# Decomposing Signal Transition Graphs

**M. A. Escalante [1] and M. H. M. Cheng [2]**

[1] *Department of Electrical and Computer Engineering*
[2] *Department of Computer Science*

University of Victoria

## Abstract

*Nets, process algebras and modal logics are alternative representations of concurrent processes. This work tackles the problem of decomposing a subclass of interpreted Petri nets used in the specification of delay-insensitive interface protocols of VLSI chips called signal transition graphs. The decomposition procedure generates a representation of the graph in Milner's Calculus of Communication Systems (CCS) such that the behavioral properties of the protocol are preserved. The resulting protocol representation in CCS is compositional and amenable to formal methods of verification. For instance, the Edinburgh Concurrency Workbench has been utilized to study the behaviour of protocols which were originally represented by signal transition graphs.*

## 1 Introduction

Timing diagrams are commonly used to describe the interface behaviour of hardware components. Signal transition graphs (STG) have been used to specify interfacing protocols which can be derived from timing diagrams [4]. An STG is a Petri net description of the interface behaviour of components, which abstracts out unnecessary implementation details. However nets in general suffer from the complexity problem: while toy problems are easy to solve, large-scale real systems are difficult to analyze. Furthermore there is no known procedure that combines arbitrary Petri nets to construct more complicated systems. On the other hand composition is an integral part of process algebras [8]. This paper presents a procedure to decompose STG's into agent expressions in Milner's Calculus of Communicating Systems (CCS). As a result the new specification is now composable.

In section 2, related work on the area of specification of delay-insensitive VLSI circuits is introduced. The specification of a bus arbitration protocol serves as a motivation example in section 3. Signal transition graphs are presented in section 4 in the context of hardware protocol specification. In section 5 Milner's CCS is briefly introduced. The procedure to decompose STG's into CCS agent expressions is expounded in section 6. This paper concludes with the final remarks and future work.

## 2 Related work

Traditionally either simulation or exhaustive verification have been used to prove correctness of VLSI systems. However as the system complexity increases, the use of formal verification methods becomes imperative. Formal specification is playing a major role in the development of formal verification. As in other areas, such as software programs or data communication protocols, the specification of digital hardware components must deal with objects which are concurrent in nature. Petri nets, process algebras, and temporal logic are formalisms suitable to model concurrency and communication [10].

Digital asynchronous design, which does not assume a global clock to implement sequencing and communication, is attracting considerable interest, especially in the area of delay-insensitive circuits. The abstraction of timing from sequence in delay-insensitive circuits makes them ideal candidates to specification using the above formalisms. Not surprisingly this has been recently an active research area.

In the Petri net thread, signal transition graphs have been used to specify and synthesize delay-insensitive circuits [2]. Independently action graphs, a generalization of STG's, have been applied to the design of delay-insensitive bus arbitration interfaces [4].

Composition is an important property exploited in the work done on hardware specification, verification, and synthesis using process algebras. A CSP-like program describing delay-insensitive digital systems is synthesized by applying a "correct by construction" procedure in [7]. A similar approach based on trace theory is discussed in [3]. Formal verification of delay-insensitive circuits using the $\mu$-calculus on a CCS specification is reported in [6].

Temporal logic is more effective in capturing the correctness of a system behaviour such as liveness and safety properties. The use of temporal logic in digital hardware specification and verification is further discussed in [1].

## 3 Protocol specification: an example

Protocol descriptions of hardware components are often offered by the manufacturer in the form of timing diagrams. STG's have been used to design and implement delay-insensitive bus arbitration interfaces for microprocessor-based systems. The set of primitive implementation blocks includes two-state asynchronous state machines, event detectors, mutual-exclusion blocks and connectors [5].

Figure 1a shows the timing diagram of a fully-interlocked handshake bus arbitration protocol. A potential master requests the bus to the arbiter by setting the value of its REQ signal to true. When the input signal ACK is set to true by the arbiter, the requestor can take over the data transfer bus (DTB). At the completion of the transfer the master gives up the bus by resetting REQ and

waits until the arbiter has reset ACK to initiate another cycle.

Figure 1b shows the signal transition graph corresponding to the timing diagram. Nodes in the STG represent signal transitions and edges describe the precedence between transitions. Tokens indicate the initial state of the protocol. Output signal transitions are denoted by placing a bar on top of the name. Observe that it is not clear from the timing diagram if $r+$ must wait for transition $\underline{a}-$, which is formally specified in the graph.

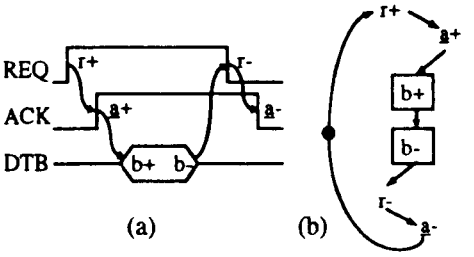



Figure 1. Fully-handshake bus arbitration protocol: (a) timing diagram; (b) signal transition graph.

The sequential aspect of transitions in the case of a fully-interlocked handshake protocol is evident from the presence of a unique cycle in its corresponding STG. Only one token is propagated throughout the edges in this graph. A more involved bus arbitration protocol used in the VMEBus standard [12] is shown in Figure 2.

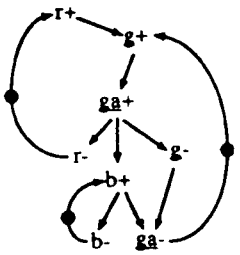



Figure 2. STG describing the VME bus arbitration protocol.

One can see that as more signals participate in a protocol, the corresponding STG becomes more complicated. The aim of this paper is to demonstrate that for the class of *valid* STG's it is possible to construct an equivalent CCS expression that can be written as a parallel composition of agents. The number of agents in the composition is linear with respect to the number of signals involved in the protocol.

For example the STG shown in Figure 2 can be transformed into the CCS expression $Prot \equiv (R \mid G \mid GA \mid B)$ where $R$, $G$, $GA$, and $B$ are CCS agents. The precedence between transitions is described within the agents using synchronization actions. Were another signal $s$ added to the protocol, the new CCS expression would be written simply as $Prot \equiv (R \mid G \mid GA \mid B \mid S)$.

# 4 Signal transition graphs

In this section signal transition graphs are introduced within the framework of digital hardware. Microprocessor components transfer information to one another in the form of signals via wires that connect their ports. Input ports accept incoming signals generated in output ports. A protocol enforces the correct transfer of information by defining the order (and timing) of elementary operations. The elementary operations in the protocol are called actions. Signal transitions are used to encode actions.

## 4.1 Ports and signal transitions

Ports are designated by unique names. Input port names are written $\underline{a}, \underline{b}, \underline{c}$, while output port names are written $a$, $b$, $c$. Each pair of input/output ports $(\underline{a}, a)$ is assumed to be connected through a wire. Transitions in the (binary) value of a port are denoted by suffixing an +/- symbol to the port name. For example $a+$ represents a positive transition[1] in the value of the output port $a$ which is connected to the input port $\underline{a}$.

The alphabet $T$ is the set of all signal transitions. Transitions $a+$ and $a-$ are called *opposite* transitions. When the type of transition is not important we shall use the notation $a!$ for $a+$ or $a-$. For each transition $a!$ its opposite transition is written $a!^*$. For a (binary) signal, it is true that $(a!^*)^* = a!$. The alphabet $T$ is partitioned into disjoint sets, each containing the transitions that a port can perform. The set of transitions that port $a$ can perform is denoted by $\lambda(a) = \{a+, a-\}$.

Wire delay is modelled by using two different transitions for the signal that propagates through the wire (see Figure 3). The two signal frames are the initiation of the transition in the output port and the reception of the transition in the input port. Informally a delay-insensitive circuit does not depend on gate and wire delays for its correct operation.

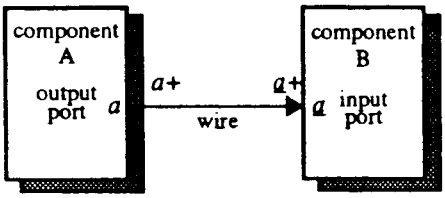



Figure 3. Wire delay model.

## 4.2 Signal transition graphs

In this section STG's are defined in terms of a precedence relation and its Petri net representation is presented.

A binary relation $P$ (for precedes) is defined on the set of transitions $T$ such that $a!\ P\ b!$ *iff* $a!$ must immediately precede $b!$ in the protocol. An STG [2] is a triple $\langle T, P, M_0 \rangle$ where $T$ is the set of transitions, $P$ is the precedence relation, $M_0 \subseteq P^\infty$ is the initial marking, where $P^\infty$ denotes a multiset of $P$. The pair $\langle T, P \rangle$ is a directed graph where $T$ is the set of nodes and $P$ is the set of edges. Thus there is an edge from $a!$ to $b!$ *iff* $a!\ P\ b!$.

A transition is enabled by a marking $M$ *iff* every incoming edge to the transition belongs to $M$. Every enabled transition can fire. After a transition fires, a new marking $M'$ is obtained from $M$ by removing the firing transition's incoming edges and adding the firing transition's outgoing edges. One token is assigned to each element of a marking $M$.

1. Without loss of generality we assume positive-logic or asserted-high signals.

A marked Petri net [9] is a quintuple $PN = \langle Tr, Pl, I, O, M_0 \rangle$ where $Tr$ is a non-empty set of transitions, $Pl$ is a non-empty set of places, $I: Pl^{\infty} \rightarrow Tr$ is the input function, $O: Tr \rightarrow Pl^{\infty}$ is the output function, and $M_0 \subseteq Pl^{\infty}$ is the initial marking. The underlying Petri net of a STG is the quintuple $\langle T, P, I, O, M_0 \rangle$ where

$$I = \{((a!,b!), b!) : a! \; P \; b!\}, \text{ and}$$

$$O = \{(a!, (a!,b!)) : a! \; P \; b!\}.$$

Not every STG is a valid one. An STG $\langle T, P, M_0 \rangle$ is said to be *valid* if it has the following properties:

i) If $a! \in T$ then $a!^* \in T$.

ii) There is at least one simple cycle[2] containing both transitions $a!$ and $a!^*$.

iii) In every simple cycle containing both transitions $a!$ and $a!^*$, the transitions alternate.

iv) There is one and only one token in each simple cycle of the graph.

Some invalid graphs are shown in Figure 4. The STG in Figure 4a lacks transition $a$-. The STG in Figure 4b does not have a cycle containing the alternation of transitions in $\lambda(a)$. The STG in Figure 4c has a simple cycle with two tokens. In the STG shown in Figure 4d there is no simple cycle with a token containing both $a$+ and $a$-.

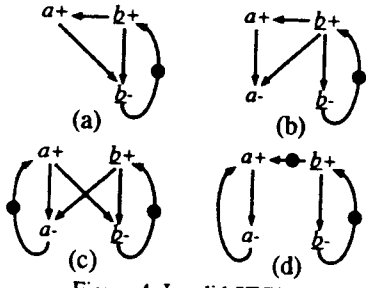



Figure 4. Invalid STG's.

Because every edge connects only a pair of transitions, the underlying Petri net of an STG as defined above belongs to the subclass of marked graphs.

### 4.3 Signal transition graph constructs

The graphical representation of STG's consists of nodes corresponding to the transitions of the protocol connected by edges indicating transition precedence. Four basic constructs in STG's are shown in Figure 5. Sequence is described by a single edge. A transition spawns other concurrent transitions in a fork. In a join a transition is enabled only when all its predecessors have already occurred. The fork/join is the most general construct.

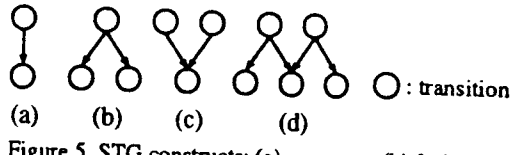



Figure 5. STG constructs: (a) sequence; (b) fork; (c) join; (d) general fork/join.

2. A (directed) path is a sequence of transitions. A (directed) cycle is a path in which the initial and the final transitions coincide. In a simple cycle, there are no duplicated transitions.

The basic constructs described above cannot model mutual exclusion. An extended model of STG's includes inhibitory/enabling edges to model preconditions to firing, and exclusive-OR join and fork constructs analogous to the fork and join constructs described above to model mutual exclusion (cf. [2]). In this paper we only consider the basic model of STG's presented in section 4.2.

### 4.4 STG operational semantics

In this section we give to STG's an operational semantics close in spirit to the operational semantics given to CCS by Milner (see section 5). The firing of an enabled transition $a!$ in marking $M$ is written $M \xrightarrow{a!} M'$ where $M'$ is the new marking after firing $a!$. The pair $(a!,M')$ is called an immediate $a!$-derivative of $M$. In general $M'$ is an $(a! \ldots b!)$-derivative (or just derivative) of $M$ if $M \xrightarrow{a!} \ldots \xrightarrow{b!} M'$.

A labelled transition system is the triple $\langle S, T, \{\rightarrow_t : t \in T\} \rangle$, where S is a set of states, $T$ is a set of transition labels, and $\rightarrow_t \subseteq S \times S$ is a transition relation for each $t \in T$. We define the meaning of an STG in terms of the labelled transition system $\langle SM, T, \{\xrightarrow{a!}\} \rangle$ where $SM$ is the set of reachable markings from $M_0$, $T$ is the set of transitions, and $\xrightarrow{a!} \subseteq SM \times SM$ such that $(M,M') \in \xrightarrow{a!}$ *iff* $M \xrightarrow{a!} M'$.

A derivation tree of the initial marking $M_0$ is a tree which collects all the derivatives of $M_0$. The nodes of the tree are the reachable markings from $M_0$. An edge of the tree joining $M$ and $M'$ is labelled with the firing transition $a!$ if $M \xrightarrow{a!} M'$. Derivation trees are usually infinite. A transition graph can be drawn from a derivation tree by collapsing identical markings, which have the same immediate derivatives, into a single node. Observe that concurrency is treated in our interpretation as the interleaving of sequences.

## 5 Calculus of Communicating Systems

Let $A$ be a set of action names and $\bar{A}$ be the set of co-names. Let $\tau$ denote the silent action. Finally let $Act \equiv A \cup \bar{A} \cup \{\tau\}$. The set $\varepsilon$ of CCS agent expressions[3] consists of all expressions generated by the following context-free production rules, where $E_1$, $E_2$ are already in $\varepsilon$:

| | | | |
|---|---|---|---|
| $E$ | ::= | $Nil$ | (Null process) |
| | \| | $X$ | (Process variable) |
| | \| | $a.E_1$ | (Action Prefix) |
| | \| | $E_1+E_2$ | (Summation) |
| | \| | $E_1 \mid E_2$ | (Composition) |
| | \| | $E_1 \backslash L$ | (Restriction) |
| | \| | $E_1[\varphi]$ | (Relabelling) |
| | \| | $\mu X.E_1$ | (Recursion) |

$L \subseteq Act$ is a set of labels and $\varphi : Act \rightarrow Act$ is a renaming function such that $\varphi(\tau) = \tau$ and $\varphi(\bar{u}) = \overline{\varphi(u)}$. We use the interleaving transitional semantics based on a labelled transition system as presented in [8]. We shall use frequently the following notation.

3. For an introduction to CCS see [11].

Definition.- The agent $\langle a_1, \ldots, a_n \rangle.P$ is defined recursively by

$$\langle \ldots, a_i, \ldots \rangle.P \equiv \sum_i a_i.(\langle \ldots, \ldots \rangle.P)$$

$$\langle a \rangle.P \equiv a.P$$

where $\sum_i P_i \equiv \ldots + P_i + \ldots$ .

Agent $\langle a_1, \ldots, a_n \rangle.P$ is capable of performing each action $a_1, \ldots, a_n$ once in any order and then it becomes agent $P$.

As discussed in section 3, protocols are cyclic in nature. One may want to differentiate the activation of a transition from one cycle to another. This can be done easily by introducing a cycle counter $t$ which parameterizes an agent or an action. For example, $A^t \equiv a^t.s^t.A^{t+1}$ describes the agent which during cycle $t$ can perform sequentially actions $a$ and $s$ before proceeding to the next cycle. The indexing mechanism will prove effective in the definition of the transitional rule for *AND* actions which is presented in section 6.1. If confusion does not arise, we shall not write the cycle index, although we shall assume it implicitly. Thus we write $A \equiv a.s.A$ instead of the previous *indexed* agent expression.

# 6 From signal transition graphs to CCS agent expressions

Desirable features which we look for in a decomposition procedure are amenability to *composition*, suitability to be *automated* and generation of expressions in *standard concurrent form*. To achieve the above characteristics in our decomposition procedure we take advantage of the parallelism of ports to yield a compositional description. Also we introduce a special type of synchronizing actions to express precedence between the transitions in the protocol.

The importance of a port is that it represents the finest grain in concurrency. In our framework, we try to express maximum concurrency, and thus we associate one agent to each port. For the sake of simplicity we shall utilize the return-to-zero assumption according to which at the end of a protocol cycle every port returns to its original value, although it is not difficult to generalize this scenario to agents having larger alphabets in which the symbols not necessarily alternate (i.e. multivalued logic ports) or even reach the same final value (i.e. non-return-to-zero schemes).

## 6.1 XOR and AND synchronization

CCS operational semantics provides synchronization between agents via composition and complementary actions $a$ and $\bar{a}$, possibly localized by restriction. For example consider the following three agents $A1 \equiv \bar{a}.A1'$, $A2 \equiv a.A2'$, and $A3 \equiv a.A3'$. The composite agent $A \equiv (A1 \mid A2 \mid A3) \setminus \{a\}$ has two immediate $\tau$-derivatives: $(A1' \mid A2' \mid A3) \setminus \{a\}$ and $(A1' \mid A2 \mid A3') \setminus \{a\}$. We call the behaviour of actions $a$, $\bar{a}$ of type *XOR* because only one pair of complementary actions is allowed to occur.

We define a new type of actions called *AND* actions such that all of them must occur simultaneously. For example if $s$, $\bar{s}$ are *AND* actions then agent $S \equiv (s.S1' \mid s.S2' \mid s.S3') \setminus \{s\}$ has only one $\tau$-derivative: $(S1' \mid S2' \mid S3') \setminus \{s\}$. At first glance it may seem unnecessary to assert directionality in *AND* actions. In section 6.2 we shall give an interpretation to output *AND* actions in the context of *sync* edges.

Thus in our framework actions are partitioned into two disjoint sets, $Act_{XOR}$ and $Act_{AND}$ such that $Act \equiv Act_{XOR} \cup Act_{AND} \cup \{\tau\}$. In the sequel we shall write actions $a, b, c \in Act_{XOR}$ and $s, t, u \in Act_{AND}$.

The interpretation of the operational semantics of CCS using a labelled transition system in the context of *XOR* and *AND* actions is extended as follows:

i) The transition rules defined in [8] apply only to *XOR* actions.

ii) For *AND* actions $s$, $\bar{s}$ we introduce the following *AND* synchronization rule:

$$\frac{\forall i, A_i \xrightarrow{\alpha_i} A_i' \qquad \alpha_i = s \text{ or } \bar{s}}{\Pi A_i \mid \Pi B_j \xrightarrow{\tau} \Pi A_i' \mid \Pi B_j}$$

with $\forall j, \Lambda(B_j) \cap \{s, \bar{s}\} = \phi$, where $\Lambda(\varepsilon)$ is the syntactic sort of expression $\varepsilon$ and $\Pi A_i$ denotes the parallel composition of agents $A_i$.

For example consider the agent $D \equiv A^0 \mid B^0 \mid C^0$, where $A^t \equiv a^{t'}.(\bar{s}^t.Nil \mid A^{t+1})$, $B^t \equiv s^t.b^{t'}.B^{t+1}$, and $C^t \equiv s^t.c^{t'}.C^{t+1}$. A fragment of the derivation tree describing the behaviour of agent $D$ is shown in Figure 6. After $a^{t'}$ occurs, agent $D$ can perform $a^{t'+1}$ or $\tau$, the latter corresponding to the firing of the *AND* actions $(s^t, \bar{s}^t)$. After $\tau$, both actions $b^{t'}$ and $c^{t'}$ are enabled. Observe that $s^t$ does not belong to the syntactic sort of $A^{t+1}$.

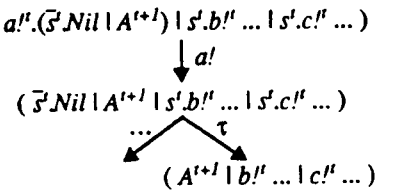



Figure 6. Partial derivation tree illustrating the operational semantics of *AND* actions.

Transitions of STG's can be expressed by CCS actions (i.e. $a+, a- , a^t \in Act_{XOR}$) and a new set of *AND* synchronization actions ($s, \bar{s} \in Act_{AND}$) is added to $Act$[4].

## 6.2 Decomposition procedure

We view each port as an independent process and thus we assign one agent to each port. Before stating our decomposition procedure, some definitions come to order.

Definition.- For every port $a$ there is a CCS agent $S(a)$.

Definition.- For each action $a^t$ in the graph the *fan-in* and *fan-out* sets of edges are defined respectively by

$$fi(a^t) \equiv \{(b^t, a^t) : b^t \; P \; a^t \wedge b^t \notin \lambda(a)\}, \text{ and}$$

$$fo(a^t) \equiv \{(a^t, b^t) : a^t \; P \; b^t \wedge b^t \notin \lambda(a)\}.$$

Often we use the shorthand $A$ to denote agent $S(a)$ which represents port $a$ in CCS. The fan-in and fan-out sets of an action $a^t$ contain the incoming edges to and the outgoing edges from $a^t$ respectively not including $(a^t, a^{t*})$ and $(a^{t*}, a^t)$. Both $fi(a^t)$ and $fo(a^t)$ may be empty.

4. *AND* actions can be simulated in standard CCS.

Definition.- For each non-empty $fi(a!)$ there is a unique *sync action name* $s_i$ associated to $fi(a!)$. Each action name $s_i$ generates the input and output *sync actions* $s_i, \bar{s}_i \in Act_{AND}$.

In the absence of sync actions, the behaviour of each agent $S(a)$ is a sequence of actions $a!.a!^*. \ldots$ *ad infinitum*. Sync actions are used to describe the precedence between the *XOR* actions in the protocol by restricting the initially free behaviour of each agent $S(a)$. Observe that each edge in the graph that connects two actions belonging to different ports is associated to a sync name. All edges sharing the same sync name $s_i$ are called generically $s_i$ sync edges.

Lemma 1.- The edge $(a!, b!)$ in an STG does not have a sync action name associated to it if $b! \in \lambda(a)$.

Proof. Both $fi(b!)$ and $fo(a!)$ do not contain $(a!, b!)$ if $b! \in \lambda(a)$.

The following definitions are used to determine the action $a!$ that is performed first by agent $S(a)$.

Definition.- For each cycle in a valid STG, the initial edge is the pair $(a!, b!) \in P$ such that it also belongs to the initial marking $M_0$.

Definition.- The *initial* action of an agent $S(a)$ is the first action $a! \in \lambda(a)$ to appear in a simple cycle containing both $a!$ and $a!^*$ starting from the initial edge.

Lemma 2.- There is a unique initial action for each port $a$.

Proof. We have to show that if there are several simple cycles containing both $a!$ and $a!^*$ all the respective tokens are close to only one of the actions. Suppose that there are two simple cycles in the valid STG which include both $a!$ and $a!^*$ such that the token in one of them is closer to $a!$ while the token in the other cycle is closer to $a!^*$. Then one can construct another simple cycle consisting of the two partial paths with a token on them of each of the cycles. This simple cycle would contain two tokens, which is a contradiction.

Definition.- The *set of initially enabled edges* of an STG is:

$$IEE \equiv \{(a!, b!) : b! \notin \lambda(a) \wedge (a!, b!) \in M_0\}$$

Definition.- The set of initially enabled edges w.r.t. an action $a!$ is:

$$IE(a!) \equiv IEE \cap fo(a!)$$

The decomposition procedure recursively writes agent $S(a)$ starting from its initial action by annotating sync edges to actions belonging to other agents $S(b)$. Input and output sync actions are used to represent such edges. We write $s_i$ to identify the sync actions corresponding to the head of a sync edge; likewise input sync actions $\bar{s}_i$ represent the tail of sync edges.

**Decomposition procedure**

Given a valid $STG = \langle T, P, M_0 \rangle$ do:

1. For each port $a$ and corresponding agent $S(a)$ identify the initial action and call it $a!$.

2. For each port $a$ and initial action $a!$ define agent $S1\ (a!)$ recursively in two steps as follows:

- Phase 1 (Prefix):

If $fi(a!)$ is empty then $S1(a!) \equiv a!.S2(a!)$,

else $S1(a!) \equiv s_i.a!.S2(a!)$

where $s_i$ is the sync action name associated to $fi(a!)$.

- Phase 2 (Suffix):

If $fo(a!)$ is empty then $S2(a!) \equiv S1(a!^*)$,

else $S2(a!) \equiv \bar{s}_{i1}.Nil \mid \bar{s}_{i2}.Nil \mid \ldots \mid S1(a!^*)$

where $\bar{s}_{ij}$ is the sync action name associated to $fi(b!)$ for each $b!$ such that $(a!, b!) \in fo(a!)$.

3. If $IE(a!^*)$ is empty then write $S(a) \equiv S1(a!)$,

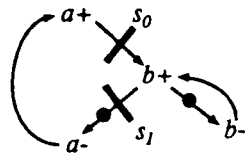
else write $S(a) \equiv \ldots \mid s_{\alpha i}.Nil \mid \ldots \mid S1(a!)$

where there is a term $s_{\alpha i}.Nil$ for each sync action name $s_{\alpha i}$ associated to $fi(b!)$ such that $(a!^*, b!) \in IE(a!^*)$.

4. The CCS agent expression *Prot* that corresponds to the STG is the composition of all agents $S(a)$ written as follows:

$$Prot \equiv (\ S(a) \mid \ldots\ )$$

□




$$A \equiv s_1.a-.a+.(\bar{s}_0.Nil \mid A)$$
$$B \equiv b-.s_0.b+.(\bar{s}_1.Nil \mid B) \quad \text{(before step 3)}$$
$$Prot \equiv (\ A \mid B\ )$$

$$B \equiv \bar{s}_1.Nil \mid S1(b-) \quad \text{(after step 3)}$$
$$S1(b-) \equiv b-.s_0.b+.(\bar{s}_1.Nil \mid S1(b-))$$

Figure 7. Enabling the initial marking in CCS agents.

The recursion in step 2 of the decomposition procedure terminates when expression $S2(a!^*)$ is expanded in phase 2. At that moment $S1(a!)$ is recursively defined by a CCS expression $\varepsilon$ which only contains $S1(a!)$ as free variable. Step 3 ensures that the initial marking is initially enabled, otherwise the resulting agent could deadlock, as shown in the example shown in Figure 7. Agent *Prot* cannot execute any action with $B \equiv S1(b!)$ defined after step 2 because $A$ is guarded by action $s_1$ which is blocked in $B$. The correct expression for agent $B$ enables the sync edge $s_1$ of the initial marking (see Figure 7).

Definition.- A *pure* fork is represented by a set of edges $\{(a!, b_i!)$ such that each $fi(b_i!)$ has cardinality one and $fo(a!) = \bigcup fi(b_i!)$.

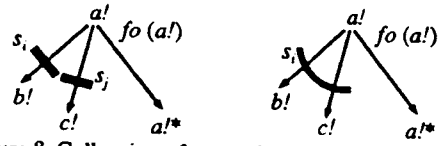
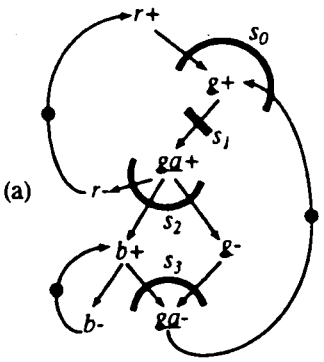



Figure 8. Collapsing of sync edges in a pure fork.

In the case of a pure fork it is possible to associate a unique sync action name to $fo(a!)$ which replaces the original sync actions in the fork[5], one for each $fi(b_i!)$, as shown in Figure 8.

The interleaving transitional semantics of a valid STG and its corresponding CCS agent expression are *observation-equivalent* [8]. This is important because it means that the decomposition procedure preserves the external behavioral properties (i.e. liveness and safety) of the original graph. Therefore it is possible to study the behaviour of the protocol in the CCS domain.

5. Also observe that $\langle s_i, s_j \rangle.Nil = s_i.Nil \mid s_j.Nil$.




(b) $R \equiv r+.(\bar{s}_0.Nil \mid s_2.r-. R)$
$G \equiv s_0.g+.(\bar{s}_1.Nil \mid s_2.g-. (\bar{s}_3.Nil \mid G))$
$GA \equiv \bar{s}_0.Nil \mid SI(ga+)$
$SI(ga+) \equiv s_1.ga+.(\bar{s}_2.Nil \mid s_3.ga-.(\bar{s}_0.Nil \mid SI(ga+))$
$B \equiv s_2.b+. (\bar{s}_3.Nil \mid b-.B)$
$Prot \equiv (R \mid G \mid GA \mid B)$

(c) $R \equiv r+.\bar{s}_0. s_2.r-. R$
$G \equiv s_0.g+.\bar{s}_1. s_2.g-. \bar{s}_3. G$
$GA \equiv \bar{s}_0. s_1.ga+.\bar{s}_2. s_3.ga-. GA$
$B \equiv s_2.b+.(\bar{s}_3, b-).B$
$Prot \equiv (R \mid G \mid GA \mid B)$

Figure 9. Bus arbitration protocol: (a) STG; (b) corresponding CCS expression; (c) reduced CCS agent.

The decomposition procedure described in this section produces agent expressions *Prot* which are not in standard concurrent form (SCF), i.e. *Prot* is not written as the composition of *purely* sequential agents $S(a)$. In some cases it is possible to reduce the CCS expressions further to SCF in the presence of handshake edges[6] between two agents. Figure 9a shows the STG of the bus arbitration protocol defined in the VMEbus standard [12]. The CCS agent in Figure 9b is obtained after applying the decomposition procedure. Due to handshake edges, particularly between agents *G* and *GA*, the CCS agent can be reduced to the one shown in Figure 9c.

# 7 Final remarks

## 7.1 Results

We have used the Edinburgh Concurrency Workbench (CWB version 6.0) to test our ideas. We have been able to write CCS expressions of various protocols. CWB has allowed us to verify some properties of the protocols, such as the absence of deadlock. We have also experimented with adding extra sync edges and ports to the protocols. For instance we have corroborated empirically that the more handshake edges an STG has, the more restricted the behaviour of the protocol becomes. Thus by being able to represent protocols in CCS we have also gained access to a set of verification tools.

## 7.2 Summary

In microprocessor-based systems the components' external behaviour can be modelled by their interfacing protocols. Signal transition graphs, which can be derived from timing diagram specifications, are capable of expressing the interaction that occur between the elementary operations in a protocol. However STG's are not very amenable to composition. Process algebras such as CCS overcome this problem by using a parallel composition operator that allows the designer to combine modules to build up more complicated systems.

In this paper we have suggested a decomposition procedure that converts STG's into CCS agent expressions. Not only does this procedure exploit the natural concurrency among the signals involved in protocols but also does express the handshake edges between agents more explicitly. A new type of actions called *AND* actions is introduced to describe multi-way synchronizations which occur in general join/forks constructs. *AND* actions can be simulated using standard CCS actions so that tools such as the CWB can be used to analyze our protocol agents.

## 7.3 Future work

The main contribution of this paper is perhaps the demonstration of how to transform valid STG's into a composition of CCS agents. The application of the general decomposition procedure results in agent expressions which are not written in standard concurrent form (SCF). We are currently investigating the conditions under which it is possible to reduce further an agent *Prot* to SCF. The next step we would like to pursue is the development of interface design methodologies that can deal with incompatible protocols. A complete system consisting of two components would be of the form $System \equiv Prot1 \mid Interface \mid Prot2$ such that the interface preserves the partial order defined on the transitions by the protocols.

6. A pair of edges $(a!, b!)$ and $(b!, a!^*)$ between the actions of two agents $S(a)$ and $S(b)$.

# References

[1] G. V. Bochmann, "Hardware specification with temporal logic: An example," *IEEE Trans. on Computers*, vol. C-31, pp. 223–231, Mar. 1982.
[2] T.-A. Chu, "On the models for designing VLSI asynchronous digital systems," *INTEGRATION, the VLSI journal*, no. 4, pp. 99–113, 1986.
[3] J. C. Ebergen, *Translating programs into delay-insensitive circuits*, No. 56 in CWI Tract, Centrum voor Wiskumde en Informatica, 1987.
[4] M. A. Escalante, "Bus arbitration modelling and design in DAME: An expert microprocessor-based-systems designer," Master's thesis, University of Victoria, 1991.
[5] M. A. Escalante et. al., "The implementor subsystem in DAME: Using OASIS to complete the design automation of microprocessor-based systems," in *Canadian Conference on VLSI*, pp. 139–146, Oct. 1992.
[6] Y. Liu, "Reasoning about asynchronous designs in CCS," Tech. Rep. No. 92/492/30, University of Calgary, 1992.
[7] A. J. Martin, "Synthesis of asynchronous VLSI circuits," in *Formal methods for VLSI design*, North-Holland, 1990.
[8] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.
[9] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. of the IEEE*, vol. 77, pp. 541–580, Apr. 1989.
[10] E.-R. Olderog, "Nets, terms and formulas: Three views of concurrent processes and their relationship," Tech. Report, Universität Oldenburg, FB Informatik, 1989.
[11] D. Walker, "Introduction to a calculus of communicating systems," Tech. report, University of Edinburgh, 1987.
[12] *IEEE Standard for a Versatile Backplane Bus: VMEBus*. IEEE Press, 1988.