deal with the problem of contradiction in a state diagram, the author of [3] introduces the so-called persistency (which should not however be confused with the persistency property used for PNs) constraint on the relations of S-semantics between SG vertices. In terms of the above coupledness relations, this constraint allows to use only such SGs in which all the components are strongly coupled. In some cases, this constraint over-bridles the concurrency paradigm potentially admissible in the behaviour, which may consequently slow down the final implementation's operation.

5 Delay-Insensitive Implementation

The process of deriving a logical circuit implementation for a binary signal graph specification, using the ICV strategy, starts through checking the normalicy condition and, if needed, introducing auxiliary variables into the structure. The corresponding vertices labelled with the changes of states of these variables must be inserted into the SG, thereby:

(i) providing necessary operational coupling between the components belonging to separate coupledness classes, and

(ii) preserving the original causal semantics between the components.

Then, in order to derive logical equations for the components which are either output signals or internal memory elements, we convert our specification into the state diagram format, from which, by an appropriate Boolean function minimization algorithm, we obtain the corresponding guard functions for the implementation.

Of course, if the number of components is large enough, the construction of a global state diagram is rather unpleasant. To deal with complexity, we suggest using the concept of *projection* of an SG on some set of relevant variables. This is quite similar to what is called net contraction in [3]. With the help of this concept, for each implemented variable, we build the *minimal normal* SG projection including only the direct predecessors of the transitions of this variable (as well as the remaining transitions associated with the variables involved in such predecessors). Taking the target variable of this projection as the output component we make the minimization procedure and obtain the corresponding logical guard function (or two functions, one for the 0-1 transition (the S, set, function) and the other for the 1-0 transition (the R, reset, function) in the form

$$x_i = f_i(x_1, x_2, \dots, x_n)$$

(or in the form $x_i = S_i + x_i R_i$ where S_i and R_i are independent of x_i).

It is easy to prove that the set of SG projections covering the causal semantics of the original SG, under the application of the synchronization operator, yields the interleaving semantics which is equivalent to the semantics of the original SG. This gives necessary justification to using the implementation technique based on the projections of an SG.

Provided that the derived logical guard function of the component is implemented on a single physical element, we obtain a delay-insensitive self-timed circuit whose behaviour strictly satisfies to what has been defined by the original SG specification. The delay-insensitivity is however restricted only with respect to the delays that can be attached to the outputs of these elements. This includes the element's own delays and the delays in wires prior to their possible forking. In many cases "after-fork" delays are also tolerated by the self-timed discipline, but, for example, in cases of using self-dependent, flip-flop, elements the local feedback connections are always assumed to be delayless. The VLSI implementation of wire-delay-sensitive parts should follow the isochronic region layout principle [21].

6 Designing a Self-Timed VME-bus Data Transfer Controller

In this section we demonstrate a real application example for the above technique. In our design of a VME-bus SLAVE-INTERRUPTER controller chip, we used this technique for specifying and deriving the implementation of a reactive part of the controller. Such a part, called SYNCLOGIC, forms a synchronization skeleton of the controller and provides, as its primary goal, the necessary causal coordination of both the actions performed on the bus control lines and the sequencing organized at the internal links of the module, between the controller chip and the logic of the target SLAVE-INTERRUPTER operation mode device (such devices are quite often used in various instrumentation applications). SPNs and SGs were used for specifying the address broadcast cycles, data read/write cycles and interrupt cycles, as well as for defining the behaviour of SYNCLOGIC.

During the design process, a number of alternative SG solutions gave rise. The pursuit for a good semantical tradeoff between largest possible parallelism in the dynamic behaviour of the signal sequencing and minimum circuit complexity plus higher circuit structuredness resulted in the SG specifications which, initially, were not normal, in the above formal sense. The insertion of auxiliary variables allowed us to proceed to the final implementation step, using the ICV strategy.

It is important to point out the following. Because the methodology of using PNs in designing hardware logic is attributable, at its major extent, to the "reactive parts" of the circuit design, our discussion here eliminates any details related to the adjacent "transformational cosmetics", which would of course be necessary for the integrity of the design description, from the wholistic approach. Rather, we suggest that the reader having a large experience in bus standards and logic design support and/or more interested in concrete aspects of our circuit design would refer to [22] for these details, but, here, would pay more attention to the methodology of designing reactive hardware (conceptually, close to the general ideas of [5]), which play crucial role in such design objects as interface controllers. This will of course stipulate some relaxations of the various transformational details of the chip functioning.

6.1 VME-bus Slave Module Specification

In the present discussion, for the sake of clarity, we restrict ourselves with looking only into the SLAVE part of the above-mentioned controller chip design.

The operation of the SLAVE module involves participation in the two basic types of cycles: addressing and data transfer cycles [23]. The top view upon the functional module with the VME-bus SLAVE capabilities is shown in Fig. 4. The standard meaning of the bus lines involved is as follows:

ADDRESSING LINES: A01-A31, AM0-AM5, DS0, DS1, LWORD;
DATA LINES: D00-D31;
CONTROL LINES: AS, DS0, DS1, DTACK, BERR, WRITE.

The two data strobes, DS0 and DS1, serve a dual function:

(1) the levels of these strobes are used to select which byte(s) are accessed,
(2) the edges of these strobes are also used as synchronization signals which coordinate the transfer of the data between MASTER and SLAVE.

Signals A01-A31, LWORD are used for addressing the data and specifying which byte locations within the 4-byte group are accessed during the data transfer cycle.

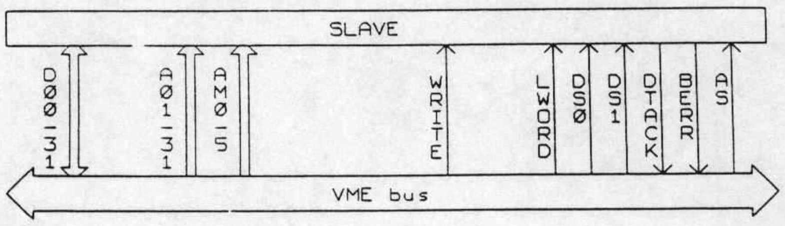


Figure 4. VME-bus SLAVE module

Signals AM0-AM5 (address modifier) are used to indicate which address bit groups are valid within the 32-bit address during the addressing cycle.
Signal WRITE is used to command which of the two data transfer directions between MASTER and SLAVE is needed, the read (WRITE = 0) and write (WRITE=1) operations.
Signal DTACK (data transfer acknowledge) is used by SLAVE to indicate that the data has been successfully received on a write cycle, or placed on the bus on a read cycle.
Signal BERR (bus error) is used by SLAVE to indicate to the MASTER that the data transfer (read or write) was unsuccessful.

The original protocol specifications taken from [23] are defined by timing diagrams. An example of such a diagram, defining the data transfer sequencing during the read operation, is presented in Fig.5,a. In this diagram a special notation is used for data strobes, DS0 and DS1, which do not always make their transition simultaneously. Therefore, DSA represents the first data strobe to make its transition (whether that is DS0 or DS1). The reason why signals DTACK and BERR "share" the same waveform is that they operate in alternative cases (either DTACK or BERR).

Following the technique described in [4], we represent the same protocol by Petri net, as shown in Fig.5,b. A special notation is used here for depicting the bus line signals which are modelled by the places with corresponding names. Note that the inhibitor arcs are used in this notation for describing the condition produced after the resetting of a signal on a bus line. The transversal lines on arcs leading from the "bus line" places are used for better readability of the model and their relationship with usual PN notation is shown in Fig.5,c. Also, as in [4], we use a special notation for "transformational" elements corresponding to the signals serving as information signals, either parallel bus signals, e.g. A01-A31, or command signals, e.g. WRITE, which are either decoded and/or strobed. This notation, called Data transitions, uses input and output data places that can be marked with data tokens and the control place marked with an ordinary, "control", token. Briefly (for more detailed treatment, see [4]), data transitions fire under the presence of a token in the control place and the firing process involves copying the data token from the input data place into the output data place. Such a behaviour is similar to a gate in the data path.

6.2 Structural Decomposition of SLAVE Controller

After defining the behavioural specification of the SLAVE module at the VME-bus link we proceed to the structurization of the controller,first, through establishing the interfacing between the controller and the target slave device, and, second, through separating a reactive part of the controller from the two transformational units, the data transceiver module (TR) and address decoder (DC), as shown in Fig.6.

The meaning of the signals created as a result of such structurization is as follows:

LDS - local data strobe, LDTACK - local data transfer acknowledge, LBERR - local error indication, SL_SEL - target slave device selection, DEN - data transceiver enabling, VAL_AD - valid address, LDB0-LDB3 - data byte number code. The meaning of the remaining signals, LD00-LD31, LA01-LA31 and LAM0-LAM5 is obvious.

Subsequent refinement of the controller's reactive substructure allows to single out two more levels of decomposition yielding several more transformational circuit layers. The first decomposition, shown in Fig.7, gives two transformational units, DS_ER_CHECK and SLAVE_SELECT, and one reactive unit, SYNCHRO.

DS_ER_CHECK is the circuit producing the double-rail error condition signal, ER and NER, and the combined data strobe signal DSI, corresponding to the DSA signal in Fig.5.

SLAVE_SELECT is the circuit producing the target device selection signal which combines the effect of AS and VAL_AD, as a result of detection of the valid address, with the possibility of pipelining the

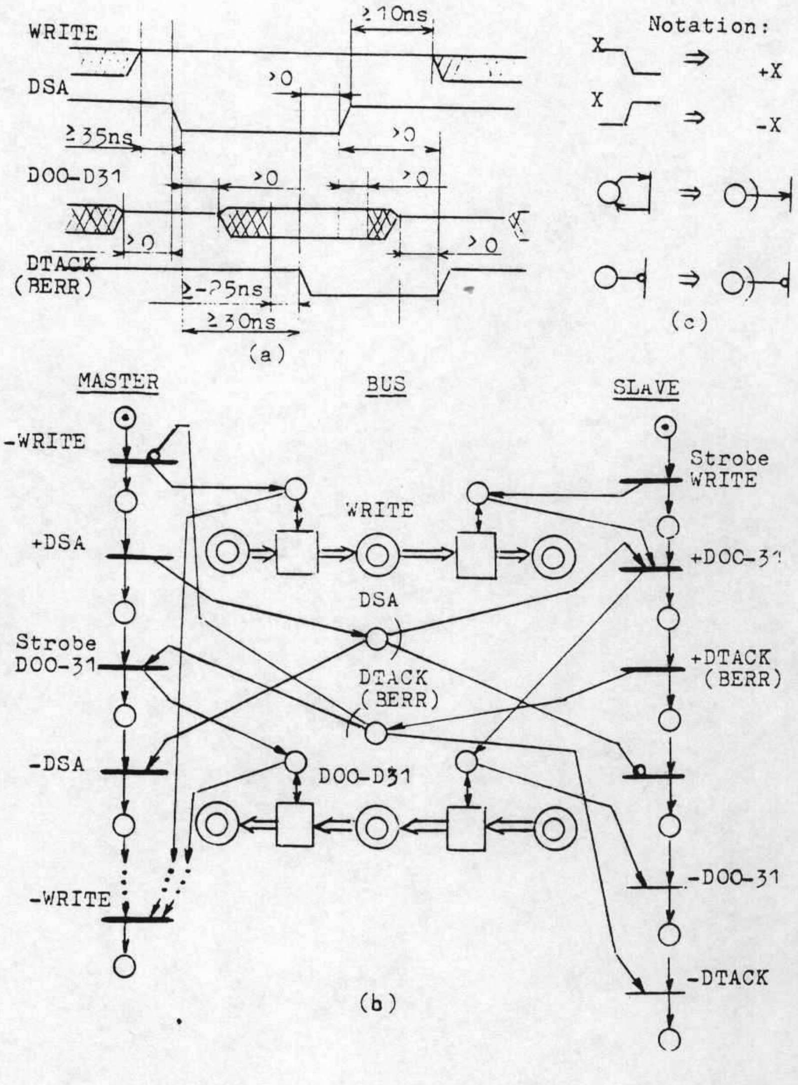


Figure 5. Data read protocol specification

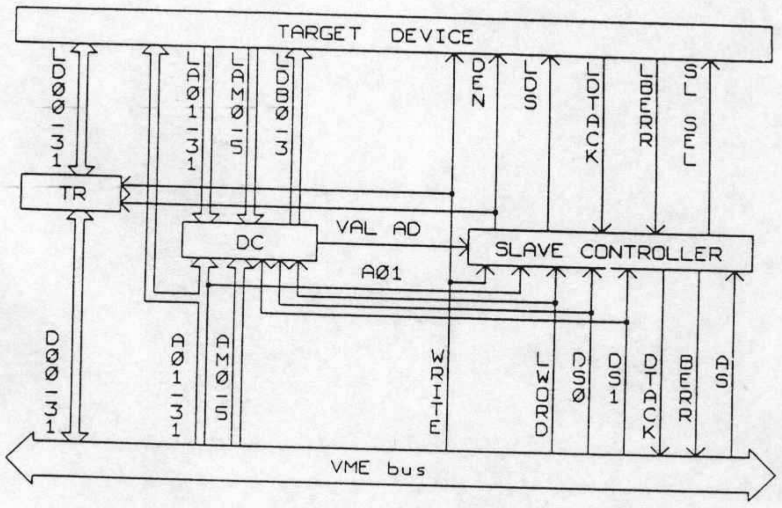


Figure 6. Structure of SLAVE module

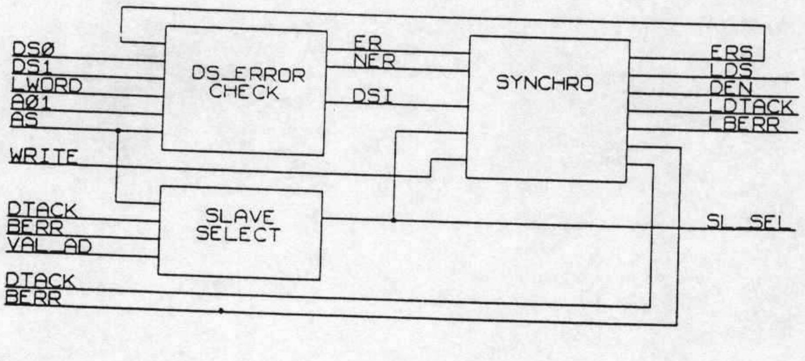


Figure 7. Decomposition of SLAVE controller: first step

current addressing cycle with the completion of the previous data transfer cycle (the latter is provided by the use of DTACK and BERR pair).

The main task of SYNCHRO is the coordination of the signalling at the links with the bus (through condition/command signals ER, NER, WRITE, and synchronization signals DSI, DTACK and BERR), with the target device (through synchronization signals LDS, LDTACK and LBERR), with transceiver unit TR (through signal DEN) and the coordination of the action internal for the controller, strobing the error conditions by means of signal ERS.

The second decomposition step refines the SYNCHRO unit as shown in Fig.8. This gives us a reactive kernel of the controller, the SYNCLOGIC unit, surrounded by several transformational units:

READ_FF and WRITE_FF, the flip-flops producing the unitary synchronization signals DSR and DSW, for the read and write cycles respectively;

CON, the single-rail-to-double-rail code converter for the WRITE signal;

the OR gate merging the mutually exclusive LDTACK and LBERR signals into the single LACK (local acknowledgement) signal;

ACK_FF and BERR_FF, the pair of output flip-flops producing the DTACK and BERR signals as results of combining the error conditions from the device with error conditions manifested by ER and NER values; and

the OR gate producing the ERS signal to the DS_ER_CHECK unit.

6.3 Behavioural Specification of SYNCLOGIC

Starting at this point, we convert the top level specification of the type shown in Fig.5,b into an SPN (or SG) specification of the bus side link of the SYNCLOGIC. In doing so, we preserve the original ordering between transitions bearing in mind that we may substitute DSI for DSA because DSI is just a result of delayed transformation of DS0 and DS1. Similarly, we substitute ACK for DTACK (BERR). In the same way we build the specification for the target device link involving the handshake signals LDS and LACK. These two partial views upon the controller, during the data read operation, are shown by SGs in Fig.9,a and b.(For further readability, data strobe signal DEN is denoted by D.) In order to distinguish which of the signals are the inputs to the controller and which are the outputs generated by the controller components, we use boxes and circles for depicting their respective vertices in the SGs. The inclusion of the vertex labelled by +D into the signalling of Fig.9,b respects the order of strobing the data from the device into the transceivers after the device's acknowledgement with +LACK.

The semantically consistent, with respect to the read operation meaning, coordination of signalling at both links can only be achieved if we include also the ordering between DSI and LDS transitions shown in Fig.9,c.

As the overall specification of SYNCLOGIC, during the read operation, must be compliant with all the above three partial signalling orders, it can be obtained as a result of synchronization operation (this operation is associative [11]) on them, which is presented by the SG in Fig. 9,d.

Now, for more convenient implementation, we can make a transformation of this specification to a more compact form. It is based on the so-called *handshake compression principle*, often used in the self-timed design discipline [13].

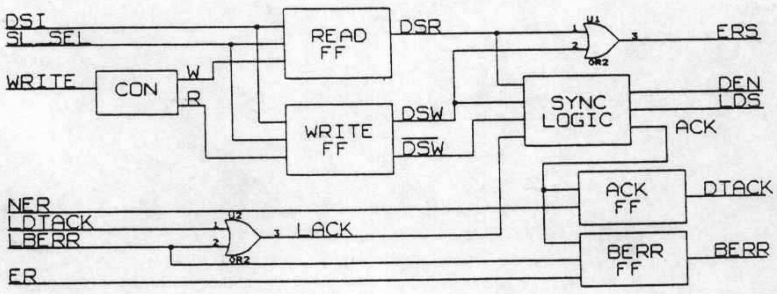


Figure 8. Decomposition of SLAVE controller: second step (refining SYNCHRO)

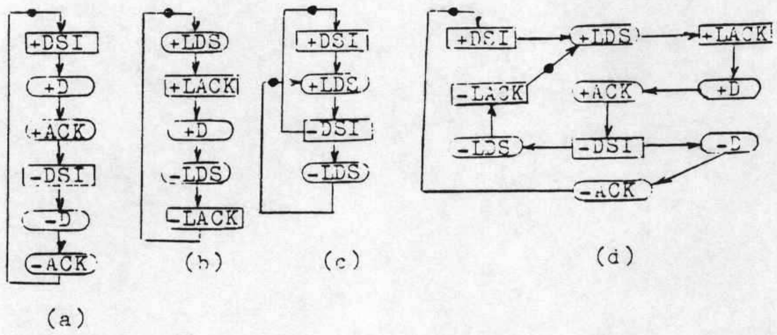


Figure 9. Constructing behavioural specification of SYNCLOGIC from partial views

The handshake compression principle.

From the viewpoint of a self-timed circuit any request-acknowledgement pair of variables can be represented by a single (output) variable produced by the circuit. The synthesis of the circuit is thus performed on a reduced number of variables. After obtaining the logical functions for these components, one can return back to the original handshake pairs by simple breaking the output wires of the compressed components and thus forming the necessary pair. Two important conditions should however be ensured when this principle is applied. The first is that the transitions of the request signal, in such a pair, may be direct predecessors only for their acknowledging counterpart and for none of the other signals. The second is that depending on the initial value of request-acknowledgement variables, one should sometimes insert an invertor into the above-mentioned break before connecting the incoming request to the newly produced input.

The result of applying the above principle is shown in Fig.10,a where variable B (from "bus") stands for the (ACK,DSI) pair and L (from "local") for the (LDS,LACK) pair. This SG has very clear and easily understandable interpretation of its own: the initial marking corresponds to the situation when the controller is ready for receiving the activation from the bus (-B). After this, controller activates its link with the target device, and when the data is produced by the device (+L), it can be strobed into the transceivers (+D). This leads to sending an acknowledgement to the master and waiting a reset of his strobe (+B), after which the two processes may go concurrently: the reset of the local link (-L), and the reset of the data in the transceivers followed by the reset of the acknowledgement and the "invitation" of the bus link for another data transfer cycle.

The signal graph shown in Fig.10,b is obtained in the same way for the write operation cycle.

6.4 Implementation of SYNCLOGIC

We illustrate the process of using the formal technique introduced in Sections 3,4 and 5 for the SG shown in Fig.10,a.

First, the result of checking this graph shows its coherence - all the transitions associated with each component have a compliant labelling. But the SG is not normal - we have two coupledness classes {L} and {D,B} (note that D <u>dsc</u> B). Second, to have necessary coupling, we introduce an auxiliary variable, S_r, as shown in Fig.11, which gives us the desired relations: (S_r,B) <u>dsc</u> and (S_r,L) <u>dsc</u>. Due to transitivity, all four variables are now in the same coupledness class. Third, using the method of component projections, we derive the logical circuit implementation of the given SG as shown in Fig.11 for the Sr component. The final read control circuit is shown in Fig.12. The original handshake pairs are restored here, providing the insertion of an invertor at the DSI input. This circuit is delay-insensitive with respect to the delays of all logical elements and all wire delays except for the wires marked with "#".

In the same manner, we could derive the implementation for the write cycle specification. Then both implementations could be combined by using special merging logic for the shared variables, B, L and D. Unfortunately, the introduction of extra circuitry, the merging gates, leads to slowing down the operation of the controller. This had been the main reason for us to choose a slightly different implementation way in real chip design. We made a combination of the read and write operations before we proceeded to the logical functions for components. We used the SPN shown in Fig.13, which is the choice-based composition of SG specifications shown in Fig.10 (for the purposes of better physical implementation we have slightly restricted parallelism, by introducing the order between transitions -D and -L) augmented with separate bus link signals, B_r and B_w (see DSR and DSW in Fig.8), and two separate coupling variables S_r and S_w. These augmentations guarantee that the S-semantics of the PN in Fig.13 is

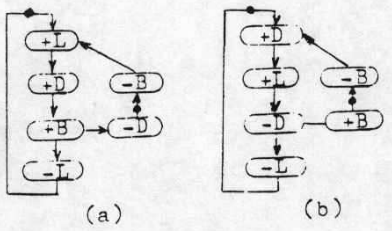
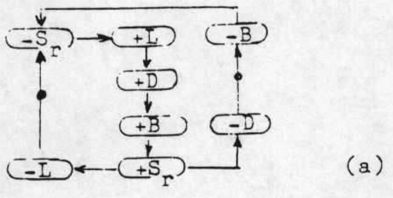
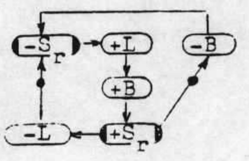


Figure 10. "Compressed" signal graph specifications for data read(a) and data write (b) operations





$\Rightarrow$

L B S_r
011
001
000
100
110
111
101

$\Rightarrow$

S_r:

S_r \ LB	00	10	11	01
0	0	0	1	*
1	0	1	1	1

$\Rightarrow S_r = B + S_r L$

"*" is a"don't care" value

(b)

Figure 11. Implementation of SYNCLOGIC: normal signal graph (a), deriving logical functions (b)

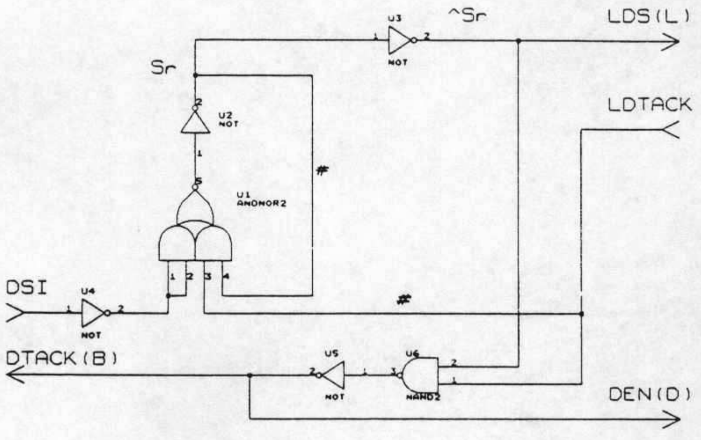


Figure 12. Read control circuit

consistent and complete, which is demonstrated by the global state diagram shown in Fig.14. The final version of SYNCLOGIC was obtained from this diagram.

7 Conclusion

We presented a formal technique for designing reactive hardware, which is based on signal-labelled Petri nets and their important subclass, signal graphs. The technique has been effectively tested on such an instructive example as asynchronous parallel bus controllers whose behaviour has both the concurrency and reactiveness paradigms. In combination with the basic ideas of reactive system development methodology originated in the work by Pnueli and Harel [5] and a stepwise (transformational+reactive) decomposition/refinement process, this technique can be very useful for VLSI design community.

Subsequent theoretical efforts could be directed towards studying the ways of more graceful composition technique for the usually separately specified bus protocol cycles (a set of timing diagrams defined on the shared signalling domain). One of the efficient compositional methods could likely be obtained by using some simple forms of high-level Petri nets [24], where the individual tokens might represent the various cycle modes of bus protocols thereby modelling different reactive skeletons on the same signal transition domain. An example of such a Petri net, for the above read and write cycles in SYNCLOGIC is shown in Fig.15. Within this framework, it is desirable to obtain some clear control flow characterisation similar to the ECV/ICV strategies described in the present paper.

Acknowledgement

We would like to thank our colleagues from Leningrad Electrical Engineering Institute and Russian Academy of Sciences, Science and Technology Corporation, for supporting our work, at its major stages. We are also grateful to our colleagues in Polytechnig Cymru (Polytechnic of Wales, U.K.) and Tekniska Laroverket i Helsingfors (Swedish Institute of Technology, Helsinki, Finland) for their help at the final stage.

Prof. Leo Rosenblum, from Boston, Massachussetts, is kindly acknowledged for stimulating discussions about signal graphs.

Special thanks to Luciano Lavagno, from Berkeley, California, and Peter Vanbekbergen, from IMEC Labs in Belgium, - the first pointed us to the work of the second.

References

1. J.B. Dennis, "Modular, asynchronous control structures for a high performance processor," Proc. Project MAC Conference, pp.55-80,1970.
2. D.P. Misunas, "Petri nets and speed independent design," Comm.ACM, Vol.16, No.8,pp.474-481, 1973.
3. T.-A.Chu, "Synthesis of self-timed VLSI circuits from graph-theoretic specifications," Proc. Int. Conf. Comput. Design (ICDD'87), pp.220-223, 1987.
4. P. Civera, G. Conte, D. Del Corso, F. Maddaleno, "Petri net models for the description and verification of parallel bus protocol," Computer Hardware Description Languages and their Applications, M.R. Barbacci and C.J. Koomen (Eds.), Elsevier (North-Holland), pp.309-326, 1987.
5. D. Harel, A. Pnueli, "On the development of reactive systems," Logics and Models of Concurrent Systems, K.R.Apt (Ed.), NATO ASI Series F, Vol.13, Springer-Verlag, pp.477-498, 1985.

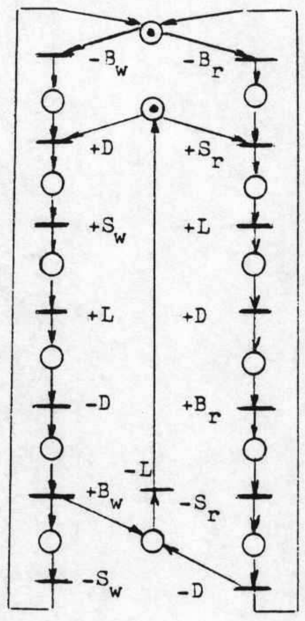


Figure 13. Using signal Petri net for SYNCLOGIC specification

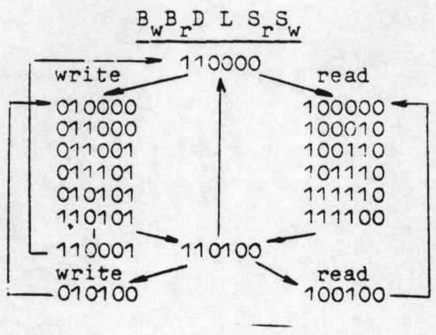
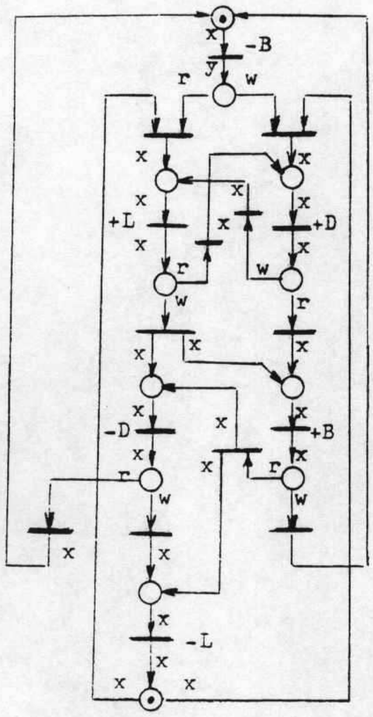


Figure 14. State diagram for the SPN in Fig. 13



r = "READ"; w = "WRITE"

y = F(x) is the function modelling the master's setting of the WRITE line

Figure 15. Using signal Petri net with individual tokens for SYNCLOGIC specification

6. A. Yakovlev, "Designing self-timed systems," VLSI Systems Design, Vol.6, No.9,pp.70-90, Sept. 1985.
7. L.A. Hollar, "Direct implementation of asynchronous control units," IEEE Trans. Comput., Vol. TC-32, No.12, Dec. 1983.
8. W. Reisig, "Concurrency is more fundamental than interleaving," EATCS Bulletin, No.35, June 1988.
9. L. Rosenblum, A. Yakovlev, V. Yakovlev, "A look at concurrency semantics through lattice glasses," EATCS Bulletin, No.37, pp.175-180, Feb.1989.
10. J. Misra, "Axioms for memory access in asynchronous hardware systems," Lecture Notes in Computer Science, Vol. 197, Springer-Verlag, pp.96-110, 1985.
11. A.Mazurkiewicz, "Trace theory," Lecture Notes in Computer Science, Vol.255, Springer-Verlag, pp.279-324, 1987.
12. F.Commoner et al., "Marked directed graphs," J. Comp. Syst. Sci., Vol.5, pp.511-523, 1971.
13. V.I. Varshavsky et al., Self-Timed Control of Concurrent Processes, Kluwer AP, 1990.
14. I.E. Sutherland, "Micropipelines: The Turing Award Lecture," Comm.ACM, Vol.32, No.6, pp.720-738, June 1989.
15. V.I. Varshavsky, M. Tiusanen, "Hardware support of concurrent process interaction and synchronization: on the principle of autocorrect implementation," Techn.Report of Helsinki University of Technology, Digital Systems Laboratory, 1988.
16. R.E. Miller, Switching Theory,Vol.2, Chapter 10, Wiley, N.Y., 1965.
17. V.I. Varshavsky et al., "On the models for specifying and analysing processes in circuits," Soviet J. Comp. Syst. Sci. (USA), 1988, No.2, pp.171-190.
18. P.Vanbekbergen, et al., "Optimized synthesis of asynchronous control circuits from graph-theoretic specifications", Proceedings of ICCAD-90.
19. A.V. Yakovlev, Design and Implementation of Asynchronous Bus protocols, Ph.D. Thesis, Leningrad Electrical Engineering Institute, Leningrad, 1982 (In Russian).
20. A.Yu. Kondratyev, L.Ya. Rosenblum, A.V.Yakovlev, "Signal graphs: a model for designing concurrent logic, Proceedings of ICPP'88, Pennstate University Press, University Park, PA, 1988, p.51-54, Vol.1.
21. C.L. Seitz, System Timing, Chapter 7 in: Introduction to VLSI Systems, Addison-Wesley, Reading, 1980.
22. A.V. Yakovlev, A.I. Petrov, "The architecture and circuit implementation of VME-bus controllers," Research Report for Science and Technology Corp., USSR Academy of Sciences , Leningrad, 1989 (In Russian).
23. VME-bus Specification Manual, Rev. C.1, VITA, Temple, AZ,Oct. 1985.
24. W. Reisig, "Petri nets with individual tokens," Informatik-Fachberichte 66, pp.229-249, 1983.