Automatic handshake expansion and reshuffling using concurrency reduction

Jordi Cortadella[1], Michael Kishinevsky[2], Alex Kondratyev[3], Luciano Lavagno[4], and Alex Yakovlev[5]

[1] Univ. Politècnica de Catalunya, Barcelona, Spain, email: jordic@ac.upc.es
[2] Intel Corp., Hillsboro, OR, USA, email: mkishine@ichips.intel.com
[3] The Univ. of Aizu, Japan, email: kondraty@u-aizu.ac.jp
[4] Politecnico di Torino, Italy, email: lavagno@polito.it
[5] Univ. of Newcastle upon Tyne, U.K., email:alex.yakovlev@ncl.ac.uk

Abstract. We present an automatic technique for handshake expansion and reshuffling in asynchronous control circuit design. A designer can define the behavior of a control circuit using a CSP-like or STG-like notation. This definition may be *partial*, i.e. involve only *some* of the actual events.
The method solves handshake expansion by inserting reset events with maximal concurrency and then automatically explores reshuffling of events under interface and concurrency constraints using a single basic operation – concurrency reduction. The algorithm trades off the number of required state signals and logic complexity.
Experimental results show that this approach provides more flexibility in the design process than completely specified CSP or STG. It significantly increases the solution space for logic synthesis compared to existing CSP-based or STG-based synthesis tools.

1 Introduction

[11] and [1] proposed two techniques for compilation of CSP-like specifications into asynchronous circuits. Those techniques, in principle, assume that two design steps should precede logic synthesis:

- *handshake expansion*: replacement of each communication action of a CSP program with its implementation as signal transitions on the two wires that constitute the channel,
- *reshuffling*: selecting the order of actions (reset signal transitions in four phase expansion of the channels) for optimizing area, performance or power.

This paper presents an automatic technique for solving these steps by using concurrency reduction as a basic operation (first introduced in [21, 23], where it was used to solve the state encoding problem).

We apply the same method also in a more general context of Signal Transition Graph-based (STG) synthesis as follows. Starting from a *partial* specification of the control, which specifies only *some* signal transitions important for the designer, insert reset signal transitions such that:

- consistency of the specification is maintained automatically (i.e., rising and falling transitions for each signal alternate);
- speed-independence of the specification is preserved;
- interface behavior of the specification is preserved;
- specified constraints on concurrency of events are satisfied for performance optimization.

The technique that we propose can be used in several applications in asynchronous circuit design. The first case, that we illustrate by example in this section, is the implementation at the gate level of high-level components, such as those used in some Delay Insensitive compilation techniques [1]. This is useful, for example, to re-target the compiler to use a new technology by exploiting a synchronous (perhaps slightly enhanced with a few asynchronous elements) standard cell library. A second case is the optimization of several components generated by compilation. This optimization can be performed either by combining complete STG specifications [17], or (potentially with better results) by combining only the active transitions, and then adding return-to-zero transitions by the techniques illustrated in this paper. A third case, obviously, is to help manual designers in concentrating on the most significant functional aspects (e.g., the relations between the active transitions in four phase handshaking and latch control), leaving to a tool the most repetitive aspects such as reshuffling and concurrency reduction.

The idea of using the concurrency reduction as an efficient tool in the optimization loop was first proposed in [22]. According to the heuristic strategy used in that paper, a specification is simplified by the iterative application of a basic operation that reduces the concurrency for a certain pair of transitions. The forward reduction mechanism suggested in Section 4 is a generalization of this basic operation, in that it satisfies several new correctness requirements. The method of concurrency reduction was further developed in [10] in the framework of the satisfiability approach to State Graph-based asynchronous circuit synthesis, where several reductions of concurrency by pairs of transitions were performed by solving an instance of the satisfiability problem. The satisfiability approach is attractive due to its generality (every combinatorial optimization problem can be formulated in that way), but it suffers from performance problems, and offers little control over the quality of the final solution. In particular, due to the need to reduce the solution space, the approach of [10] was limited to a smaller set of transformations than those originally described in [22].

The main distinctive features of our approach with respect to [22, 10] are:

1. We use a finer reduction, by removal of State Graph arcs, than their technique based on removal of states.
2. Not every form of concurrency reduction can be modeled by a sequence of pairwise reductions. Hence in Section 5.1 we develop a more general (albeit expensive) technique.
3. The reduction procedures presented in our paper are aimed at the general minimization of logic, while in [22, 10] they were aimed only at the solution of the CSC problem.

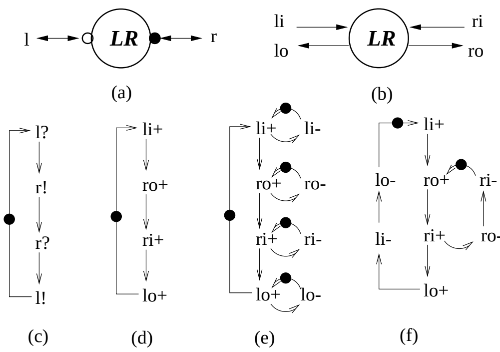

Fig. 1. Specification of the LR-process

1.1 A motivating example

Figure 1,a shows the structure of an LR-process [11] using the "handshake component" notation [1]. The process has a passive port l and an active port r and should transfer control from the left port to the right port. Figure 1,b shows expansion of each channel with two wires such that $l = \{li, lo\}$ and $r = \{ri, ro\}$. Figure 1,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). Assuming four-phase implementation of the channel and positive encoding of channel actions, Figure 1,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 partial four phase STG. It cannot be directly implemented by existing STG-based synthesis tools since the falling (reset) transitions of the signals are not specified. There are many different solutions for inserting falling signal transitions. Among them there are two extreme solutions: with maximum and minimum concurrency for falling transitions. The solution with minimum concurrency is not unique and depends upon the actual chosen reshuffling for falling transitions. Starting from the solution with maximum concurrency one can derive any other valid reshuffling of transitions by concurrency reduction. Figure 1,e shows an STG with maximal concurrency for all falling transitions assuming that all signals li, lo, ri, ro are independent and no interface constraints are 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 1,f presents a valid maximal concurrency handshake expansion for the LR-process taking interface constraints into account.

The corresponding State Graph (SG) is presented in Figure 3,a. This specification can be implemented with the current STG based synthesis tools. Two state signals are inserted in this example for resolving the Complete State Coding (CSC) conflicts. I.e., a pair of states labeled with the same binary code (e.g., 10*00, 1000*) require different next values for output signals lo and ro and hence must be distinguished by means of state signals (see Section 2.1).

Table 1 presents the area and performance results for different implementations of the LR-process. The row corresponding to the "Max. concurrency" corresponds to the implementation of the STG with maximum concurrency of the reset signal transition. The circuit area is 168 units, and two state signals have been inserted for resolving CSC conflicts. 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.

The well-known implementation of the LR-process is a Q-module [11] (also called a S-element [1]). It corresponds to the SG in Figure 3,b and STG in Figure 4,a. Its area is 104 units, while the critical cycle under the same assumptions is longer. This solution can be automatically obtained by our tool by reducing concurrency of signal transitions under the assumption that signal transition $lo+$ is late with respect to $ri-$. This assumption, if violated, affects only the performance, not the correctness of the implementation, since the circuit is synthesized to be speed-independent.

The cost function for concurrency reduction strives to minimize the number of CSC conflicts and to reduce the estimated complexity of the logic (cf. Section 6.2). Therefore, a complete reduction of concurrency under no constraints will give another solution – two wires, which is the best in terms of area (see Figures 3,c and 4,b). This solution however does not allow to decouple the left and the right neighbors of the LR-process, and hence would not be accepted as an efficient implementation in the four-phase expansion of the process, due to the increase in latency [18].

To get other solutions we perform concurrency reduction under concurrency constraints. We do not allow to reduce concurrency for any pair of events (a, b) which are declared to be maintained concurrent $(a \parallel b)$ (This definition can be naturally extended to signals, i.e. to any pair of events of the involved signals.) The results are reported in the last four lines of Table 1. The two best solutions, for $li \parallel ri$ and $lo \parallel ri$, corresponding to the SGs in Figure 3,d and c are reported in Figure 4,c and d. All the circuits have been obtained automatically with the tool `petrify`, by technology mapping into a library of two-input gates.

1.2 Algorithm overview

The overall algorithm for handshake expansion and reshuffling used in this example is shown in Figure 2. Line 1 of the algorithm corresponds to handshake expansion, line 4 corresponds to reshuffling.

A number of questions arise from this informal example. We will try to answer them in the rest of the paper.

	Area		Performance	
Circuit	area units	# CSC signals	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

- What is concurrency? (Section 2.3)
- How is concurrency exploited starting from a partial specification of an asynchronous controller? (Section 3)
- What are the valid reductions of concurrency? (Section 4)
- How can concurrency be reduced by iterative application of a single, elementary operation? (Section 5)
- How is the quality of the solution estimated? (Section 6)

In Section 7 we assess the effectiveness of the overall strategy by means of a set of experiments.

```
Inputs: Initial STG
        Interface constraints /* e.g., keep interleaving for channels */
        Concurrency constraints (concurrent events, late/early events)
Output:Reduced State graph and the corresponding STG
1:  Force handshake expansion of STG corresponding to maximal concurrency
    to satisfy all interface constraints
2:  Generate SG A by the STG
3:  while the best solution is not found and the search limit is not hit do
4:      Reduce concurrency of SG A,
        satisfying interface constraints and concurrency constraints
        reducing CSC conflicts and logic complexity
    endwhile
5:  Generate new STG for the best reduced SG
```

Fig. 2. Handshake expansion and reshuffling for STGs

Note that, as shown in the example, the method presented in this paper does not take into account exact delay information during the optimization process. Our goal is to work only at the level of causality constraints (as captured by the concurrency relation defined below), by taking into account rough performance information (such as signal transition $lo+$ being late with respect to $ri-$) as provided by the designer. More detailed interaction between logic optimization and delay is left to future analysis and development.

2 State Graphs and concurrency

In this section we introduce models and theoretical concepts required for our method: State Graphs, speed-independence and concurrency. We assume that the reader has a general understanding of Signal Transition Graphs [20, 5, 9] which will be used for informal illustration of the method.

2.1 State Graphs

A *State Graph* (SG) A is a labeled directed graph whose nodes are called *states*. Each arc of an SG is labeled with an *event*, that is a rising ($a+$) or falling ($a-$) transition of a signal a in the specified circuit. We also allow notation $a*$ if we are not specific about the direction of the signal transition. Each state is labeled with a vector of signal values (signals that can change in the state are marked with an asterisk). 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 3,a shows the SG for the Signal Transition Graph in Figure 1,f, which is consistent. The initial state of this SG is 0*000 (signal li can change from 0 to 1). We write $s \xrightarrow{a}$ ($s \xrightarrow{a} s'$) if there is an arc from state s (to state s') labeled with a and $s \overset{\tau}{\Rightarrow} s'$ is there is a path from state s to state s' labeled with a sequence of events τ.

In the following, we will use the overloaded notation $s \in A$, $s \xrightarrow{a} \in A$ and $s \xrightarrow{a} s' \in A$ to denote that a node or an arc belong to the graph A.

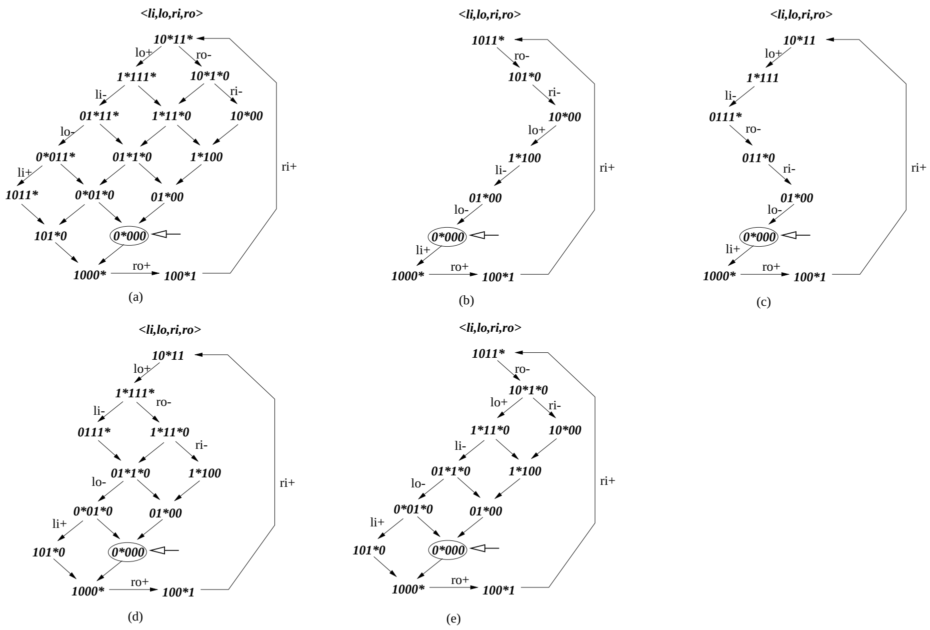

Fig. 3. SGs corresponding to different concurrency reductions

The set of all signals whose transitions label SG arcs are partitioned into a (possibly empty) set of *inputs*, which come from the environment, and a set of outputs or state signals that must be implemented. Outputs observable by the environment are called *external outputs*, while other outputs are called *internal outputs* or simply *internal signals*.

In addition to consistency, the following two properties of an SG are needed for implementability as a speed-independent logic circuit.

The first property is *speed-independence.* It consists of three constituents: determinism, commutativity and output-persistency. An SG is called *deterministic* if for each state s and each label a there can be at most one state s' such that $s \xrightarrow{a} s'$. A SG is called *commutative* if whenever two transitions can be executed from some state in any order, then their execution always leads to the same state, regardless of the order. An event a^* is called *persistent* in state s if it is enabled at s and remains enabled in any other state reachable from s by firing another event b^*. An SG is called *output-persistent* if its output signal events are persistent in all states. Any transformation (e.g., insertion of new signals for decomposition), if performed at the SG level, may affect all three properties.

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 in Figure 3,a is *output-persistent*, but it does not have CSC (cf. binary codes 10*00 and 1000* corresponding to two different states). The SGs in Figure 3,c and d have CSC.

2.2 Excitation Regions

A set of states is called an *excitation region* (ER) for event a^* (denoted by $ER_j(a^*)$) if it is a *maximal connected* set of states such that $\forall s \in ER_j(a^*)$: $s \xrightarrow{a^*}$. Since any event a^* can have several separated ERs, an index j is used to distinguish between *different connected occurrences* of a^* in the SG.

Similarly to ERs, we define *switching regions* (denoted by $SR_j(a^*)$) as connected sets of states reached *immediately after* the occurrence of an event.

A *minimal state* of an excitation region is a state without predecessors that belong to the excitation region.

2.3 Concurrency relation between events

Approximate identification of concurrency

Definition 1. Two events a and b are said to be concurrent in SG A if the following diamond structure of states and transitions belongs to A:

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

Definition 1 allows one to reason only approximately about concurrency. For example:

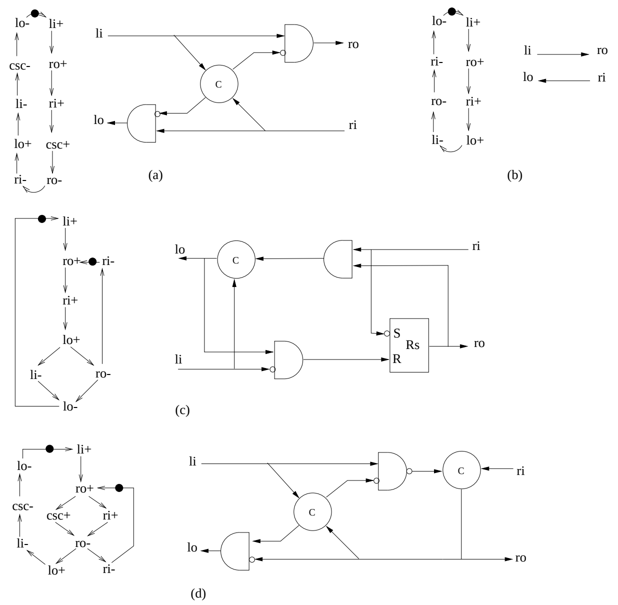

Fig. 4. Implementations of the LR-process

- it approximates concurrency for *sets of events* by using only pairwise concurrency. E.g., if the SG is not persistent, then $(a||b) \wedge (b||c) \wedge (a||c)$ does not necessarily imply $||(a, b, c)$. Consider, for example, an arbiter with three requesting inputs a, b, c and two available resources that can be granted concurrently, but not to all three requesters.
- instead of defining concurrency at the level of *event occurrences* (i.e., within an acyclic infinite structure, as discussed below), it defines it for all occurrences due to the same SG events. The distinction, although required in theory due to pathological examples, seems to be seldom necessary in practice.

If the SG is known to be speed-independent and a, b are *output* events, and hence output persistent and commutative, then concurrency of a and b can be checked just as intersection of their ERs:

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

In this paper we have chosen the approximate definition, because in our experience it works well in practice and is easily implementable. Moreover, concurrency reduction at the level of individual occurrences of events, as discussed in the next section, is much more expensive to implement, and hence can miss potentially useful optimizations due to excessive complexity.

Exact identification of concurrency Each event is represented by a single ER in an SG. However, not all states of this ER are potentially reachable in a single run of the process. The subset of reachable states can differ depending on the initial state of the system, on the path that is followed in the SG, and hence on which particular state is selected for entering the ER. Different *occurrences* of the same event are characterized by sub-regions of the ER reachable in a single run of the process. The concurrency relation can be therefore precisely captured only in terms of occurrences of events.

The theory of unfoldings for SGs [14, 7, 16] and for Petri Nets and STGs [13, 6, 8] has been developed for this purpose. An unfolding of an STG is an equivalent acyclic STG where different occurrences of an event are split and therefore represented by different ERs in the corresponding unfolding of an SG. Ordering relations for occurrences of events, including concurrency, can be easily derived by unfoldings [8].

Hence, we can precisely reason about concurrency reduction for occurrences of events in terms of unfoldings by comparing concurrent runs [19] (also known as Mazurkiewicz's traces [12]) of the initial specifications with interleaved traces of a reduced system.

Using unfoldings in a synthesis tool is feasible, but in this case it is more expensive and complex than using the SG. Even though the STG unfolding can be much more compact than a corresponding SG, performing operations such as reduction of concurrency on the *set of occurrences* of a pair of events *simultaneously and consistently* is much more complicated than performing them at the SG level.

3 Handshake expansion

This section explains how handshake expansion is performed. The syntax of our specifications allows 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.

3.1 Channels

In this section we use a notation similar to the one 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 transitions from $a?$ to $a_i\tilde{}$ and $a!$ to $a_o\tilde{}$, where the suffix $\tilde{}$ denotes a toggle transition of the signal. The process of determining the binary encoding for each state will finally deduce whether the actual transition is rising or falling.

The expansion to a 4-phase protocol is performed by relabeling transitions and inserting return-to-zero events. The transformations performed at Petri net level consist in 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.a. 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 acknowledgement ($a!$ 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 completed 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 and 5.e show how channel events are translated into actual signal events by simple structural transformations on the Petri net. Each $a?$ event 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 STG pieces of Figures 5.a.d.e gives an overall picture of channel behavior in set and reset phases. Note that the specification must properly interleave the events on the channel according to the handshake protocol, otherwise the expansion may 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 6.1.

3.2 Partially specified signals

We describe here the transformations required to expand partially specified signals by adding the return-to-zero transitions. The Petri net structure added to the specification is shown in Figure 5.b. The transformation of events is shown in Figure 5.c. Each signal event in the specification corresponds to a rising transition of the same signal. Each rising transition is enabled only when the return-to-zero transition has been fired (arc $\mathtt{rdy} \rightarrow b+$). The return-to-zero transition is enabled as soon as the rising transition has been fired (arc $b+ \rightarrow \mathtt{rtz}$).

3.3 Example

Figure 6 presents an example illustrating all the previous transformations. The original specification (Figure 6.a) has a channel (a), a partially specified signal (b) and a completely specified signal (c).

A 2-phase refinement of the same specification is shown in Figure 6.b. In this case, no return-to-zero transitions are inserted and the events of signal b are transformed into alternating rising and falling transitions. Similarly, channel events are expanded into signal transitions with alternating phases.

The 4-phase refinement is shown in Figure 6.c (This picture has actually been obtained automatically by `petrify` after re-synthesizing a new Petri net from

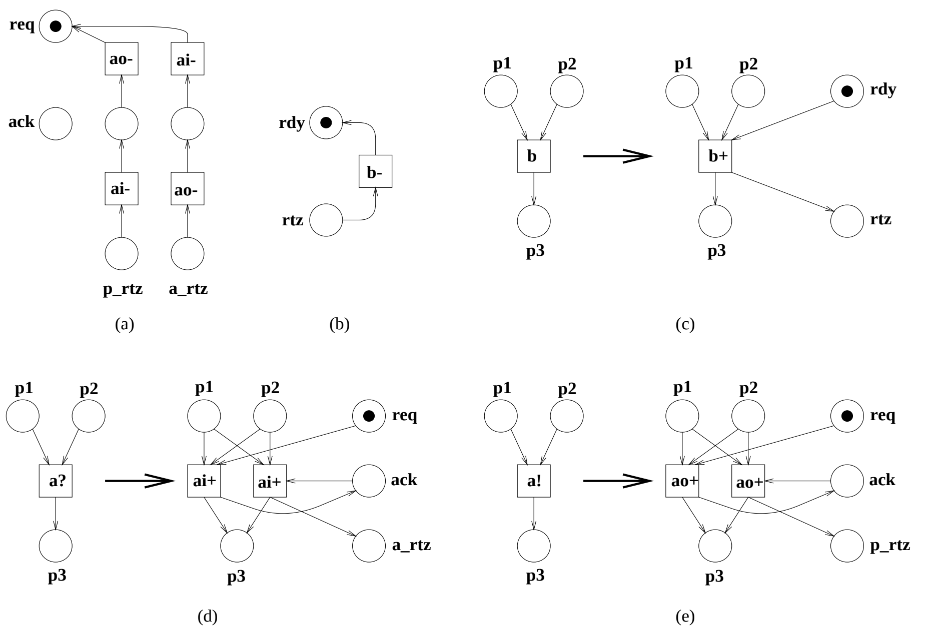

Fig. 5. Petri net structures for return-to-zero transitions in 4-phase refinement: (a) for channels and (b) for partially specified signals. Event transformations: (c) for partially specified signals, (d) for input channel events and (e) for output channel events.

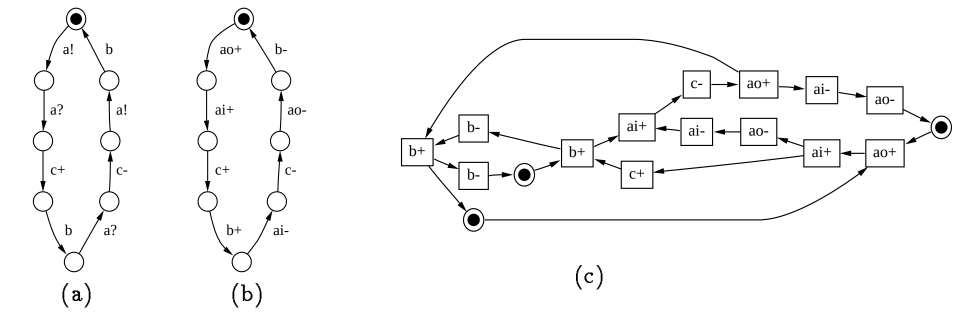

Fig. 6. (a) Original specification (state graph), (b) 2-phase refinement (state graph), (c) 4-phase refinement (Petri net).

the one derived by applying the structural transformations for 4-phase refinement [2].) All the events of signal b are now transformed into rising transitions with an automatic insertion of return-to-zero transitions. Events of channel a are also transformed into rising transitions of signals a_i and a_o. All return-to-zero transitions are incorporated with maximum concurrency.

4 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 base. 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.

4.1 Validity

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. As a result of arc removal some of the states may become unreachable and should be removed from the SG.

No new states or new 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 of the conflicts or all of them can disappear due to state removal).

Validity includes the following properties:

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 (Both I/O 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.):
   (a) No signal transitions of input signals can be 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 $s_0 \overset{\tau}{\Rightarrow} s_0'$ or $s_0' \overset{\tau}{\Rightarrow} s_0$ in A such that sequence τ contains only events of *internal* signals, not observable by the environment.
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: $s \overset{e}{\rightarrow} \in A$), then there exists some other event e' such that: $s \overset{e'}{\rightarrow} \in A$ and $s \overset{e'}{\rightarrow} \in A_{red}$.

Definition 2 Valid reduction. If a reduced SG satisfies all properties (1)–(4) above, then we say that concurrency reduction is valid.

4.2 Invalid reductions

In this section we show by examples how each of the properties (1)-(4) from Definition 2 can be violated during concurrency reduction.

Violating output persistency Figure 7 shows that persistency for an output event (event a in the example), can be violated, if after reduction of the $ER(a)$ disabling of the event becomes possible. It can happen if excitation of a can occur in a state preceding the states with reduced concurrency.

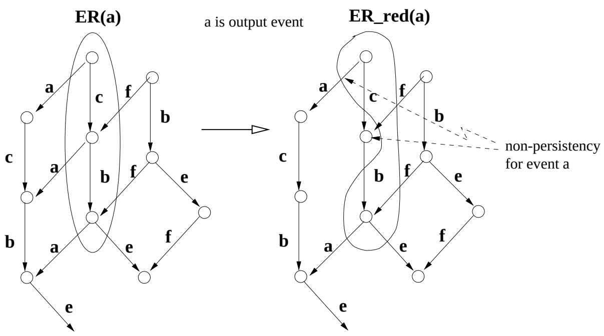

Fig. 7. Invalid reduction: output persistency violation

Delaying input events Figure 8 shows another invalid reduction of concurrency. Input event $ri-$ is delayed until output event $lo+$ fires (it is not excited in the top right state 1010). This creates an additional requirement for the active channel r: the external module observing channel r must also observe passive channel l and reset $ri-$ only after $lo+$. Hence, the active channel r is not decoupled from the passive channel l. This means that the environment cannot be separately synthesized for both sides of the component, thus defeating the purpose of using handshake components.

Changing the initial state Figure 9 shows an incorrect reduction which leads to the removal of the initial state. In such situation the initial state has to be moved to one of the closest predecessors (state 0011), or successors (state 1000). If the new initial state is equivalent to the old one with respect to interface (input/external output) signals, then it is valid. However, in the example of Figure 9, the shift backward requires to change the phase of the active channel r (the initial value of ro becomes 1 instead of 0), while the shift forward requires to change passive channel l into an active channel (the first event occurs on lo instead of li).

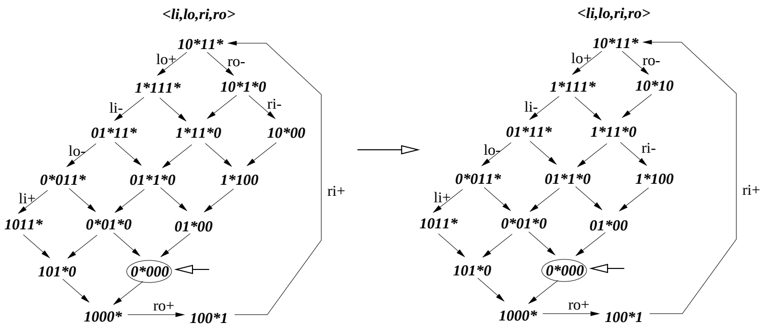

Fig. 8. Invalid reduction: input event $ri-$ is delayed until output event $lo+$ fires.

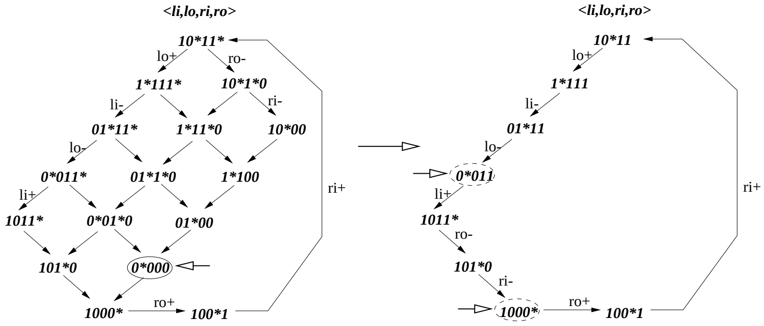

Fig. 9. Invalid reduction: initial state changes with respect to the environment.

Disappearance of events Figure 10 shows how events can disappear in concurrency reduction: state s becomes unreachable after removing transition $s_0 \xrightarrow{a} s$. This state, however, is the only one which belongs to $ER(i)$. After its removal $ER(i)$ will disappear together with all its successors. The condition to check is simple: each reduced ER should not become empty.

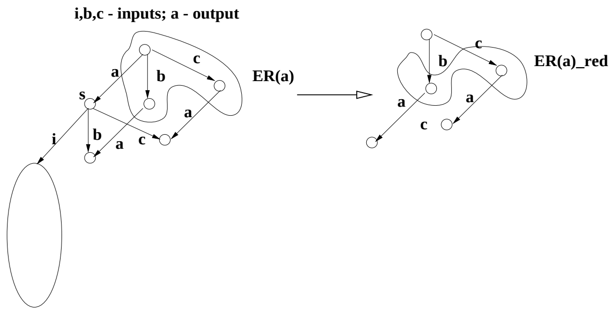

Fig. 10. Invalid reduction: events disappear.

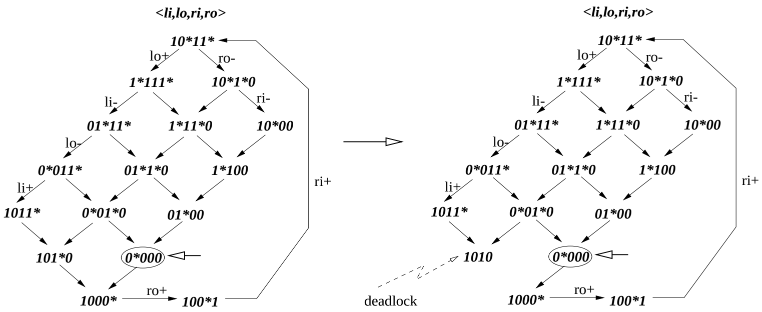

Fig. 11. Invalid reduction: new deadlocks appear.

New deadlocks Assume, for the sake of argument, that the designer allowed reducing concurrency for input signal transition $ri-$ at the price of changing the environment. Then one of the possible reductions is shown in Figure 11. Since $ri-$ is an input event, this reduction does not imply output non-persistency. However, since the last state of $ER(ri-)$ is removed, this reduction converts state 1010 into a deadlock state, which is not acceptable.

5 The basic operation: forward reduction

The algorithm sketched in Figure 12 defines our basic operation for concurrency reduction which is 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, `FwdRed(a,b)` creates an arc between event b and event a at the STG level.

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

Fig. 12. Reduction of concurrency for output event a by event b.

The application of the forward concurrency reduction `FwdRed` to an STG with choice (non-persistency) and concurrency is illustrated in Figure 13. At the first step the algorithm removes two arcs labeled with a from state s_2 (since $s_2 \in ER(a) \cap ER(b)$) and from state s_1 (since $s_1 \in backward\_reachable(ER(a), ER(a) \cap ER(b))$) due to line 2 of the algorithm. At the next stage we iteratively remove all states that become unreachable, together with their fan-out arcs. 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 line 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.

In this case, the multiple reduction is due to the need to preserve consistency and speed-independence of the SG. Initially a occurs in parallel with either d or b. Simply adding a constraint from b to a would stop a from occurring in the other case. The simplest, persistent way to avoid this problem is to make a follow *both* b and d. This is taken care of *automatically* by our conditions.

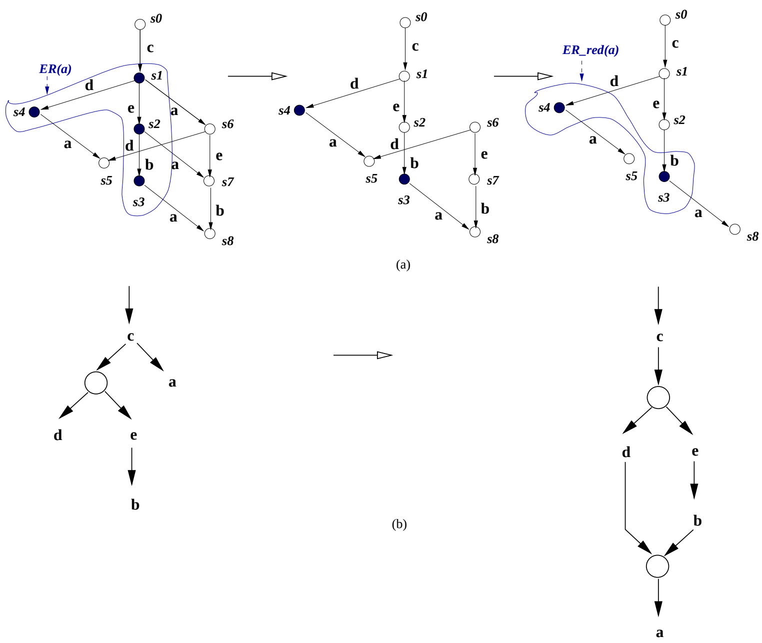

Fig. 13. Applying `FwdRed(a,b)` to SG (a) and the corresponding STG transformation (b).

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

Proposition 3 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.*

5.1 Completeness of concurrency reduction

An application of `FwdRed(a,b)` completely reduces any concurrency between events a and b. Hence this operation is incomplete (too coarse) because not every valid reduction can be obtained by its iterative application. Excluding the concurrent firing of a and b only in some states (not totally) cannot be achieved by `FwdRed(a,b)`.

The most general formulation of concurrency reduction can be done via the removal of a single arc from the corresponding SG. Let us consider an arbitrary arc $s \xrightarrow{a}$ in SG A. A *backward reduction* with respect to this arc (denoted by `BkdRed(a,s)`) is an operation that disables event a in state s (note that, in order to preserve the I/O interface, event a should be an output event). However, speed-independence of the reduced SG can be preserved only if event a is disabled also in any predecessor of s in $ER(a)$.

A formal definition of `BkdRed(a,s)` can be obtained by replacing line 2 in Figure 12 with
$ER_{red}(a) = ER(a) - (\{s\} \cup backward\_reachable(ER(a), \{s\}))$
and by adding a check for the appearance of new deadlock states, since now that part of Proposition 3 is no longer satisfied.

While `FwdRed(a,b)` can be modeled by a set of the `BkdRed(a,s)` reductions applied for every $s \in ER(a) \cap ER(b)$, the converse is not true. This fact is illustrated by Figure 14, where `BkdRed(b+,1000)` is applied to the SG in Figure 14,b. Disabling $b+$ in state 1000 requires its disabling in the initial state 0000 as well. Hence states 0100 and 1100 will become unreachable in the reduced SG (shown in Figure 14,c). A change diagram specification (an STG-like specification allowing OR-causality between events) corresponding to the reduced SG shows OR-causality between events $d+, c+$ and $b+$. This result cannot be achieved by any set of forward reductions.

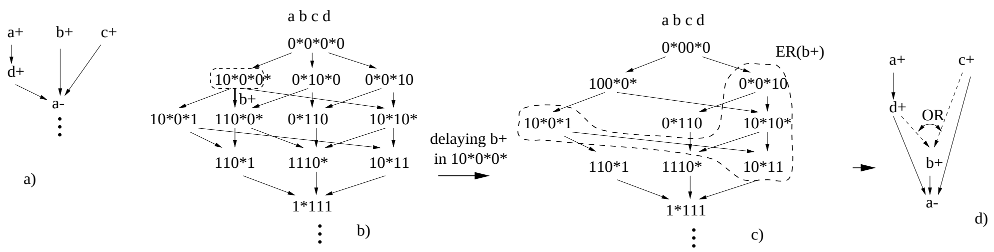

Fig. 14. Applying `BkdRed(b+,1000)` to SG (b)

Since `BkdRed(a,s)` allows one to exclude any feasible trace from the SG, this operation is complete in a sense that every valid reduction of a SG can be obtained by its iterative application. However, contrary to `FwdRed(a,b)`, `BkdRed(a,s)` in general does not have a clear interpretation in terms of ordering relations between events. Therefore, our practical implementation (Section 6) is restricted to applications of `FwdRed`.

6 Implementation

This section describes the actual strategy implemented for concurrency reduction in the tool `petrify`.

As we mentioned in Section 1 and we will experimentally illustrate in Section 7, 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 (or totally eliminated, if the reduced SG has CSC). Secondly, the number of reachable states is reduced, and hence the don't care set that can be exploited by logic minimization is increased. However, in this case one must also consider the potential increase in the number of *"trigger"* signals for each output. (Loosely speaking, these are the signals that directly enable a transition of the output, and that *must* be part of any speed-independent implementation.) 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.

6.1 Concurrency reduction

The algorithm in Figure 15 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 and 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 any of the configurations in `frontier`.

6.2 Cost function

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

```
Inputs:  State graph initial_SG
         Keep_Conc ⊆ E × E (set of pairs of events whose concurrency must be preserved)
         size_frontier: maximum size of the frontier for exploration
Output:  State graph reduced_SG with reduced concurrency

frontier = explored_SGs = {initial_SG};
while frontier ≠ ∅ do
    new_solutions = ∅;
    foreach SG ∈ frontier do
        foreach (e1, e2) such that e1 || e2, e2 is not an input event
                and (e1, e2) ∉ Keep_Conc do
            new_SG = FwdRed (SG,e2,e1);   /* Add causality e1 → e2 */
            explored_solutions = explored_solutions ∪ {new_SG};
            new_solutions = new_solutions ∪ {new_SG};
        endfor
    endfor;
    frontier = "the best size_frontier elements in new_solutions";
endwhile;
reduced_SG = "best element in explored_SGs";
```

Fig. 15. Algorithm for reducing concurrency.

of the logic. The cost function, $C(SG)$, for each SG obtained by reducing the concurrency from initial_SG is as follows:

$$C(SG) = W \cdot \Delta csc(SG) + (1 - W) \cdot \Delta logic(SG)$$

where W is a real number between 0 and 1 given as a parameter by the designer. W defines the trade-off between biasing the heuristic search towards reducing CSC conflicts ($W \rightsquigarrow 0$) or reducing estimated complexity of the logic ($W \rightsquigarrow 1$).

The function Δcsc indicates the progress of the new SG towards solving CSC conflicts. It returns a value between 0 and 1. The value 1 indicates that all CSC conflicts have been solved (this function is not used when the original SG has no CSC conflicts).

$$\Delta csc(SG) = 1 - \frac{\text{CSC conflicts}(SG)}{\text{CSC conflicts}(\texttt{initial\_SG})}$$

The function $\Delta logic$ indicates the progress in reducing the estimated complexity of the logic. It returns the value 1 when no gates are required to implement the logic (only wires), the value 0 when no reduction is achieved and a negative value if the estimated logic is more complex than the one of the initial_SG:

$$\Delta logic(SG) = 1 - \frac{logic(SG)}{logic(\texttt{initial\_SG})}$$

Here $logic(SG)$ is the sum of the estimated complexity (in number of literals) of each output signal. The accuracy of this estimation is compromised when CSC conflicts still remain in the SG due to the improper definition of the on-set and off-set of the signals. Improving this estimation is one of the aspects that require further investigation in the future.

7 Experimental results

7.1 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 [1].

Figure 16.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 purpose of our CAD strategy is to assist the designer in the selection of the best implementation according to the requirements of the application.

Figure 16.e depicts a 2-phase refinement of the specification, obtained by merely considering each event of the original specification as a transition on the corresponding wire. An implementation of such behavior is shown in Figure 16.i.

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. Figures 16.c [1] and 16.d [18] (corresponding to implementations in Figures 16.g and 16.h respectively) have been obtained manually. The latter is the one actually used by the current Tangram compiler.

Our tool can automatically perform a 4-phase expansion by using the structural Petri net techniques discussed in Section 3, and derive the specification shown in Figure 16.b. After this transformation, the return-to-zero signalling is performed with maximum concurrency with regard to the rest of events. However, a direct implementation of this behavior may result in a complex circuit due to the need of inserting extra logic for state encoding and logic decomposition. An actual implementation for maximum concurrency has more than twice the complexity of any of the circuits shown in Figure 16.

Figures 16.f and 16.j depict the solution automatically obtained by reducing the concurrency of the 4-phase refinement in Figure 16.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 c. The circuit is slightly smaller (by 12% in our standard cell library) than those obtained by manual design, although its estimated performance may be worse than that of Figure 16.h, if b and c have balanced delays (the critical cycle is longer by 11% under the assumption: delay of a combinational gate – 1 time unit, delay of a sequential gate – 1.5 time units, and delay of an input event – 3 time units).

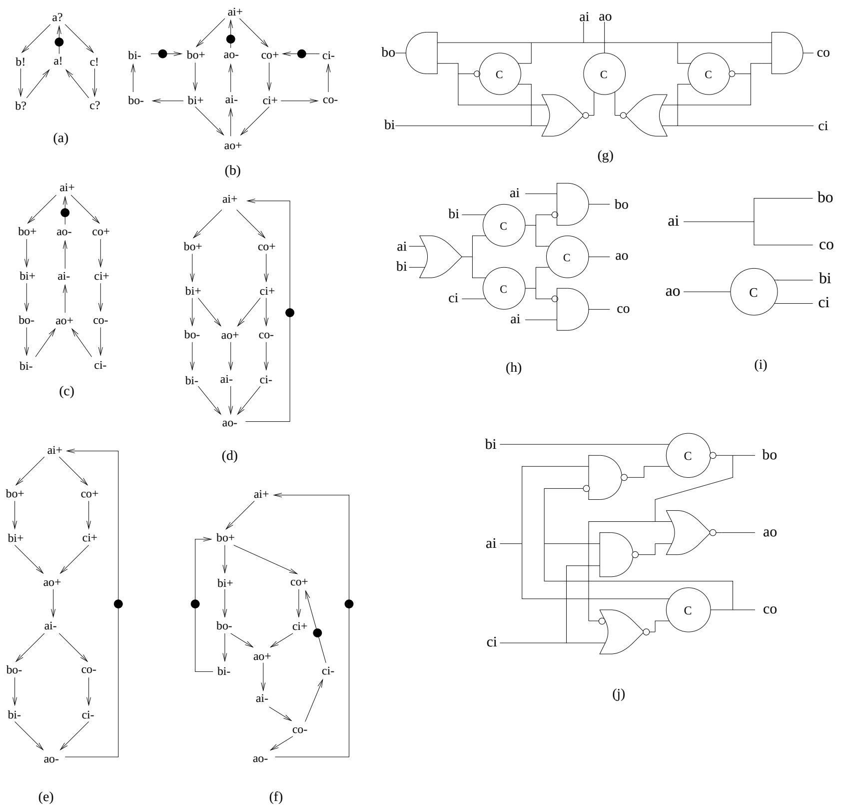

Fig. 16. Different specifications and implementations for a PAR component.

7.2 Second case study: MMU control

In [15] it was shown that by timing optimization using known timing constraints on the behavior of the environment it is possible to reduce the area of the MMU-control circuit by over 50 percent 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 both techniques, our high-level transformation and Myers' lower level timing optimizations, can conceivably provide even better optimization results, since they exploit different sorts of flexibility (don't cares due to externally irrelevant transitions, and don't cares due to timing respectively). All the numbers are reported for speed-independent implementation by using a library of two-input gates.

We can conclude that

- With respect to the original solution, reshuffling can yield an area reduction to less than one half.

Circuit	Area		Performance	
	area units	# CSC signals	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-control

- This area gain 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 [15]. In case [15] used a finite delay interval, we considered the average delay, while in case the upper bound was infinite, we considered the lower bound.

7.3 Completely specified STGs

Two types of experiments were performed to evaluate the influence of reducing concurrency on the quality of the circuits in case the initial STG is already completely specified and there is no freedom in increasing concurrency for reset transitions. In the first experiment, we selected a set of specifications without CSC conflicts. In the second experiment, we selected some specifications with CSC conflicts. In all cases, specifications without concurrency were discarded. For each example, three circuits were derived: without any concurrency reduction (`ini`), by allowing to reduce concurrency without any constraint, but preserving the I/O interface (`full`) and by assuming a slow environment and preserving the pairwise concurrency of input events (`env`). The results are presented in Table 3.

In all cases we used the parameter values $W = 0.5$ and `size_frontier`$=3$ (Section 6.1). 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 [3].

The number of states of the specifications and the area of the circuit before and after concurrency reduction are reported. We also report in Table 3.b (in parentheses) the number of state signals required to solve CSC after concurrency reduction.

The results in Table 3.a indicate that significant savings in complexity of the circuit can be obtained: about 20% when preserving the concurrency of the input events and about 50% when no constraints are imposed to reduce concurrency. In some cases (nowick and vbe6a) the complexity of the circuit increases. This corroborates the observation that reducing concurrency does not always result in simpler circuits, even though the dc-set increases. It also shows the inaccuracy of the cost function when estimating logic in terms of number of

Circuit	States ini	env	full	Area ini	env	full
chu133	24	23	16	200	176	176
chu150	26	26	20	144	144	112
master-read	2614	1218	100	496	456	216
mmu	280	189	36	768	560	96
mp-forward-pkt	22	20	20	224	200	200
mr1	244	83	27	560	400	304
nak-pa	58	31	22	304	224	104
nowick	20	19	19	216	240	240
ram-read-sbuf	39	37	28	352	304	208
sbuf-ram-write	64	44	35	408	408	296
sbuf-send-ctl	27	25	25	336	240	240
trimos-send	336	132	53	576	352	184
vbe5b	24	22	12	184	152	16
vbe6a	192	112	20	416	472	280
vbe10b	256	124	22	568	488	296
wrdatab	216	204	39	688	480	248
Total	4442	2309	494	6440	5296	3216

(a)

Circuit	States (CSC signals) ini	env	full	Area ini	env	full
adfast	44 (2)	30 (0)	12 (0)	232	272	16
nak-pa	56 (1)	35 (1)	18 (1)	304	224	152
nowick	18 (1)	17 (1)	17 (1)	208	208	208
ram-read-sbuf	36 (1)	32 (0)	25 (0)	288	304	208
sbuf-ram-write	58 (2)	38 (1)	29 (1)	384	296	280
sbuf-read-ctl	14 (1)	12 (1)	12 (1)	240	208	208
mmu	174 (3)	96 (0)	40 (0)	768	472	200
mr0	302 (3)	94 (1)	22 (2)	600	584	240
mr1	190 (3)	72 (0)	22 (0)	560	368	104
par_4	628 (4)	274 (1)	22 (2)	648	432	400
vme-read	90 (2)	76 (1)	37 (0)	368	312	168
vme-write	236 (2)	139 (2)	63 (0)	408	376	232
Total	1846 (25)	915 (9)	319 (8)	5008	4056	2416

(b)

Table 3. Experimental results: (a) without CSC conflicts, (b) with CSC conflicts.

literals, with respect to the final cost of the implementation (after decomposition and mapping). The designer should select, by interacting with the tool, the most appropriate implementation in each case.

Table 3.b illustrates the fact that concurrency reduction may simplify the task of solving CSC conflicts and, indirectly, the complexity of the circuit as the number of non-input signals decreases. In two cases (mr0 and par_4), the resulting SG after full reduction of concurrency requires more state signals than the SG after a constrained reduction. This illustrates the fact that the cost function should not only aimed at reducing CSC conflicts.

8 Conclusions

Specifying the behavior of an asynchronous system is a complex task that often requires a certain level of abstraction from the point of view of the designer.

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 the circuit synthesis, concerning phase refinements and event reshuffling. Thus, the designer is only left the intellectually challenging task of defining the causality among actions and specifying the desired concurrency in the system. The tedious 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.

Acknowledgements. We thank Steve Furber for emphasizing the need to tackle the problem of automatic handshake expansion and concurrency reduction. This work was supported by the following grants: ESPRIT ACiD-WG (21949), Ministry of Education of Spain (CICYT TIC 95-0419), UK EPSRC GR/K70175 and GR/L24038, and British Council (Spain) Acciones Integradas Programme (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. J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri nets from state-based models. In *Proc. of ICCAD'95*, pages 164–171, November 1995.
3. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev. Decomposition and technology mapping of speed-independent circuits using boolean relations. In *Proceedings of the International Conference on Computer-Aided Design*, November 1997.
4. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Trans. Inf. and Syst.*, E80-D(3):315–325, March 1997. Petrify can be downloaded from URL: http://www.ac.upc.es/vlsi/
5. T.-A. Chu. On the Models for Designing VLSI Asynchronous Digital Systems *Integration: the VLSI journal*, 4: 99–113 (1986).
6. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Lecture Notes in Computer Science*, 1055:87–106, 1996.
7. J. Gunawardena. On the causal structure of the Muller unfolding. Technical Report STAN-CS-93-1466, Dept. of Comp. Sci., Stanford University, March 1993.
8. A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten. A structural approach for the analysis of Petri nets by reduced unfoldings. In *International Conference on Application and Theory of Petri Nets, Proc. 17th Int. Conference*, volume 1091

of *Lecture Notes in Computer Science*, pages 346–365, Osaka, Japan, June 1996. Springer-Verlag.

9. L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits*. Kluwer AP, Boston, 1993.
10. 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.
11. 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.
12. A. Mazurkiewicz. Concurrency, modularity and synchronization. In *Lecture Notes in Computer Science, Vol. 379*. Springer-Verlag, 1989.
13. K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.
14. D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.
15. Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.
16. M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.
17. M. A. Peña and J. Cortadella. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.
18. Ad Peeters. Implementation of a parallel component in tangram. Personal communication, 1997.
19. W. Reisig. Formal methods for concurrent systems design: A survey. In *Proceedings of the Conference on Programming Models for Massively Parallel Computers*, pages 12–23. December 1993.
20. L.Y. Rosenblum and A.V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets, Torino, Italy*, pages 199–207, July 1985.
21. P. Vanbekbergen. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *Proceedings of the International Conference on Computer-Aided Design*, pages 184–187, November 1990.
22. C. Ykman-Couvreur, P. Vanbekbergen, and B. Lin. Concurrency reduction transformations on state graphs for asynchronous circuit synthesis. In *Proceedings of the International Workshop on Logic Synthesis*, May 1993.
23. Ch. Ykman-Couvreur, P. Vanbekbergen, and B. Lin. Concurrency reduction transformations on state graphs for asynchronous circuit synthesis. In *Proceedings of the International Workshop on Logic Synthesis*, May 1993.