Designing Asynchronous Circuits from Behavioural Specifications with Internal Conflicts *

J. Cortadella
Dept. of Computer Architecture
Univ. Politècnica de Catalunya
08071 Barcelona, Spain

L. Lavagno
Dipartimento di Elettronica
Politecnico di Torino
10129 Torino, Italy

P. Vanbekbergen
Synopsys Inc.
Mountain View
CA 94043-4033, USA

A. Yakovlev
Dept. of Computing Science
Univ. of Newcastle upon Tyne
NE1 7RU, UK

Abstract

The paper presents a systematic method for synthesizing asynchronous circuits from event-based specifications with conflicts on output signals. It describes a set of semantic-preserving transformations performed at the Petri net level, which introduce auxiliary signal transitions implemented by internally analogue components, Mutual Exclusion (ME) elements. The logic for primary outputs can therefore be realized free from hazards and external meta-stability. The technique draws upon the use of standard logic components and two-input MEs, available in a typical design library.

1 Introduction

Self-timed circuit design has been mainly aimed at the modelling and implementation of behaviour that is either speed-independent or delay-insensitive. Today, there is more interest in mixed systems, where modules may have local clocks and interact asynchronously: hence the demand for corresponding models and synthesis methods. Designers often want to represent the desired behaviour as a combination of causality and ordering (timing) constraints. Such a behaviour may not satisfy some of the initial requirements imposed by the existing synthesis methods and tools. For instance, the asynchronous synthesis subsystem of SIS [9] requires that a Signal Transition Graph (STG), used to define the behaviour of a circuit, be output-persistent (only input signal transitions can be disabled). Moreover, the Forcage synthesis tool, described in [3], puts even stronger requirements on the specification. It must be totally free from choice, allowing only concurrency and two types of causality (AND and OR).

A simple example, the behaviour of a transparent latch considered in the next section, shows that both such restrictions are a serious limiting factor for the *practical* use of asynchronous design techniques.

Our aim is to be able to derive such and similar designs *mechanically* (ultimately, using a CAD tool) from their formal behavioural specifications. This should ideally be done within the already existing framework of synthesis methods developed for STGs, without requiring any custom design. It is well-known that synchronization of causaly unrelated events and signal disabling unavoidably leads to *hazards* in combinational logic and *meta-stability* in sequential logic. Hence the implementation of output disabling must use some circuitry (partly analogue) to resolve conflicts at the logical level. Such circuits should preferably be built using some standard library components, e.g., a two-input mutual exclusion (further called *mutex* or ME) element with an internal meta-stability detector [8]. Due to design methodology requirements, it is highly unlikely to expect designers to build their own special transistor interconnections, which would require putting effort extensive into analogue modelling and simulation.

Thus, our goal is to be able to start from a specification of a most general class of behaviours describable by Petri nets and STGs (obviously bounded, to be implementable with finite memory), and then perform the following procedure:

1. determine a set of output signal transitions that are *not persistent* (can be disabled by input or output signal transitions);
2. insert an appropriate set of mutex elements (with internal analogue meta-stability detector), making semantic-preserving transformations at the specification level (i.e. STG or its State Graph);

*For J. Cortadella this work was supported by ACiD-WG (Esprit 7225) and CICYT TIC 91-1036. The research of L. Lavagno was partially supported by MURST under 40% project "VLSI Architectures". For A.Yakovlev this work was partly done during his visit to Torino in March-April 1994, sponsored by the British Council and CNR (Italy).

0-8186-6210-7/94 $4.00

3. factor them out of the model, making their outputs to be additional inputs to the circuit, which should now be *output-persistent*;

4. synthesise the "logical part" of the circuit by the present day STG-based methods and tools;

5. combine the "mutex part" with the "logical part" by interconnecting the MEs and the gate network through the set of request-acknowledgement handshakes (if the standard n-input n-output MEs are used).

In this paper, we make our first attempt on this challenging path. We present some techniques for:

- modelling the behaviour with conflicts and output non-persistency;
- transforming the initial specification into one with mutex element actions;
- implementing the specification using standard MEs and logic synthesis techniques.

The modelling and transformation are both shown at the STG level, because it is more suitable for descriptive purposes. An algorithmic implementation, though, may also be done at the SG level, which has the advantage of being canonical, albeit often exponentially larger than a corresponding STG.

2 A Simple Example: the Transparent Latch

Specification. We first provide an intuitive description for a transparent latch (henceforth, simply referred to as a "latch" unless it creates confusion) using timing diagrams and plain English. The latch is a device with two inputs D and Ck, and one output Q. We assume throughout this discussion that D is temporally unrelated to Ck, but otherwise hazard-free and locked in handshake with Q. This means that D cannot change value twice without waiting for Q to change in between. This is a reasonable assumption, e.g., for the synchronizer for an interrupt input to a microprocessor. If D could change multiple times without Q changing in between, then we would need a more complex STG and timing assumptions, that would unnecessarily complicate the example. We will later need some further timing constraints, e.g., relating the clock frequency to the meta-stability resolution time.

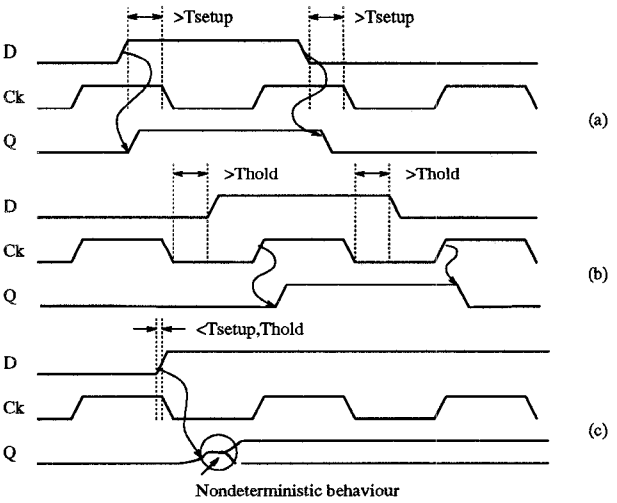

Figure 1: Timing diagrams for transparent latch

The timing diagrams shown in Figure 1 describe the three different situations that may occur when the latch is initially set at $Q = 0$ and subject to signal transitions at its inputs. The inputs are initially both at logical 0, so that Q is stable. The first case, shown in Figure 1(a), corresponds to the following sequence of signal transitions: $Ck+, D+, Q+, Ck-, Ck+, D-, Q-, \ldots$, where $x+$ $(x-)$ denotes the positive (negative) edge of signal x. Here we assume that both $D+$ and $D-$ occur *well ahead* of the falling edge of Ck. The second case, shown in Figure 1(b), corresponds to the following sequence of signal transitions: $Ck+, Ck-, D+, Ck+, Q+, Ck-, D-, Ck+, Q-, \ldots$. The assumption here is that transitions $D+$ and $D-$ occur *well after* the falling edge of Ck. The third case, shown in Figure 1(c), in which D changes *very close* to $Ck-$, is nondeterministic. We cannot precisely say which state Q will assume. If we specified that Q had always to change to logical 1, it would be an unrealistic requirement. It is known from theory and practice of building synchronisers (e.g., [6]) that it is impossible to construct such a device that would behave in determinate way when the asynchronous input edge is close to the edge of the strobe signal.

The most difficult case to deal with, both in terms of its formal modelling and circuit implementation, is certainly the third one. Although the designer may clearly understand the impossibility of building a determinate implementation, and allow for two potential alternatives in the subsequent action of the device and its environment, there is an additional issue to be looked at: meta-stability.

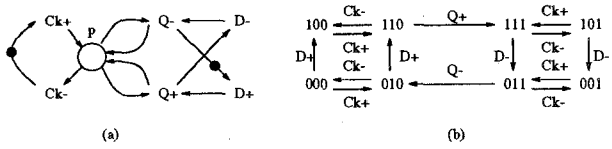

Figure 2: STG description of the transparent latch (a) and its state graph (b)

Incorrect implementation of the latch. We can be very naive and try to implement the above specification using for example a formal method based on STGs (see Section 3 for a more precise definition of the STG syntax and semantics). The STG which completely satisfies the informal specification given above in terms of timing diagrams is shown in Figure 2(a). It contains a transition vertex for each signal transition and its arcs stand for causal relationship between transitions. Compared to the standard Petri net notation, we have replaced most place vertices, together with their incoming and outgoing arcs, with a single arc (from the cause to the effect). The only explicit place, denoted by p, is needed because we have several transitions incident to it. The placement of tokens on the arcs as shown in the figure corresponds to the initial state of the latch, $D = Q = Ck = 0$.

If we now consider the behaviour of this STG, which can be represented by the corresponding State Graph (SG), we will see that all the cases of Figure 1 occur in it. The SG is shown in Figure 2(b). In this SG, vertices stand for the markings of the Petri net underlying our STG. Furthermore, these markings are labelled with binary codes, which correspond to vectors of values of the modelled circuit signals.

It is easy to traverse the SG to find the specified cases. The first two cases are simply the feasible firing sequences in the STG. In analysing them we should bear in mind that the firing of either transition $Q+$ or transition $Q-$ does not disable transition $Ck-$ since we have two-way arcs connecting those two transitions to place p. At the same time the arcs connecting the transitions of D with those of Q depict the assumed causality (when the switching of D causes the transitions of Q) and ordering (when input D cannot change before the corresponding change of Q has arrived) relations. As for the third case, consider the situation when, after firing $Ck+$, the STG reaches the marking in which the place p contains a token. This marking obviously enables transitions $Ck+$ and $D+$. Now, assume that $D+$ fires *very close to* $Ck-$*, but before it.* In this case, the net reaches the marking (state 110) under which both $Ck-$ and $Q+$ are enabled. Since Q is an output signal, transition $Q+$ cannot fire instantaneously. The associated delay is at least the delay of a gate whose output produces Q. Thus we assume that the STG transitions labelled with signal Q have a non-zero enabling time. This time may however be longer than the time difference between $D+$ and $Ck-$. Hence, there is a non-zero probability that transition $Q+$ may not manage to fire and thus can be disabled by $Ck-$. A similar situation happens when $D-$ arrives close enough before $Ck-$. The SG of the STG thus correctly represent the nondeterminism in the behavioural specification.

In order to build a circuit implementing this model, let us apply the same procedure for logical implementation of an SG as described, e.g., in [2]. For each state label (the order of signals is D, Ck and Q) we write down the implied (next-state) value of the output signal Q: $000 \rightarrow 0$, $100 \rightarrow 0$, $010 \rightarrow 0$, $110 \rightarrow 1$, $001 \rightarrow 1$, $101 \rightarrow 1$, $011 \rightarrow 0$, $111 \rightarrow 1$. This produces the standard logic function for Q:

$$Q = D \cdot Ck + (D + \overline{Ck}) \cdot Q$$

It is well-known that this implementation works only assuming that transitions of D occur far enough from the falling edge of Ck (setup and hold times). In the asynchronous case, though, this is not satisfactory, because we cannot safely make such an assumption. The danger here is that the latch may reach a meta-stable state, lasting an unbounded amount of time, in which its output has a value which is neither logical 0 nor logical 1. This intermediate value is even more dangerous because it can be interpreted differently by different logical gates driven by the latch.

In order to ensure safe and correct operation while complying with the initial STG model, we have to alter our synthesis procedure so as to avoid non-persistency of output signal transitions. The approach that we pursue in the rest of the paper is based on the explicit use of ME elements, to *protect* the transitions of outputs which are non-persistent in the initial specification. In this approach we assume that a ME is implemented *safely* and generates no hazards at its outputs if the input changes satisfy the basic handshake protocols on the request/acknowledgement terminals of the ME element.

We are not eliminating meta-stability (this cannot be done), but we are limiting it to some specific circuit component, the ME, which has a well-defined behaviour and produces valid logic outputs even in the meta-stable state.

3 Signal Transition Graphs

Signal Transition Graphs and Petri nets The *Signal Transition Graph* (STG) was independently introduced by [2] and [7] as a specification formalism for asynchronous sequential circuits. An STG is an interpreted Petri net, and as such it is capable to explicitly capture *causality, concurrency* and *choice*.

A Petri net is a triple $N =< P, T, F >$, where P is a set of *places*, T is a set of *transitions* and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. A place $p \in P$ is a *predecessor* of a transition $t \in T$, and t is a *successor* of p, if $(p, t) \in F$. Conversely, a transition $t \in T$ is a predecessor of a place $p \in P$, and p is a successor of t, if $(t, p) \in F$.

An STG is an *interpreted Petri net*: transitions of the net are *interpreted* as value changes on input/output signals of the specified circuit. *Positive* transitions (labeled with a "+") represent $0 \rightarrow 1$ changes, *negative* transitions (labeled with a "-") represent $1 \rightarrow 0$ changes. *Input transitions* are those that occur on input signals of the circuit, *output transitions* are those that occur on its output signals.

The conventional graphical representation of an STG (slightly different from the Petri net convention) is a directed graph, where transitions are simply identified by their name, places are denoted by circles, and directed edges represent elements of the flow relation. Places with only one predecessor and one successor are usually omitted. Directed edges whose successor is a transition represent sequencing constraints, either on the circuit to be synthesized (if their successor is an output transition), or on the environment (if their successor is an input transition). They specify what set of transitions causes each transition.

A *token marking* of a Petri net is a non-negative integer labeling of its places. A transition is *enabled* (i.e. the corresponding event can happen in the circuit) whenever all its predecessor places are marked with at least one token.

An enabled transition may *fire*. This means that the corresponding signal changes value in the circuit. When it fires, a token is removed from every predecessor place, and a token is added to every successor place.

If a place marked with only one token has more than one enabled successor transition, then only one of them may non-deterministically fire. The other transitions are *disabled* by its firing.

A marking M'' is *reachable* from another marking M' if there exists a sequence of enabled transition firings that produces M'' starting from M'.

A marking M' is *live* if for all markings M'' reachable from M', every transition can be enabled through some sequence of firings from M''. A marked net is live if its initial marking is live.

A marking M' is *bounded* if the number of tokens that any place can be holding after any sequence of firings from M' is bounded. A marking M' is *safe* (sometimes referred to as 1-bounded) if it is 1-bounded. A marked net is bounded (resp. safe) if its initial marking is bounded (resp. safe).

A transition t is called *persistent* if there exist no such reachable marking M under which t can be disabled by the firing of some other transition t'. The Petri net and its corresponding STG is called persistent if all its transitions are persistent. An STG is called *output-persistent* if all of its transitions labeled with output signals are persistent.

Note that our definition of output-persistency is rather conservative (cf., [12]). We do not allow to exploit the interleaving semantics of concurrency between labeled actions at the STG level. For example, let t_1 be a transition labeled with an output change t^* and disabled by t_2 under a reachable marking M. Let also exist another transition t_3 which is labelled with the same label and enabled after the firing of t_2. Thus, t_3 "takes over" the *signal transition* t^* and thus *preserves* its enabling. We classify this case as non-output-persistent.

This restriction is important for our approach to model transformation, as will be described in the next section. This approach is entirely net-based. ¿From practical reasons this limitation is not crucial since the descriptive power of Petri nets allows keeping a one-to-one relationship between signal transition events and Petri net transitions. Furthermore, it disciplines the designer in an *optimal* utilisation of net transitions.

State Graphs The reachability graph of a Petri net is a directed graph where each node corresponds to a marking and an edge joins a pair of markings M', M'' if there exists a transition t^* that firing from M' produces M'' (the transition labels the edge).

The State Graph (SG) ([2]) of an STG is the reachability graph of the underlying net where each node (henceforth called *state*) is labeled with a vector v of signal values. This node labeling must be *consistent with the* SG *edge labeling*, in other words for each edge $s' \rightarrow s''$, for each signal t:

1. if the edge is labeled t^+ then signal t must be 0 in v' and 1 in v''
2. if the edge is labeled t^- then signal t must be 1

in v' and 0 in v''

3. otherwise signal t must have the same value in both v' and v''.

Another important property of the SG is its semimodularity with respect to output signals. An SG is called *semimodular* if for each output signal t and any reachable state s in which t is enabled (there is an edge labelled with t^* leading from s to some s') t remains enabled in any other state reachable from s through the firing of some other signal transition enabled in s.

The STG specification, generating the consistent and semimodular SG, can be implemented as a logic circuit as described in [2, 7]. One combinational logic implements the *next-state* function of each output signal, mapping each SG label into the corresponding *implied value* for each output signal. The implied value of an output signal t in an SG state s is defined as:

- the value of t in the label of s if no transition of t is enabled in s.
- the complement of that value otherwise.

Chu showed ([2]) that the next-state function is well-defined (i.e. it has only one value for each point in the domain) if and only if for each pair (s', s'') of SG states that have the same label, *the same set of output signal transitions is enabled* in both markings corresponding to s' and s''. If the SG has this characteristic, then we say that the STG from which the SG was derived has the Complete State Coding property (CSC).

It is clear that the SG produced by an STG can be non-semimodular for some output t only if the STG is not persistent with respect to some transition labelled with t. One cannot derive the next-state function to produce a hazard-free logic for t, as was shown in Section 2. In the following section, we show how this problem can be resolved at the STG level, be means of adding special-purpose transitions, called *semaphore actions*. These transitions are aimed at *isolating* all the output non-persistency from the logic part of the implementation. They can be implemented by standard ME elements.

4 Conflict places and Petri net transformations

Similarly to many situations in concurrent programming, the problem of guaranteeing a persistent behaviour in a Petri net can be solved by forcing a mutually exclusive access to conflicting places. Thus, a place becomes a *critical section* of the system, which has several processes (producers) adding tokens to the place and several processes (consumers) taking tokens from the place [1].

The problem of non-persistency arises when more than m consumers (transitions) are simultaneously enabled by a place, while the place has only n ($n < m$) tokens. Let us assume these m transitions signify changes in output signals. Since signal changes are not instantaneous in digital circuits, all m transitions may start their process of changing a signal. However only n of them (the fastest to complete) will be able to fire and, therefore, the other $m - n$ transitions will have to be cancelled. This cancellation may produce undesirable effects on the circuit behaviour: *hazards*, manifested by glitches generated by the cancellation of the slowest signals, and *meta-stability*, perceived when the circuit is not able to decide the completion order of transitions.

These circuit malfunctions will be avoided if at most n transitions are allowed to be enabled when the conflicting place has n tokens.

4.1 The producer-consumer problem

The problem to be solved in our domain of Petri nets is a simplified version of the consumer-producer problem in which

- messages (tokens) have no explicit meaning, i.e. they are not used for communication but only for synchronization, and
- the buffer (place) used to store messages is large enough to handle the maximum number of messages that can be produced (this number must be bounded for the specification to be implementable as a logic circuit).

Since producers will never find the buffer full, there is no need for them to have a mutually exclusive access with the other processes of the system. Only consumers, competing for tokens in the conflicting place, must be controlled.

A classical solution to control the concurrent access to a critical section is the use of *semaphores*. The simplest implementation of a semaphore consists of a 1-bit variable (`sem`), and two atomic operations:

- `wait (sem)`, that halts the process until the value of `sem` is 1. After its execution the value of `sem` becomes 0. This operation grants the access to the critical section.

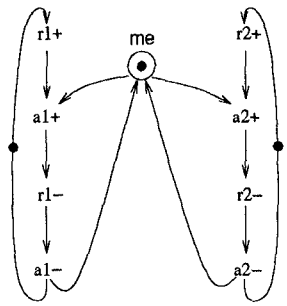

Figure 3: Behavioural description of a 2-input mutex

- **signal (sem)**, that writes a 1 in **sem**, allowing other processes to access the critical section.

Thus, a process willing to access a critical section c, protected by the semaphore **sem** must execute the following sequence of operations:

... **wait(sem)**; access(c); **signal(sem)**; ...

An ME is the hardware implementation of a semaphore. Figure 3 depicts the behaviour of a 2-input ME in which place **me** plays the role of the 1-bit variable of the semaphore. The operation **wait (sem)** is implemented by the handshaking pair $R+ \rightarrow A+$, whereas **signal (sem)** is implemented by $R- \rightarrow A-$. In general the behaviour of a semaphore controlling n concurrent processes can be implemented by an n-input ME, which can either be a primitive gate in a library, or be constructed from cascaded 2-input MEs.

4.2 STG-level transformations

The next step in our methodology to solve the problem of output non-persistence is to re-describe the behaviour of the circuit by means of semantic-preserving transformations from the original description. This basically involves interleaving conflicting transitions with the primitives that guarantee mutual exclusion. The resulting Petri net has the property that *no more than one of the original conflicting transitions can be enabled simultaneously*. The problem of output non-persistence is thus moved out to the outputs of the ME (which are now inputs of the circuit being synthesized) and converted into a situation of input non-persistence.

Figure 4 shows how the primitives **wait** and **signal** can be described at the STG level. Each primitive has two parameters: the number of the ME channel to which requests are done (i) and the ME used for that particular critical section (in general, several critical

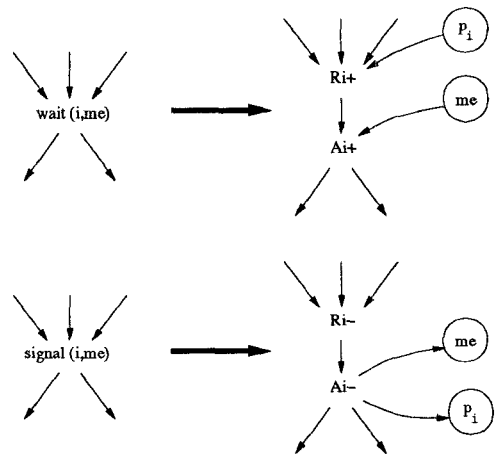

Figure 4: STG-level description of the semaphore primitives

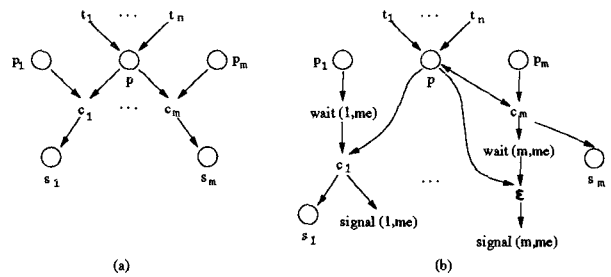

Figure 5: (a) Original Petri net with a conflict place. (b) Transformed Petri net

sections can be used in the same description). In order for each channel i to force a 4-phase handshake between signals R_i and A_i, a 1-safe place p_i is inserted in such a way that a token in p_i indicates that the handshake has completed and a new handshake can be initiated.

Let us assume that we have a conflict place p such as the one depicted in Figure 5.(a), with n producer transitions $(t_1, \ldots, t_n)$ and m consumer transitions $(c_1, \ldots, c_m)$. In the transformed description (Figure 5.(b)) no changes must be included for the producers, but mutually exclusive access must be guaranteed for the consumers[1].

A distinction between input and non-input transitions must be done when inserting primitives for mutual exclusion. Specifically the circuit must not put additional constraints on the behaviour of the environment. This is illustrated by the different transformations applied to c_1 and c_m (assumed to be non-input

[1]Notice that a consumer can also be a producer at the same time (self-loop transition)

and input transitions respectively).

More precisely, the enabling conditions of *output* signal transitions, that are under direct control, can be conditioned to follow the `wait` handshake. On the other hand, the enabling conditions of *input* signal transitions cannot be changed, because the environment cannot be constrained in general. This means that the `wait` handshake corresponding to an input transition can be initiated only *after* the corresponding transition has fired. In consequence, the disabling of the output transitions can occur only after this `wait`. In Figure 5.(b) this is represented by an unlabeled, internal transition ϵ that removes the token from p, and hence prevents c_1 from potentially firing, only when we know for sure that c_1 has lost the arbitration, and it is not currently enabled. In practice, this internal transition can be the A_m^+ transition of the corresponding `wait`

Another interesting point to note is that in general we cannot control the marking of successor places of input transitions (e.g., s_m). This means that, in principle, we cannot ensure that the resulting STG is *safe*, or even bounded. Moreover `signal`s do not have a successor transition, implying a potentially non-strongly-connected STG. The first problem is the most serious, because it forces the introduction of *timing constraints* to ensure resolution of arbitration (and potentially of meta-stability) before places p and p_m can be safely marked again. Otherwise, the resulting circuit could not be built in a hazard-free manner.

The second problem can also be solved with timing constraints, or, in the case of 1-safe conflict places, with simple transformations at the STG-level (see Figure 6). Once the token of the conflict place has been consumed, no other "consuming" transition will be enabled until a "producing" transition is fired. For this reason, there is no need to free the critical section after consuming the token and, thus, `signal` operations can be delayed until after having put the token into the place. The appropriate sequencing between operations of the same ME channel must be guaranteed by the net (denoted by dashed arcs in the figure).

4.3 ME channel assignment

Two transitions, t_1 and t_2, are called *in conflict* with respect to place p (denoted by $t_1 C_p t_2$), if $t_1, t_2 \in p^\bullet$ and there is a marking under which they are both enabled and the firing of one of them disables the other one. Note that this conflict may be asymmetric, i.e. one transition disables the other while not vice versa. Since $t_1, i_2 \in p^\bullet$, it is clear that such disabling can only take place when the marking of p changes. This

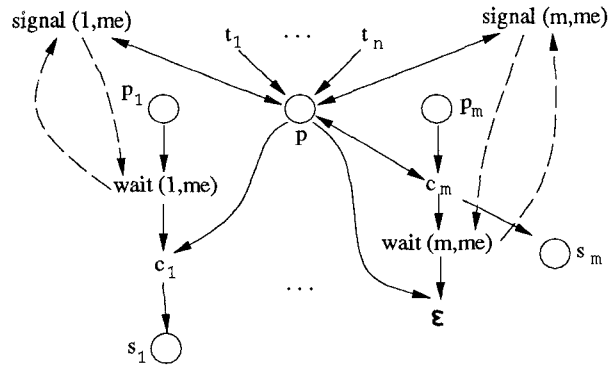

Figure 6: Transformed Petri net for a 1-safe conflict place

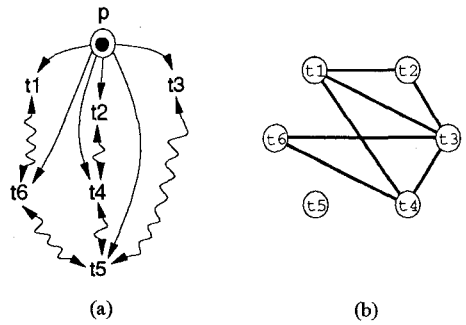

Figure 7: (a) Conflict place. (b) Conflict graph for ME channel assignment.

implies that p is a critical resource that must *protected* by a semaphore mechanism.

C_p can be used to derive a *conflict graph* in which each node represents a transition $t_i \in p^\bullet$. There is an arc between t_i and t_j iff $t_i C_p t_j$. Figure 7.(a) depicts a conflict place with all its successor transitions. Arcs denoted by $t_i \rightsquigarrow t_j$ indicate that t_i and t_j are never simultaneously enabled. Figure 7.(b) shows the conflict graph for such a set of transitions.

Given a conflict graph, the problem of assigning transitions to ME channels can be reduced to the problem of colouring the conflict graph[2]. Therefore, to guarantee mutual exclusion between transitions, an n-input ME will be inserted, n being the number of colours required for the graph.

In the example of Figure 7, a possible channel assignment for a 3-input ME could be the following: $a = \{t_1, t_6\}, b = \{t_2, t_4\}, c = \{t_3\}$.

[2]A dual approach would consist in deriving a *compatibility graph* and reduce the problem to *clique partitioning*

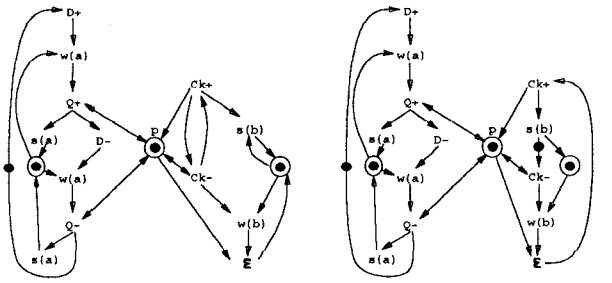

Figure 8: STG of Figure 2 after `wait` and `signal` insertion (`w(a)` stands for `wait(a,me)`).

5 The Latch Example Revisited

We can now revisit the latch example, and apply to it the transformations sketched in Section 4.

The first step, illustrated in Figure 8.(a), consists of

1. Analyzing which transitions are in conflict, building a conflict graph and assigning ME channels to conflicting transitions.
2. Protecting those transitions with a `wait/signal` pair.

Obviously Q^+ and Q^- can never be enabled simultaneously, so they can share channel `a`, while channel `b` is assigned to Ck^-. Note that:

- A place is required for each ME channel, between `signal`s and `wait`s, to ensure their proper ordering.
- We have applied the 1-safe optimization outlined above, in order to obtain a reasonable implementation.

We must then add edges $\epsilon \rightarrow Ck^+$ and $s(b) \rightarrow Ck^-$ to obtain a strongly connected STG, as shown in Figure 8.(b). These edges are *timing constraints* that must be satisfied if the latch in order to properly operate the latch. Basically they state that the clock must wait long enough for the ME to resolve meta-stability. Otherwise, the ME would be operated in a manner that is different than its "legal" I/O protocol, represented in Figure 3. In this case, the *Rb* input may fall before the *Ab* output rises, and the *Rb* input may rise before the *Ab* output has fallen. This may or may not be a problem, depending on the circuit design used to implement the ME. With the solution shown in Figure 9 neither case is problematic, because

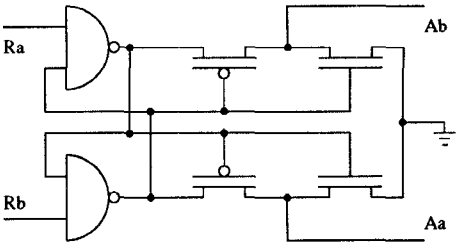

Figure 9: A possible CMOS implementation of an ME

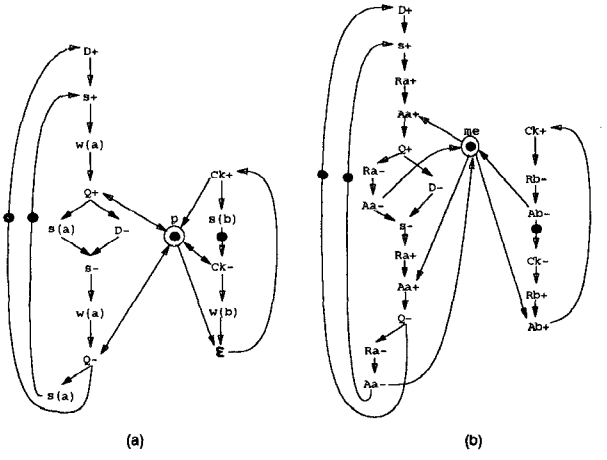

Figure 10: STG of Figure 8 after state encoding (a) and handshake expansion (b).

the former helps the ME leaving a meta-stable state, while the latter may drive it again into meta-stability while the previous meta-stability has not been resolved yet, which only adds to the length of a potentially unbounded process[3]

This first STG does not have CSC, so state signal transitions must be added to it to make it implementable [11, 5]. One possible solution, with one state signal s, is shown in Figure 10.(a). Redundant places (i.e., places whose removal does not affect the enabling/firing conditions of the net) have been removed, for the sake of clarity.

We can now expand the *symbolic* actions `wait` and `signal` to full handshakes, thus obtaining the final STG shown in Figure 10.(b). This STG can be implemented, as shown in Figure 11.(a) (where L elements are "standard" transparent latches). This implementation is interesting, because it corresponds to an *edge-triggered self-clocked flip-flop*, where the "internal" clock is activated whenever there is a change

[3] We conjecture that this observation is generally applicable to the proposed design methodology, but we have not yet fully investigated the issue.

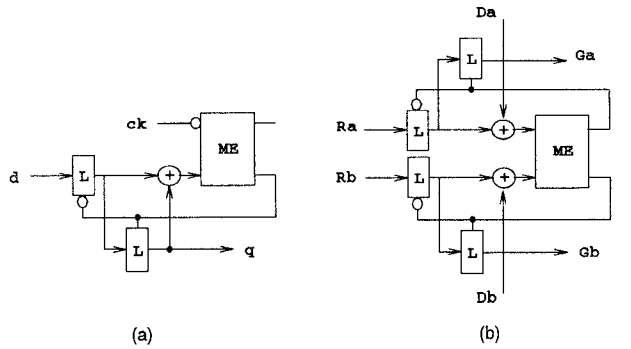

Figure 11: Circuit implemented from the STG of Figure 10(a) and an RGD arbiter built on its basis (b).

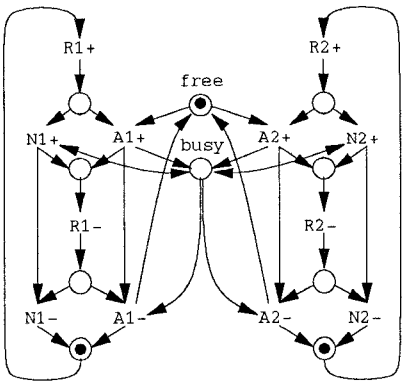

Figure 12: STG for the arbiter with reject.

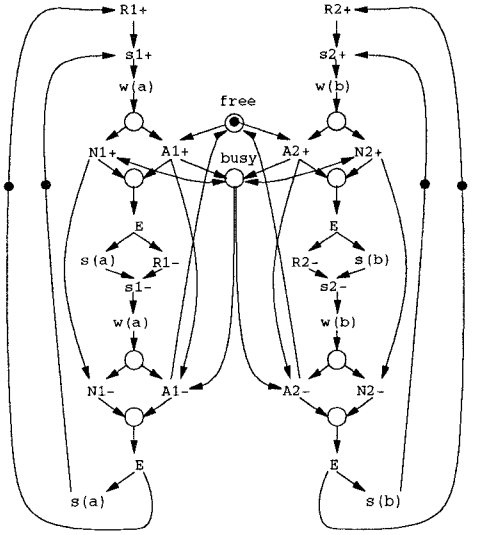

Figure 13: Transformed STG for the arbiter with reject

in the input datum, but only when the external clock Ck is high. Otherwise, Ck locks the ME. This implementation makes use of the fact that D must wait for Q to change before it can change again, in the original specification.

Furthermore, we are not violating any physical principle, because this latch cannot be used to implement an ideal synchronizer, that is known to be impossible, even if it never produces a meta-stable output. The problem is that we don't know when Q will change relative to the falling edge of Ck, while if a latch is used respecting setup and hold times, we know that its output will be stable after a known settling time has elapsed since the falling edge of Ck. So if we use Q for driving combinational logic, we cannot reliably latch its output.

We have two potential advantages from the use of this latch, and of the proposed methodology, though:

1. No output will ever have an invalid logic value, except for the "standard" rising and falling transients.

2. If the clock can be stopped, then the Ab output of the ME can be used to do so and provide reliable, synchronous operation.

Another useful byproduct of this design can be the implementation of a Request-Grant-Done (RGD) arbiter (see, e.g., [10]) shown in Figure 11(b). It makes the (D, Q, s, Ra, Aa) channel in the latch symmetric.

6 Arbiter with reject

The arbiter with reject is another interesting example of a circuit prone to suffer from meta-stability. Its behaviour is described by the STG of Figure 12. Basically, this arbiter must respond to a request either with an *Ack*, if the resource is free, or with a *Nack*, if the resource is busy, thus allowing the requester to do something else.

In this case there are two conflict places, **free** and **busy**, with three possible situations of output non-semi-modularity: **A1+** $\leftarrow$ **free** $\rightarrow$ **A2+**, **A1−** $\leftarrow$ **busy** $\leftrightarrow$ **N2+** and **A2−** $\leftarrow$ **busy** $\leftrightarrow$ **N1+**

By applying the transformations presented in Section 4, a circuit with two MEs (one for each conflict place) would be obtained. Instead of this approach, we have chosen to use only one ME by considering both places as one critical section. Thus, all the aforementioned transitions must be guaranteed an exclusive access to the places **free** and **busy**.

By assuming the reader to be somewhat familiar with the STG-level transformations, we briefly sketch the resulting STGs in Figure 13. Although six differ-

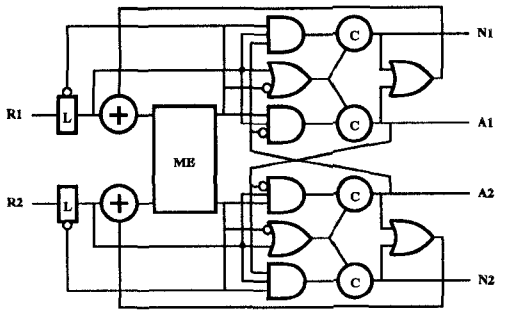

Figure 14: Speed-independent realization of the arbiter with reject

ent transitions have access to the critical section, only two ME channels are required after analysing the conflict relation among them. In the final solution two state signals, s_1 and s_2, have been inserted to obtain an STG with the CSC property. After synthesis, a speed-independent realization such as the one shown in Figure 14 is obtained.

An alternative implementation of the arbiter with reject was initially presented by Nowick and Dill in [4]. Both solutions have a similar structure, although the one presented in this paper is slightly simpler by the fact that the two toggle elements are substituted by two OR gates. The other significant difference is the use of two latches at the inputs instead of two C elements (both gates roughly have the same complexity).

7 Conclusions

This paper tackles the problem of synthesizing circuits from behavioural specifications with internal conflicts. Nowadays, conflicts are resolved by ad-hoc transistor-level circuitry usually developed by designers with high expertise in analogue methods. Otherwise, circuits obtained by direct synthesis from logic functions may suffer from hazardous or meta-stable behaviour.

We have provided a methodology that allows to mechanically solve this problem by using *standard* components devised and tested to avoid meta-stability (i.e. mutual exclusion elements). The original description is transformed into another one in which MEs can be directly inserted to resolve conflicts while existing CAD tools can be used to derive the logical part.

It is worth to emphasize that the presented methodology can be incorporated into automatic synthesis tools, thus helping even an inexperienced designer to correctly tackle synchronization problems.

References

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

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

[3] M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, and V. I. Varshavsky. *Concurrent Hardware. The Theory and Practice of Self-Timed Design.* John Wiley and Sons Ltd., 1994.

[4] S. M. Nowick and D. L. Dill. Practicality of state-machine verification of speed-independent circuits. In *Proc. of the Int. Conf. on Computer-Aided Design*, Nov. 1989.

[5] E. Pastor and J. Cortadella. Polynomial algorithms for the synthesis of hazard-free circuits from signal transition graphs. In *Proc. of the Int. Conf. on Computer-Aided Design*, Nov. 1993.

[6] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P. Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37:1005–1018, 1988.

[7] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Int. Workshop on Timed Petri Nets*, 1985.

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

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

[10] I. E. Sutherland. Micropipelines. *Comm. of the ACM*, June 1989. Turing Award Lecture.

[11] P. Vanbekbergen, B. Lin, G. Goossens, and H. D. Man. A generalized state assignment theory for transformations on Signal Transition Graphs. In *Proc. of the Int. Conf. on Computer-Aided Design*, pages 112–117, Nov. 1992.

[12] A. V. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. In *Proc. of the Int. Conf. on Computer-Aided Design*, Nov. 1992.