Automatic synthesis and optimization of partially specified asynchronous systems

Alex Kondratyev
Univ. of Aizu
Japan

Jordi Cortadella
Univ. Politècnica
Catalunya, Spain

Michael Kishinevsky
Intel Corp.
USA

Luciano Lavagno
Univ. of Udine
Italy

Alexander Yakovlev
Univ. of Newcastle
upon Tyne, UK

Abstract

A method for automating the synthesis of asynchronous control circuits from *high level* (CSP-like) and/or *partial* STG (involving only *functionally critical* events) specifications is presented. The method solves two key subtasks in this new, more flexible, design flow: *handshake expansion*, i.e. inserting reset events with maximum concurrency, and *event reshuffling* under interface and concurrency constraints, by means of concurrency reduction. In doing so, the algorithm optimizes the circuit both for size and performance. Experimental results show a significant increase in the solution space explored when compared to existing CSP-based or STG-based synthesis tools.

1 Introduction

Specifying an asynchronous circuit is a cumbersome and error-prone task because the designer has to define the behavior of every signal at every moment of time. Although the value of a signal might be sometimes irrelevant to the general functioning of the system, one must be specific about its behavior by exactly defining whether the signal is stable at 0 or 1 or making a rising or falling transition.

To circumvent this problem, the designer should be able to specify the behavior of a circuit by only defining those events that are relevant to its function - they are called *functional* events. The rest of the events (*non-functional*) can be defined arbitrarily under the requirement of preserving the correctness of the circuit behavior. This is exemplified by the gate-level implementation of a rising edge-triggered flip-flop. Only the rising edge is "functional", and must have a precise relationship with the input and the output signals (setup/hold constraints and output delay respectively). The falling edge can occur almost at any time between two consecutive rising edges. In the asynchronous context, this kind of freedom provides additional room for optimization under different cost functions aimed at area and/or performance.

There are various design scenarios in which this approach may be useful:

1. The designer concentrates on the key functional aspects and, e.g., specifies only the rising edges of signals. A tool automatically inserts non-functional events. Even when all events are functional, there is some freedom in making them either ordered or concurrent. The designer restricts some functionally important concurrency/ordering relations and allows the tool to choose how to reduce concurrency and optimize the circuit.
2. The designer uses a high-level language, such as CSP [1, 6], that ignores the binary nature of circuit signals and specifies the behavior in terms of abstract events. The following two design steps must then precede logic synthesis:

   handshake expansion : replacing each communication action of a CSP program with signal transitions on the two wires that constitute the channel,

   reshuffling : selecting the order of some non-functional events (return-to-zero signal transitions in four phase expansion of the channels) for optimizing area, performance or power.

   In this paper we solve the problem of handshake expansion in a canonical fashion, by inserting "reset" events with maximum concurrency with respect to the other signals. We then solve the problem of reshuffling by only considering the operation of *concurrency reduction*.

The idea of using concurrency reduction as an efficient method in the optimization loop was first proposed in [5]. The main distinctive features of our approach with respect to that work are:

1. The reduction mechanism is applied in a wider framework (handshake expansion, reshuffling), instead of working at the level of completely specified State Graphs.
2. A reduction based on removal of State Graph *arcs* is used, instead of coarser techniques based on removal of *states*.
3. Not every form of concurrency reduction can be modeled by a sequence of pairwise reductions. In [3] and Section 5 a more general (albeit expensive) technique is discussed.
4. The reduction procedures presented in this paper are aimed at the general minimization of logic, instead of only solving the CSC problem.

In the rest of the paper, after Section 2, devoted to theoretical background, and Section 3, devoted to an informal overview, we will answer the following questions: (1) How is concurrency exploited starting from a partial specification of an asynchronous controller? (Section 4); (2) What are the valid reductions of concurrency? (Section 5); (3) How can concurrency be reduced by iterative application of a single, elementary operation? (Section 6); (4) How is the quality of the solution estimated? (Section 7). Section 8 presents experimental results.

2 Theoretical background

This section assumes the reader to be familiar with Petri nets [7].

Figure 1.a shows a timing diagram of a simple controller between an asynchronous memory and a processor. An operational cycle is triggered by the processor requesting data (Req goes high). After this request, memory prepares data and the controller replies with an acknowledgment (Ack goes high). From now on the processor can reset the request and immediately start a new cycle. Note

Definitive Version of Record in the ACM Digital Library: https://doi.org/10.1145/309847.309891

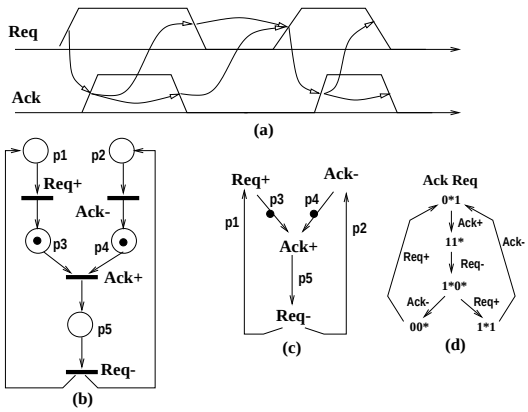

Figure 1: Simple asynchronous controller: (a) waveforms, (b,c) STG, (d) State Graph

that in order to increase the system throughput, the processor can send a new request without waiting for the reset of the acknowledgment signal by the controller. Figure 1.b shows the Petri Net (PN) corresponding to the timing diagram of the controller. All events in this PN are interpreted as signal transitions: rising transitions of signal a are labeled with "$a+$" and falling transitions with "$a-$". We also use the notation $a*$ if we are not specific about the sign of the transition. Petri Nets with such interpretation of the transitions are called *Signal Transition Graphs (or* STG*s)* [2].

STG transitions correspond to system events. A transition is *enabled* if all its input places contain a token. In the initial marking of the STG in Figure 1.c transition $Ack+$ is enabled. Every enabled transition can fire, removing one token from every input place of the transition and adding one token to every output place. After the firing of transition $Ack+$ the net moves to a new marking $\{p_5\}$ and $Req-$ becomes enabled, etc.

State graphs. Playing the token game one can generate a *State Graph (*SG*)* in which each node (a marking) is labeled with a vector of signal values (signals that can change in the state are marked with an asterisk) and arcs between pairs of states are labeled with the corresponding fired transition. An SG is *consistent* if its state labeling $v : S \rightarrow \{0,1\}^n$ is such that in every transition sequence from the initial state, rising and falling transitions alternate for each signal. Figure 1.d shows the SG for the STG in Figure 1.c, which is consistent. The notation 0*1 in the initial state of Figure 1.d indicates that signal Ack has value 0 and $Ack+$ is enabled to fire, while signal Req is stable at value 1.

We write $s \stackrel{a}{\rightarrow} (s \stackrel{a}{\rightarrow} s')$ if there is an arc from state s (to state s') labeled with a and $s \stackrel{\sigma}{\Rightarrow} s'$ is there is a path from state s to state s' labeled with a sequence of events σ.

The set of all signals is partitioned into a set of *inputs*, which come from the environment, and a set of *outputs* and *state* signals that must be implemented.

Implementability conditions In addition to consistency, the following two properties are required for an SG to be implementable into a hazard-free asynchronous circuit.

The first property is *speed independence*, with three constituents: determinism, commutativity and output-persistency. An SG is *deterministic* if for each state s and each label a there can be at most one state s' such that $s \stackrel{a}{\rightarrow} s'$. An SG is *commutative* if whenever two transitions can be executed from some state in any order, their execution always leads to the same state, regardless of the order. An event a^* is *persistent* in state s if it is enabled in s and remains

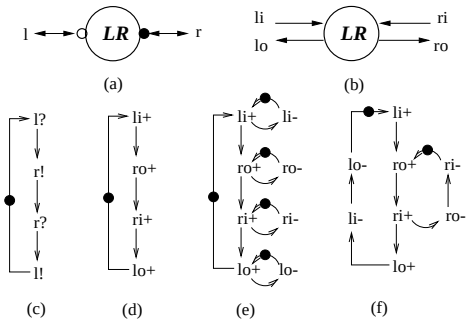

Figure 2: Specification of the LR-process

enabled in any other state reachable from s by firing another event b^*. An SG is *output-persistent* if all output signal events are persistent in all states and input signals cannot be disabled by outputs.

The second property, *Complete State Coding* (CSC), is necessary and sufficient for the existence of a logic circuit implementation. A consistent SG satisfies the CSC property if for every pair of states s, s' such that $v(s) = v(s')$, the set of output events enabled in both states is the same. The SG of Figure 1.d is *output-persistent* but does not have CSC (binary codes 11* and 1*1 correspond to different states).

Excitation Regions and Concurrency. A set of states is the *excitation region* (ER) of event a^* (denoted by $ER(a^*)$) if it is a *maximal connected* set of states such that $\forall s \in ER(a^*) : s \stackrel{a^*}{\rightarrow}$.

SGs are used in this paper as the main model for performing the concurrency reductions. Hence we need to define the concurrency notion in terms of the SG.

Definition 2.1 *Two events a and b are said to be concurrent in the* SG *A ($a||b$) if the following diamond structure of states and transitions belongs to A:*

$$(s_1 \stackrel{a}{\rightarrow} s_2) \wedge (s_1 \stackrel{b}{\rightarrow} s_3) \wedge (s_2 \stackrel{b}{\rightarrow} s_4) \wedge (s_3 \stackrel{a}{\rightarrow} s_4).$$

It can be easily shown that for a speed-independent SG two output events a and b are concurrent iff their ERs intersect:

$$a||b \Leftrightarrow ER(a) \cap ER(b) \neq \emptyset.$$

In the SG of Figure 1.d transition $Req+$ is enabled in states 1*0* and 00* ($ER(Req+)$={1*0*,00*}) while $Ack-$ is enabled in 1*0* and 1*1 ($ER(Ack-)$={1*0*,1*1}). Excitation regions of these transitions intersect, thus implying that the corresponding transitions are concurrent.

3 Overview of the method

We illustrate our methodology by means of an example. Figure 2.a shows the structure of an LR-process [6] using the "handshake component" notation [1]. The process has a passive port l and an active port r. It transfers control from the left port to the right port. Figure 2.b shows the refinement of each channel with two wires: $l = \{li, lo\}$ and $r = \{ri, ro\}$. Figure 2.c gives a specification of this process using CSP-like actions for events, where $l?, l!$ ($r?, r!$) stand for the input and output actions at channel l (r). Figure 2.d presents a handshake expansion of the previous specification. It is obtained by relabeling channel actions $l?$ and $l!$ to rising transitions on the input and output wires of the ports, $li+$ and $lo+$, correspondingly (the same for channel r). The latter specification is viewed as a partially specified STG. It cannot be directly implemented by existing STG-based synthesis tools since the falling (reset) transitions

Circuit	Area		Performance	
	area	# CSC sign.	cr.cycle	inp.events
Q-module (hand)	104	1	14	4
Full reduction	0	0	8	4
Max.concurrency	168	2	13	3
$li \parallel ri$	144	0	9	3
$li \parallel ro$	160	1	11	3
$lo \parallel ri$	136	1	11	3
$lo \parallel ro$	232	2	16	3

Table 1: Area/performance trade-off for different implementations of the LR-process

of the signals are not specified. There are many different solutions for inserting falling signal transitions. Starting from the solution with maximum concurrency one can derive any other valid reshuffling of transitions by concurrency reduction. Figure 2.e shows an STG with maximal concurrency for all falling transitions, assuming that all signals are independent, and that no interface constraints were given.

This handshake expansion however is not valid for the LR-process. Indeed, we should obey additional ordering constraints for the channels: never reset the requesting signal before receiving the acknowledgment. For example, for a passive port l one should satisfy the following interleaving of signal transitions:

$$*[li+; lo+; li-; lo-]$$

Similarly for the active channel. Figure 2.f presents a valid handshake expansion with maximal concurrency for the LR-process taking interface constraints into account.

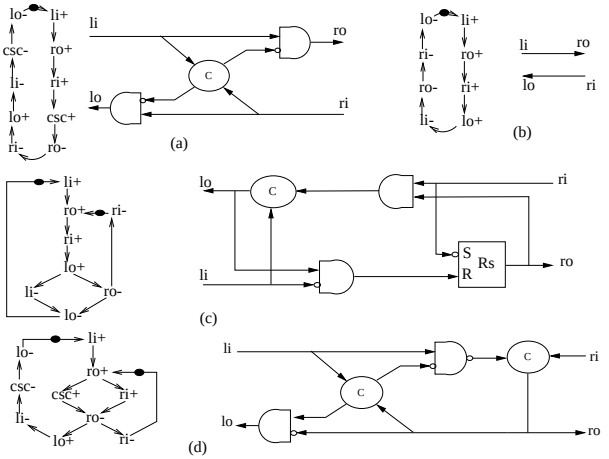

Figure 3: Implementations of the LR-process

This specification can be implemented with the current STG based synthesis tools. Two state signals are inserted for resolving the Complete State Coding (CSC) conflicts.

Table 1 presents the area and performance results for different implementations of the LR-process. The row "Max. concurrency" corresponds to the implementation of the STG with maximum concurrency of the reset signal transition. The circuit area is 168 units. Assuming that all internal and output events have a delay of 1 time unit, and that all input events have a delay of 2 time units, the critical cycle is 13 units and contains 3 input events. Other implementations are shown in Figure 3. Figure 3.a shows an implementation of the LR-process known widely as Q-module [6] or S-element [1]. Figure 3.b corresponds to the case of full concurrency reduction. It produces the best area (two wires) but does not allow to decouple the left and the right sides of LR-process.

The above examples suggest the algorithm for optimization of partially specified STGs shown in Figure 4.

```
Inputs:  Initial STG
         Interface constraints (channel interleaving)
         Concurrency constraints (concurrent events,...)
Output:  Reduced State graph and the corresponding STG
1:  Insert the ''reset'' transitions with maximal
    concurrency, satisfying all interface constraints
2:  Generate SG A from the STG
3:  while the cost improves do
4:       Reduce concurrency of SG A, satisfying
         interface and concurrency constraints
         reducing CSC conflicts and logic complexity
    endwhile
5:  Generate a new STG for the best reduced SG
```

Figure 4: Handshake expansion and reshuffling for STGs

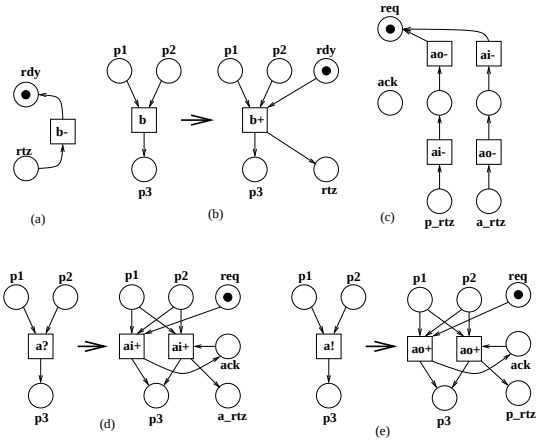

Figure 5: STG structures for 4-phase refinement. Partially specified signal: return-to-zero event (a) and functional event (b). Channel: return-to-zero part (c) and functional parts, for input channel (d) and for output channel (e).

4 Handshake expansion

This section explains how handshake expansion is performed. The syntax of our specifications allows one to describe the behavior of *channels* and *partially specified signals*. In both cases, the specification only contains the *active* transitions, whereas the handshake expansion method transforms the specification according to the refinement chosen by the designer: *2-phase refinement*, with no distinction between up and down transitions, or *4-phase refinement*, with return-to-zero signaling for each handshake.

Partially specified signals. The STG transformation required to expand a partially specified signal is shown in Figure 5.a and b. Figure 5.a illustrates an additional return-to-zero transition that must be connected (using the places labelled `rdy` and `rtz`) to the functional part corresponding to the rising transition of the signal, shown in Figure 5.a. Note that each rising transition is enabled only when the return-to-zero transition has fired (arc `rdy` $\rightarrow b+$). The return-to-zero transition is enabled as soon as the rising transition has fired (arc $b+ \rightarrow$ `rtz`).

Channels. For channel refinement we use a notation similar to that proposed for *handshake processes* [1]. Two types of events can occur in channel a: input events ($a?$) and output events ($a!$). The terminals of a channel are called *ports*. A channel a is implemented by two signals: a_i (input) and a_o (output).

The expansion from channel to signal events can be done by manipulating the structure of the underlying Petri net. For 2-phase refinement, the transformation simply requires relabeling the STG

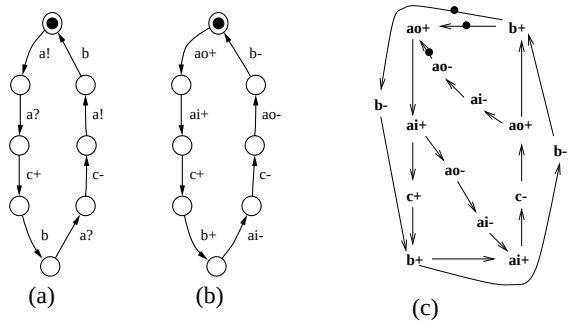

Figure 6: (a) Original specification (SG), (b) 2-phase refinement (SG), (c) 4-phase refinement (STG).

transitions from $a?$ to $a_i\tilde{}$ and $a!$ to $a_o\tilde{}$, where the suffix $\tilde{}$ denotes a transition toggling the value of the signal.

The expansion to a 4-phase protocol is performed by relabeling transitions and inserting return-to-zero events. The transformations performed at the STG level consist of adding a return-to-zero structure and defining multiple instances of the transitions representing channel events. The return-to-zero structure corresponding to a channel is depicted in Figure 5.c. The place `req` indicates that the channel is ready for a new handshake. The place `ack` indicates that the channel has received a request ($a?$ for passive and $a!$ for active handshakes) and will perform an acknowledgment ($a!$ for passive and $a?$ for active handshakes). The places `p_rtz` (for passive) and `a_rtz` (for active) receive a token as soon as the handshake is complete and activate the return-to-zero transitions. This scheme allows a channel to *act both as an active and as a passive port at different instants of the behavior of the system*.

Figures 5.d.e show how channel events are translated into actual signal events by structural transformations of the STG. Each event $a?$ is transformed into a rising transition of the input signal (a_i+). Similarly, $a!$ is transformed into a_o+. Two instances of a_i+ and a_o+ in Figures 5.d.e model different types of channel behavior (active or passive). The parallel composition of the STG pieces of Figures 5.c.d.e gives an overall picture of the channel behavior in the set and reset phases. Note that the specification must properly interleave the events on the channel according to the handshake protocol, otherwise the expansion would produce an inconsistently encoded STG. *This scheme guarantees the maximum concurrency for the return-to-zero sequence*, that is then exploited by the concurrency reduction algorithm described in Section 3.

Example. Figure 6 presents an example illustrating all the above transformations. The original specification (Figure 6.a) has a channel (a), a partially specified signal (b) and a completely specified signal (c). Two-phase and four-phase refinements of the same specification are shown in Figure 6.b.c.

5 Concurrency reduction

In this section we develop the theory and algorithms that allow us to explore only *valid* reductions of concurrency more efficiently than by working on a state-by-state basis. In particular, our notion of concurrency reduction is related to the introduction of places (causal constraints) at the STG level, and then "fixing" the STG so that consistency and speed-independence are preserved.

Valid concurrency reduction should preserve certain properties. Let A be the initial SG and A_{red} be a reduced SG. Reducing concurrency for event e means truncating some ERs of this event. In other words, some of the arcs labeled with e are removed from the SG as a result of concurrency reduction. This may cause some of the states to become unreachable and to be removed from the SG.

No states or arcs not present in the initial SG can appear in A_{red}. This trivially implies that consistency, commutativity, and determinism of the SG cannot be violated as a result of concurrency reduction. Also no new CSC conflicts can appear (in fact some or all of the conflicts can disappear due to state removal).

Validity then requires the following properties to be satisfied after concurrency reduction:

1. Speed-independence is preserved: as noted above, commutativity and determinism are automatically preserved, so the only constraint is that if A is output persistent, then A_{red} must be output persistent.
2. I/O interface is preserved:
   (a) No transition of input signals is delayed.
   (b) The initial state is preserved with respect to the I/O signals, i.e., if $s_0 \in A$ and $s_0' \in A_{red}$ are the initial states of the original and the reduced SGs respectively, then there is a path $s_0 \stackrel{\tau}{\Rightarrow} s_0'$ or $s_0' \stackrel{\tau}{\Rightarrow} s_0$ in A such that sequence τ contains only events of *internal* signals, not observable by the environment.

   Both conditions can in fact be partially relaxed if the designer can accept changing the interface behavior of the module, e.g., if also the environment will be synthesized later.
3. No events disappear: if for some event e there is $ER(e) \in A$, then $ER_{red}(e) \neq \emptyset$.
4. No deadlock states appear: if state $s \in A$ and $s \in A_{red}$, and s is not a deadlock state in A (there exists event e: $s \stackrel{e}{\rightarrow} \in A$), then there exists some other event e' such that: $s \stackrel{e'}{\rightarrow} \in A$ and $s \stackrel{e'}{\rightarrow} \in A_{red}$.

Whenever concurrency is reduced for an output signal, one must also make sure that this is reflected in the specification of the behavior assumed by the environment (e.g., by another design team). Otherwise, concurrency reduction may introduce deadlocks in the composition of the circuit and the environment, e.g., if the environment expects b after a and the circuit provides b before a as a result of two conflicting concurrency reductions for initially concurrent events a and b.

Definition 5.1 (Valid reduction) *If a reduced* SG *satisfies all properties (1)–(4) above, then the concurrency reduction is* valid.

6 The basic operation: forward reduction

The algorithm sketched in Figure 7 defines our basic operation for concurrency reduction, called *forward reduction*. It takes two concurrent events as parameters. Concurrency is reduced for the first event (a). The second event (b) defines the set of states $ER(a) \cap ER(b)$ in which concurrency for a should (at least) be reduced in one step. In the simplest case, when events enabled in $ER(a)$ are persistent, and $ER(a)$ has only one minimal state (a state is minimal in an ER if it has no predecessors in the ER), `FwdRed(a,b)` *creates an arc from event b to event a* at the STG level.

The application of the forward concurrency reduction `FwdRed` to an STG with choice (non-persistency) and concurrency is illustrated in Figure 8. The reduced SG corresponds to an STG with no concurrency between (a, b), (a, e), and (a, d). Hence, in general reducing concurrency for a pair of events can also reduce concurrency for some other pairs. Note that in lines 1,2 of `FwdRed`, states are removed from the ER of event a, not from the SG. I.e., at this step only arcs labeled with a can be removed from the SG.

The following proposition shows that iterative application of `FwdRed` to an SG results in a valid concurrency reduction.

```
FwdRed(a,b)
1:  /* remove all arcs s -a-> such that s
       is backward reachable from ER(a) ∩ ER(b) */
2:  ER_red(a)=ER(a) − (ER(b) ∪ back_reach(ER(a) ∩ ER(b)))
3:  remove unreachable states and their output arcs
4:  if exists some e such that ER(e) = ∅ or
5:      initial state wrt to I/O is changed then
6:      return (invalid reduction)
7:  else return (reduced SG)
```

Figure 7: Reduction of concurrency for output event a by event b.

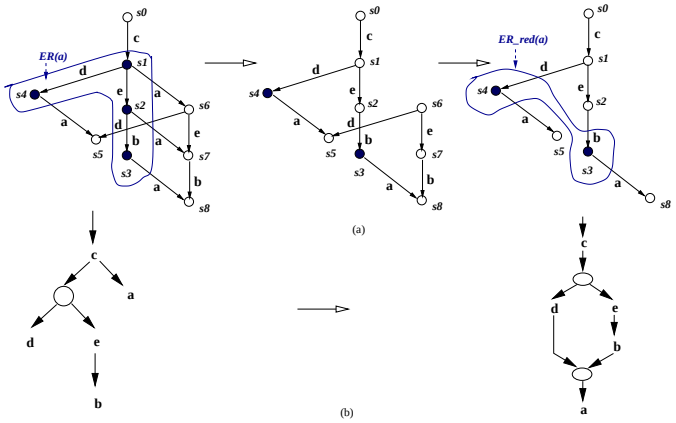

Figure 8: Applying `FwdRed(a,b)` to an SG fragment (a) and the corresponding STG transformation (b).

Proposition 6.1 (Validity of `FwdRed`) *Let A be a consistent and speed-independent* SG. *If a is an output event and a and b are concurrent in A, then* `FwdRed(a,b)` *is a valid concurrency reduction (See [3] for the proof).*

Note. A more general formulation of concurrency reduction is done via the removal of a single arc from the corresponding SG through the notion of *backward reduction* [3]. However, contrary to `FwdRed(a,b)` backward reduction in general does not have a clear interpretation in terms of ordering relations between events. Therefore, our practical implementation described in the next section is restricted to the application of `FwdRed`.

7 Implementation

As we mentioned in Section 3, concurrency reduction can reduce the logic complexity of the circuit in two ways. First of all, the number of CSC conflicts is reduced, and hence the complexity of the logic implementing the state signals is reduced. Secondly, the number of reachable states is reduced, and hence the don't care set for logic minimization is increased. However, in case one signal becomes ordered with another, the support of its boolean function increases. For this reason, we use a heuristic cost function that estimates changes in logic complexity at each step, since exact computation by state signal insertion, decomposition and technology mapping would be too expensive.

The algorithm in Figure 9 describes how concurrency reduction is performed. The designer initially provides a list of pairs of events whose concurrency cannot be reduced, e.g., because they are crucial for overall system performance. This will prevent the algorithm from adding causality relations between these pairs of events.

The exploration is done by a strategy similar to the $\alpha - \beta$ pruning commonly used in game-playing algorithms. At each level of the exploration from a given configuration, a set of neighbor configurations is generated by performing a basic transformation (forward concurrency reduction between two events). For each level of the exploration, only a few candidates, with the best estimated cost, survive to the next level. These candidates are kept in the list `frontier`. The width of the exploration is controlled by the parameter `size_frontier`.

Note that at each level of the exploration the obtained state graphs are less concurrent than their predecessors. This monotonous behavior guarantees that the algorithm will terminate when no more concurrency can be reduced in the current search space.

The cost function to select the best configurations at each level aims at reducing the complexity of the resulting circuit. Unfortunately, the estimation of the complexity of the logic for output signals with CSC conflicts can be inaccurate due to the impossibility to derive correct equations. For this reason, the cost function combines the information of CSC conflicts with the estimated complexity of the logic. A designer can specify a parameter W ($0 \leq W \leq 1$) which defines the trade-off between biasing the heuristic search towards reducing CSC conflicts ($W \rightsquigarrow 0$) or reducing estimated complexity of the logic ($W \rightsquigarrow 1$).

Inputs: State graph `initial_SG`
`Keep_Conc` $\subseteq E \times E$ (preserved concurrency relations)
`size_frontier`: size of the frontier for exploration
Output: State graph `reduced_SG` with reduced concurrency

frontier = explored_SGs = {`initial_SG`};
while frontier $\neq \emptyset$ **do**
  new_solutions = $\emptyset$;
  foreach SG $\in$ frontier **do**
    foreach (e_1, e_2):s.t. $e_1 \parallel e_2, (e_1, e_2) \notin$ `Keep_Conc`
        and e_2 is not an input event **do**
      new_SG = FwdRed (SG,e_2,e_1);
      explored = explored $\cup$ {new_SG};
      new = new $\cup$ {new_SG};
    endfor
  endfor;
  frontier = "the best `size_frontier` elements in `new`";
endwhile;
`reduced_SG` = "best element in `explored`";

Figure 9: Algorithm for reducing concurrency.

8 Experimental results

The techniques presented in this paper have been implemented in the tool `petrify` [4]. After handshake expansion and concurrency reduction, circuits have been derived by using previously published synthesis techniques for speed-independent circuits. The final area was obtained by decomposing the circuit into 2-input gates and mapping the network onto a gate library. The decomposition was performed by preserving the speed-independence of the circuit.

First case study: the *PAR* component This section presents a case study considering the handshake expansion and concurrency reduction of the *PAR* component used in VLSI programming from the concurrent language Tangram [1].

Figure 10.a shows an STG specification in terms of channel events. This specification may yield different implementations depending on the selected phase refinement and concurrency among events. The most challenging problem arises when a 4-phase refinement is desired. The freedom to schedule the return-to-zero transitions opens a spectrum of different implementations. Figure 10.c [9] (see implementation in Figure 10.f) has been obtained manually and is used by the current Tangram compiler.

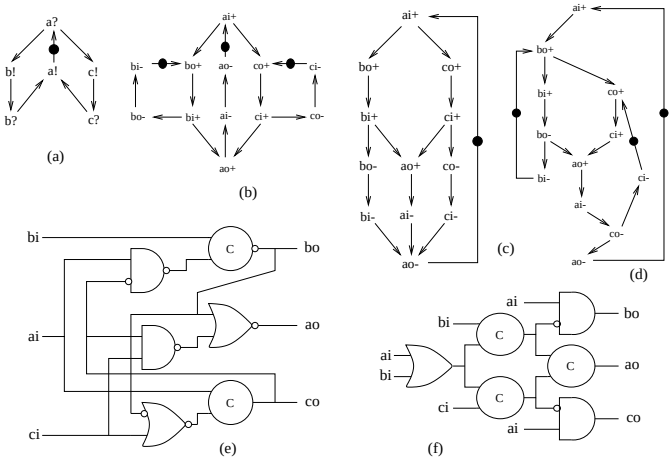

Figure 10: Different specifications and implementations for a PAR component.

Our tool can automatically perform a 4-phase expansion by using the structural techniques discussed in Section 4, and derive the specification shown in Figure 10.b. After this transformation, the return-to-zero signalling is performed with maximum concurrency. However, a direct implementation of this behavior would result in a complex circuit due to the need of inserting extra logic for state encoding and logic decomposition (twice as complex as Figure 10.e).

Figures 10.d.e depict the solution automatically obtained by reducing the concurrency of the 4-phase refinement in Figure 10.b. The reduction has been performed by preserving the concurrency between the events $b?$ and $c?$, thus maintaining the parallel execution of both processes. Interestingly, the circuit manifests an asymmetric behavior that can be beneficial to implement *PAR* components in which the process at channel b is known to be slower than that at c. The circuit is slightly smaller (by 12% in our standard cell library) than the known manual design. However, its estimated performance may be worse than that of Figure 10.f, if b and c have balanced delays[1].

Second case study: the MMU controller In [8] it was shown that by using timing assumptions on the behavior of the environment, it is possible to reduce the area of an asynchronous Memory Management Unit control circuit by over 50 %, with respect to the original speed-independent implementation. Our experiments presented in Table 2 show that approximately the same area improvement can be reached *without sacrificing speed-independence*, if we are allowed to use flexibility in playing with concurrency of the reset transitions of the four-phase protocol. A combination of our high-level transformation and Myers' lower level timing optimizations can conceivably provide even better optimization results.

We can conclude that:

- With respect to the original solution, reshuffling can yield an area reduction to less than one half.
- This area reduction can be obtained without losing performance. E.g., the solution $\parallel (b, m, r)$ with area 384 units has a critical cycle of 94 units, while the original implementation with area 744 had a critical cycle of 100 units We used the same timing delay assumptions as in [8]. In case [8] used a finite delay interval, we considered the average delay, while in case the upper bound was infinite, we considered the lower bound.

[1] The critical cycle is longer by 11% under the assumption that the delay of a combinational gate is 1 time unit, that of a sequential gate is 1.5 time units, and that of an input event is 3 time units.

Circuit	Area		Performance	
	area	# CSC sign.	cr.cycle	inp.events
original	744	2	100	4
original reduced	208	0	118	6
csc reduced	96	1	123	7
$\parallel (b, l, r)$	440	1	101	4
$\parallel (b, m, r)$	384	0	94	4
$\parallel (b, l, m)$	352	1	104	5
$\parallel (l, m, r)$	368	1	105	5

Table 2: Area/performance trade-off for different implementations of the MMU controller

9 Conclusions

Specifying the behavior of an asynchronous system is a complex task that needs to be performed at the appropriate high level of abstraction. Reasoning in terms of actions (or events) and communication channels allows the designer to describe a behavior without worrying about the implementation details.

This paper has presented a method to automate the decisions taken at the lowest levels of circuit synthesis, concerning phase refinements and event reshuffling. Thus the designer is only left the task of defining the causality among actions and specifying the desired concurrency in the system. The task of translating actions into signals transitions is automatically handled by CAD tools.

Some aspects still require further research. In particular, better logic estimation strategies when the specification has CSC conflicts must be sought. On the other hand, simple but accurate methods for performance estimation should be devised to increase the degree of automation and provide a wider exploration of the solution space.

Acknowledgments. We thank Steve Furber for emphasizing the need to tackle the problem of automatic handshake expansion and concurrency reduction. This work was supported by ESPRIT ACiD-WG (21949), CICYT TIC 98-0410 and TIC 98-0949, UK EPSRC GR/K70175 and GR/L24038, and British Council (Spain) Acción Integrada MDR/1998/99/2463.

References

[1] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

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

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Automatic handshake expansion and reshuffling using concurrency reduction. In *Workshop on Hardware Design and Petri Nets*, pages 86–110, June 1998.

[4] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.

[5] Bill Lin, Chantal Ykman-Couvreur, and Peter Vanbekbergen. A general state graph transformation framework for asynchronous synthesis. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 448–453. IEEE Computer Society Press, September 1994.

[6] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Straunstrup, editor, *Formal Methods for VLSI Design*, chapter 6, pages 237–283. North-Holland, 1990.

[7] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.

[8] Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[9] Ad Peeters. Implementation of a parallel component in tangram. Personal communication, 1997.