Designing Control Logic for Counterflow Pipeline Processor Using Petri Nets [*]

A. Yakovlev
Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU England

May 3, 1995

Abstract

This paper approaches the problem of implementing an asynchronous control for a stage of the Sproull Counterflow pipeline processor (CFPP) as an exercise in combining two synthesis techniques recently developed for Petri nets. We first synthesise a number of Petri net models of the CFPP stage control from its original "five-state-five-event" description due to C. Molnar. Secondly, we implement these net models in asynchronous circuits, using two-phase and four-phase components. The latter stage involves synthesising circuits with arbitration elements from behavioural descriptions with internal conflicts. This exercise appears to be quite instructive in the sense that it helps to estimate the scope and power of formal methods and today's automatic tools in assisting the process of asynchronous design.

Keywords: *arbitration, asynchronous circuit, counterflow pipeline processor, design automation tool, event-based signalling, micropipeline, Petri net, signal transition graph, synthesis.*

1 Introduction

Asynchronous design technology is getting more mature both in actual designing industrial strength circuits and developing design tools. Two recent processor design projects, the Amulet1 microprocessor [1] and Sproull's counterflow pipeline processor (CFPP) [2], have drawn attention of a much wider audience than what used to be a traditionally small "asynchronous club". On the tools front, there has also been much progress in the last five years. Amongst at least a dozen of existing software packages are such systems as Tangram [3], supporting syntax-driven design from high-level programming specifications, and SIS [4] and FORCAGE [5], supporting circuit synthesis from interpreted Petri nets and their "close relative", Change Diagrams. The FORCAGE system also provides tools for verification of speed-independence conditions in asynchronous designs.

There is still much to be done for the tools to enable practical circuit designers benefit from them in their everyday experience. The major shortcomings of the existing tools are following. Firstly, they are usually good in simple routine operations, such translating high-level behavioural descriptions into specially structured circuits, e.g., converting Tangram CSP-like expressions into interconnections of handshake components. The resulting circuits can often be inefficient, both in speed and in size. Secondly, the synthesis-oriented tools are capable of synthesising only from

[*]Supported by EPSRC grant GR/J52327

specifications which are special classes of state-graphs (semi-modular) and Petri nets (free-choice and safe, or non-choice) and of a fairly limited size. For example, today's tools do not allow the designer to synthesise circuits with arbitration unless the designer uses special "tricks", combining two approaches, partly manual and partly automated.

There has been some initial work on the methods that extend the class of specifications, to allow designing circuits with arbitration components [6]. This work (a) needs further formalisation and automation, and (b) it is limited to a particular modelling framework, all transformations must be carried out at the Petri net level. Both these issues can be resolved independently, and the latter one can possibly benefit from the recent developments in the area of automated synthesis of Petri nets from state-based models.

Indeed, as can be seen in the model of a CFPP stage control circuit, devised by Charles Molnar [2], the designer may find it easier to define the behaviour in a state-transition form. The stage control model is the one with an essential arbitration paradigm. Originally, it looked doubtful that circuit synthesis techniques available for Petri nets [6] could be directly applied to it. The way from the specification to the circuit, as outlined in [2], was paved by manual effort. For example, the most crucial part of this design was a structural decomposition of a stage into an inter-stage arbiter (called "cop") and the remaining stage circuitry. That has obviously been one of the ways (apparently a very successful one!) to pursue the design. It would however probably be desirable to use a more formal technique that would allow a set of transformations at the behavioural level, in which this design would be a natural option from the synthesis process. Such a wish creates the major goal of this paper.

The paper demonstrates the combined use of the following constituents:

1. Equivalent transformations at the state-transition level, which are aimed at obtaining a state graph in such a form that can be converted into a behaviourally equivalent Petri net.

2. Synthesis of a Petri net from the state graph [7]; the net must satisfy the requirements of subsequent circuit synthesis [6].

3. Synthesis of a circuit in one of the two potential technologies. The first one is a two-phase circuit consisting of special (micropipeline) elements. Such a circuit can often be obtained by a relatively straightforward conversion of the Petri net (almost similar to a syntax-driven approach of Tangram). The second possibility, quite a challenge for today's synthesis tools, is to refine the net into a Signal Transition Graph (using the so-called "signalling expansion" [8]) and perform logic synthesis using one of the STG-based tools (e.g., SIS).

These steps are not fully automated as yet but there is a good indication that design examples like this one with CFPP create a very good motivation and provide guidance for further work on tools. For example, a new tool, called petrify, whose original version has been developed by Jordi Cortadella on the ideas of [7], already supports synthesis of Petri nets from state graphs and equivalent transformations at the Petri net level. In fact, the most recent version of petrify has helped to obtain the Petri net models shown in Figure 4, c and d, which lead to the circuit shown in Figure 11. Those synthesised originally by hand had some redundant places and arcs.

Hopefully, petrify will eventually provide an important link between circuit compilation tools (e.g., TANGRAM) and circuit synthesis and verification tools (e.g., SIS and FORCAGE).

The paper is organised as follows. Section 2 introduces the description of the CFPP stage control circuit and formulates the problem. Section 3 describes the procedure to synthesise Petri net specifications from state-based models. Section 4 demonstrates the application of this procedure to the state-based description of the CFPP stage control. Section 5 presents implementations of the Petri net models of the CFPP stage control. Finally, Section 6 outlines directions of future work and draws conclusions.

2 CFPP stage control circuit. Original description

For a complete description of the CFPP architecture we refer the reader to [2]. Here, we would like to abstract away from the details of instruction execution in the CFPP, and only concentrate on the issue of the behavioural specification of control in a basic stage of the CFPP.

The overall organisation of control in a CFPP is as follows. There are two mutually synchronised pipelines, one for instructions and the other for results, where the results are used by instructions and may be produced or updated by them. These pipelines allow instructions and results to propagate in opposite directions, each of them operating as an ordinary pipeline with data items passing between any pair of adjacent stages if one of the stages is empty and the preceding stages holds a datum. Here, the role of data items is played by instructions, in the instruction pipe, and by results, in the result pipe.

Mutual synchronisation between the two pipes is essential for the functionality of the CFPP. The following important requirement is imposed on such a synchronisation: *for every instruction I, entering the instruction pipe from the bottom (by convention, instructions flow "bottom-up"), and every result R, entering the pipe at the top (results flow "top-down") while the instruction I is already in the pipe, there must be an opportunity to match in one of the stages (the matching process, including potential execution if the address of the operand in I matches the one in R, is called* garnering*).*

Abiding by the above requirement, instructions and results happenning to cohabit in the counterflow pipeline must never miss each other. This requirement is met by organising the pairs of adjacent stages in such a way that the states of control in these stages prevents certain items from advancing along their pipes until the garnering process has been accomplished.

Figure 1 shows Molnar's state diagram of a pipeline stage control. The states have the following meaning:

E: **Empty.** Neither instruction nor result is present.

I: **Instruction.** Only an instruction is present.

R: **Result.** Only a result is present.

F: **Full.** Both instruction and result have arrived.

C: **Complete.** The CFPP execution rules [2] have been enforced, and both instruction and result are free to move on [1].

The transitions in this state graph that involve motion of instructions and results are labelled AI (accept instruction from below), PI (pass instruction upward), AR (accept result from above), PR (pass result downward), and G (perform garnering, which is either executing the instruction if its operand matches the result or release both instruction and result).

Observing the state graph, we may note that there are two states in which dynamic arbitration may take place. First, this is state I, where *either* instruction may be passed before a result may

[1] As was noted in [2], in practice this state might be divided further to allow the result to advance while the instruction is being executed. We, however, abstract away from such distinctions in this paper.

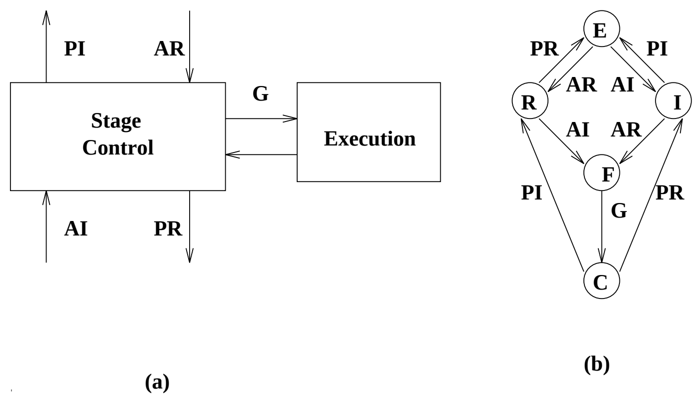

Figure 1: Counterflow pipeline stage control: (a) structural view, (b) state diagram

arrive in the stage *or* result may arrive before instruction is allowed to leave the stage. Similarly, in state R, *either* result may be passed before an instruction may arrive in the state *or* instruction may arrive before result is allowed to leave the stage.

The presentation in [2] "jumped" from definition of this state diagram directly to its structural and circuit implementation. Our task here is to demonstrate the process of deriving the specification in the form of a Petri net with signal events represented *uniquely*, which would be more amenable to formal transformations. The latter would bring us to a circuit solution, or a set of solutions, with standard arbitration elements (e.g., a 2-way mutual exclusion element), following the technique described in [6].

First, we need to review some important results on synthesis of Petri nets from transition systems (i.e., state graphs). The major theoretical background can be found in [9]. The adaptation of those results to a practical procedure of synthesising nets from state-based models was made in [7].

3 Synthesis of Petri nets from transition systems

Labelled Petri nets. The target of our specification synthesis is a labelled Petri net. We assume that the reader is familiar with the basic terminology of Petri nets [12], and give here only a brief outline of the most relevant issues.

A Petri net (PN) is a directed graph consisting of two types of vertices, *places* and *transitions*, connected by arcs, called *flow relation*, in such a way that arcs between places or between transitions are not allowed [2]. In a marked net, a special subset of places, marked with tokens (black dots), is called the initial marking of the net. A transition is *enabled* in a marking if all its input places are marked. An enabled transition may fire, producing a new marking (this marking is said to be *directly reachable* from the previous one) with one less token in each input place and one more token in each output place of the transition. The set of all markings *reachable* (ordinary transitive closure of the direct reachability) from the initial one is called the net's Reachability Set. The graph whose vertices are the net's markings and arcs correspond to the direct reachability relation is called the Reachability Graph of the net.

[2] We shall sometimes abuse this standard notation by allowing two transitions to be connected by an arc directly – this arc would stand for a place with exactly one input and one output arcs in a standard form. The "overloaded" arc thus becomes a carrier for tokens.

A labelled PN is a PN in which every transition is labelled with a symbol, called *label*, from a given alphabet. In the case of *unique labelling*, i.e., if no two transitions have the same label, each transition in the net can be uniquely identified by its label. In such a case we can use the label as the transition's name.

A PN is called *safe* if no more than one token can appear in a place. A PN is called *pure* if no pair of a place and transition are connected by mutually opposite arcs (bi-directional arcs are often used to represent such *self-loops* in PNs). A PN is called *simple* if no two transitions have the same sets of input and output places.

Transition systems. A *transition system* (TS) is a directed state graph in which every arc connecting a pair of states is labelled with a name of an event from a specific event alphabet. Such a labelled arc is called *transition*. One state is marked as the initial state. Any TS must satisfy the following *basic conditions* [9]:

A1. No self-loops, that is no transition may begin and end in the same state.

A2. No multiple arcs between a pair of states.

A3. Every event must have some occurrence.

A4. Every state is reachable from the initial state.

The basic intuitive idea behind the construction of a Petri net whose behaviour is equivalent [3] to the original TS is a correspondence between subsets of states, called *regions*, and places in the synthesised net. This allows a 1-1 correspondence between states of a region and markings of the Petri net in which the place corresponding to the region has a token.

More specifically, a region is a subset of states with which *all* transitions labelled with the same event e have exactly the same "entry/exit" relationship. Namely, we say that a subset of states r is *entered* by event e if for every transition labelled with e the source state does not belong to r while the destination state is in r. Similarly, r is *exited* by e if for every e-labelled transition the source state is in R while the destination is outside r. In the remaining cases, e is said to be *non-crossing*, either *internal* or *external*, event for a region. Thus, to become a region, a subset r must satisfy exactly one of the three cases for every event e: (i) r is entered by e, (ii) r is exited by e, and (iii) r is not crossed by e.

A region r is a *pre-region* (*post-region*) of an event e if r is exited (entered) by e.

Figure 2 illustrates a pair of regions, $r1 = \{E, R\}$ and $r2 = \{I, F, C\}$, in the TS of the CFPP stage control. Note that $r1$ is a pre-region for event AI and a post-region for PI whereas $r2$ is a pre-region for PI and a post-region for AI. Both regions are not crossed by AR and PR. Finally, G is an external event for $r1$ and internal for $r2$.

It is known from [9] that in order to generate a Petri net whose reachability graph is isomorphic to a given TS, the TS must be *elementary*. The *elementarity conditions*, additional to the above four basic conditions, are as follows:

A5. State separation property, which means that for any two different states there must exist a region which contains one of the states and does not include the other.

[3]We basically use a strong notion of equivalence, isomorphism between the given transition system and the transition system which is obtained from the reachability graph of the Petri net. Effectively, the synthesis technique of [7] supports a weaker form, bi-simulation.

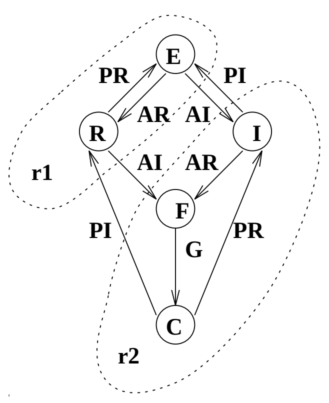

Figure 2: Illustration of regions

A6. Forward closure property, which states that, for every state s and every event e, if the set of pre-regions of e is included in the set of regions such that each of them contains s, then e must be enabled in s (i.e., there must be a transition from s labelled with e).

Following [9, 7], for any elementary TS there exists a safe, pure and simple PN such that: (1) each PN transition is labelled with an event of the TS; (2) no two transitions are labelled with the same event label; (3) the reachability graph of the PN is isomorphic to the TS.

The basic procedure, from [9], to produce a PN from an elementary TS is as follows:

1. For each event e a transition labelled with e is built in the PN;
2. For each region r a place named r is generated;
3. Place r contains a token in the initial marking iff the corresponding region r contains the initial state of the TS;
4. The flow relation is built according to the relationship between pre-regions and events, and events and post-regions.

A PN synthesised by this procedure is called a *saturated* net, since all regions are mapped into the corresponding places. A saturated net is canonical but has a lot of redundancy. As shown in [10], it is sufficient to consider only regions which are not sub-regions of other regions (such regions are called *minimal*). The net constructed from all minimal regions is also a canonical form and is called a *minimal saturated* net. Even the latter can be redundant to produce the Reachability Graph isomorphic to the TS. The method described in [7] performs additional optimisation and produces an irredundant net with minimal regions (the idea is somewhat similar to an irredundant cover of prime implicants in logic minimisation [11]).

It has also been shown in [7] that the elementarity conditions can be checked by means of a more practically efficient structural property of a TS, called "Excitation Closure". It is based on the notion of *excitation regions* for events.

A set of states is a *generalised excitation region* for event e, denoted by $GER(e)$, if it is a maximal set of states such that in every element of this set event e is enabled. Excitation Closure requires that for every event e the intersection of pre-regions of e is equal to $GER(e)$.

In our TS of Figure 2, the Excitation Closure property does not hold for several events. For example, $GER(PI) = \{I, C\}$ but the only pre-region of PI is region $r2 = \{I, F, C, \}$; $GER(G) = \{F\}$ but the set of pre-regions of G is empty. This TS is therefore not elementary.

The synthesis technique of [7] allows a number of modifications to the basic construction idea, including extensions to the class of elementary TSs.

It is possible to convert any TS to a PN by means of the so-called *label splitting*. Label splitting is the procedure that incrementally [4] distinguishes between the original labels (by enumerating their occurrences in different transitions of the TS) in such a way that the TS gradually becomes elementary with respect to the new (enumerated) alphabet. The convergence of this process is guaranteed by the fact that any TS with all transitions labelled uniquely can produce a state-machine net [12] whose structure and behaviour would be isomorphic to the original TS.

In order to synthesise non-pure PNs, the above Excitation Closure condition is generalised to allow the so-called *self-loop* pre-regions to be involved in the intersection of pre-regions for an event. A region r is a self-loop pre-region for event e if it is not a pre-region but the $GER(e)$ is contained in r. Including a place corresponding to such a region into the set of input and output places does not restruct the enabling conditions for an event unnecessarily. Yet it often allows to "trim" the intersection of pre-regions to such an extent that the given event is not enabled in the states not included in its excitation region.

Non-pure nets appear to be very useful in practice when modelling arbitration circuits and behaviour in which one event asymmetrically disables another event. The latter takes place, for example, in the models where an input signal disables an output signal (e.g., see the model of a Transparent Latch in [6]).

In a similar way, the Excitation Closure condition has been extended further, to allow *inhibitor* pre-regions. A region r is called an inhibitor pre-region for event e if its intersection with the $GER(e)$ is empty. This extension allows one to generate inhibitor nets. Such nets (e.g., for the case of safe nets) are known to be representable by ordinary PNs using the so-called *complementary* places. A place is called complementary to another place if it is marked in those and only those markings where the other place is unmarked. It is clear that a place and its complementary place correspond to a region and its complementary region (the latter is obtained by subtracting the former from the total set of states in the TS).

It is also noted in [7] that sometimes it is useful to include non-minimal regions to enforce Excitation Closure.

The class of TS, properly including elementary ones, which satisfies the Excitation Closure property extended in the above ways (self-loop and inhibitor pre-regions), is called *quasi-elementary* [7].

Finally, one can use the idea of including dummy events and corresponding transitions into the TS. Being "silent", such events do not change observational equivalence between the original TS and the modified one. Yet they allow to split some states of the original TS and construct new regions capable of satisfying the Excitation Closure condition.

To summarise, a number of techniques, some of which are computationally hard, can be applied in order to generate Petri nets from non-elementary transition systems. Below, we demonstrate the use of such techniques in practice, for the original TS model of the CFPP control, which is not elementary.

4 Deriving a Petri net for CFPP stage control

We are now ready to revisit our TS model of the CFPP stage control and transform it to such a TS that would would generate a PN using the above technique.

[4] Some useful heuristics are presented in [7] for label splitting.

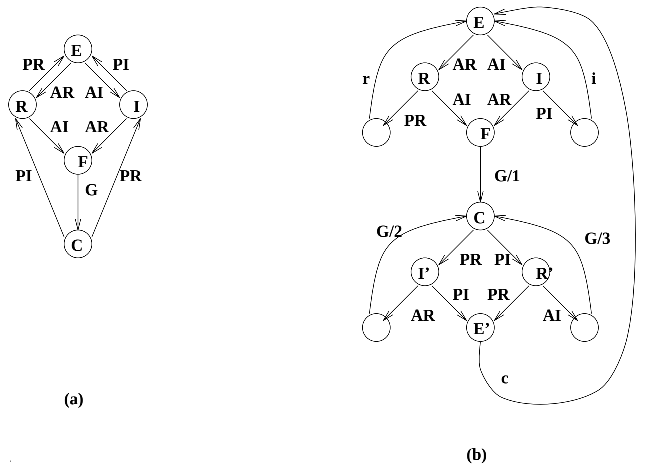

Figure 3: One way of separating the “diamonds”: (a) original TS, (b) transformed TS

It should be pointed out that the application of the label splitting method, albeit possible, does not satisfy our requirement about the *unique representation of all arbitrating events*. A PN produced by label splitting is isomorphic to the TS and has multiple transitions labelled with labels AI, PI, AR and PR.

The main obstacle in satisfying the Excitation Closure condition, even in its extended form (using self-loop pre-regions and/or inhibitor pre-regions) comes with the event G, for which we do not have appropriate pre-regions and post-regions. In order to help solving this problem, we slightly restructure the TS by introducing dummy transitions. Such dummies are added without changing the behavioural (bi-simulation and trace equivalence) semantics of the TS.

Intuitively, and this is one of the heuristics of the dummy insertion method, we need to establish proper “diamond” structures in the TS, reflecting the potential concurrency between pairs (AI, AR) and (PI, PR). For this, two ideas can be applied: “symmetric” and “asymmetric” approach.

4.1 “Symmetric” approach

This approach, eventually leading us to a circuit solution similar to the one found by C. Molnar [2], uses the idea of separating the two “state-transition diamonds”, one for the pair of events (AI, AR) and the other for the pair (PI, PR). Complete separation of these diamonds could be performed by unfolding the TS into two similar sub-graphs, as shown in Figure 3.

The new TS contains three dummy events, labelled i, r and c, and three transitions with split labelling of G. This TS satisfies the requirements of the Excitation Closure and can give us an appropriate PN.

It is however possible to make a more “economical” separation of the diamonds, with only one dummy event and event G left unsplit. This solution is shown in Figure 4, b, where states I and R are shared between the diamonds and the only dummy event is labelled with d. This dummy plays the same role for the (PI, PR) diamond as G for the (AI, AR).

The TS is not elementary in its basic form but is a quasi-elementary one since it satisfies the extended Excitation Closure condition (applied with self-loop and inhibitor regions). It gives the inhibitor net shown in Figure 4, c. The regions giving rise to the places of this net are as follows: $r1 = \{E, I, E'\}, r2 = \{E, R, E'\}, r3 = \{R, F, C\}, r4 = \{I, F, C\}, r5 = \{E, I, C\}, r6 = \{E, R, C\}$. The reader may check the pre-regions, self-loop pre-regions and inhibitor pre-region for all events

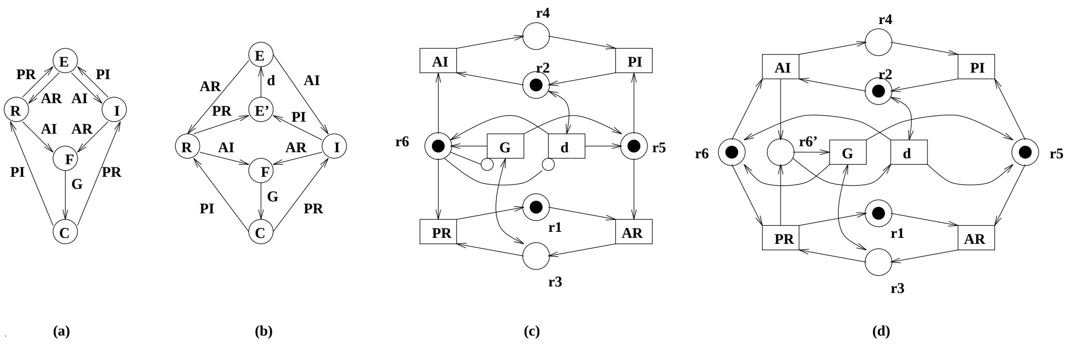

Figure 4: A better way to separate the "diamonds" and its synthesis result: (a) original TS, (b) transformed TS, (c) inhibitor net, (d) ordinary PN

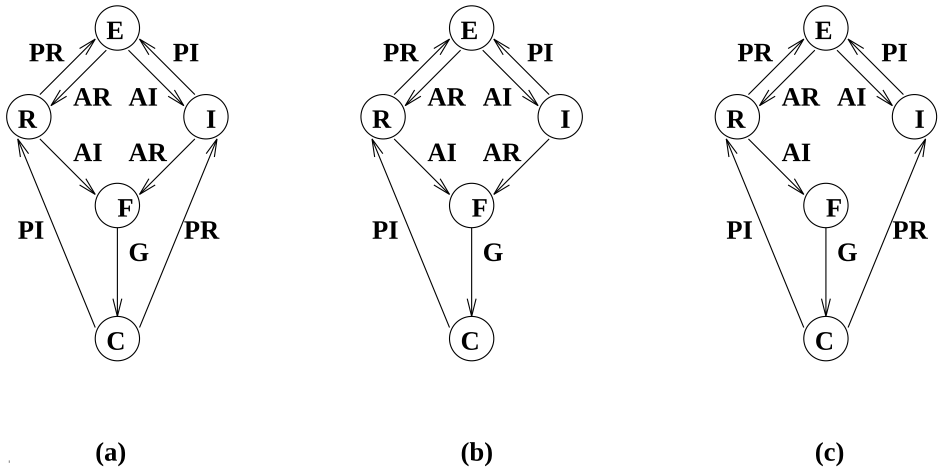

Figure 5: "Asymmetrisation" of TS: (a) original TS, (b) removing PR, (c) removing AR

by tracing them back from the net's arcs.

If we also consider an extra region $r6' = \{I, E', F\}$, which is complementary to $r6$, we can replace our inhibitor pre-region interconnections with ordinary pre-region relations, and thus obtain an ordinary PN shown in Figure 4, d. The reader may construct the reachability graph for this net and verify its isomorphic conformance to the TS in Figure 4, b. The latter is in its turn behaviourally equivalent to the original specification.

Later, we shall demonstrate how this net model is further used to produce a circuit implementation.

4.2 "Asymmetric" approach

Another way to transform the TS to its quasi-elementary form is slightly more liberal as far as the conformance to the original description is concerned. We may notice that it is possible to reduce the degree of concurrency in the original specification on one of the above-mentioned pairs of events. Namely, we can either "asymmetrise" the diamond formed by (AI, AR) or the one formed by (PI, PR), as shown in Figure 5.

Either of these transformations of the original TS restricts the trace semantics of the specification. We should therefore be able to demonstrate that the new TS is still consistent with the original requirements imposed on the CFPP interstage synchronisation. To do this, we have verified

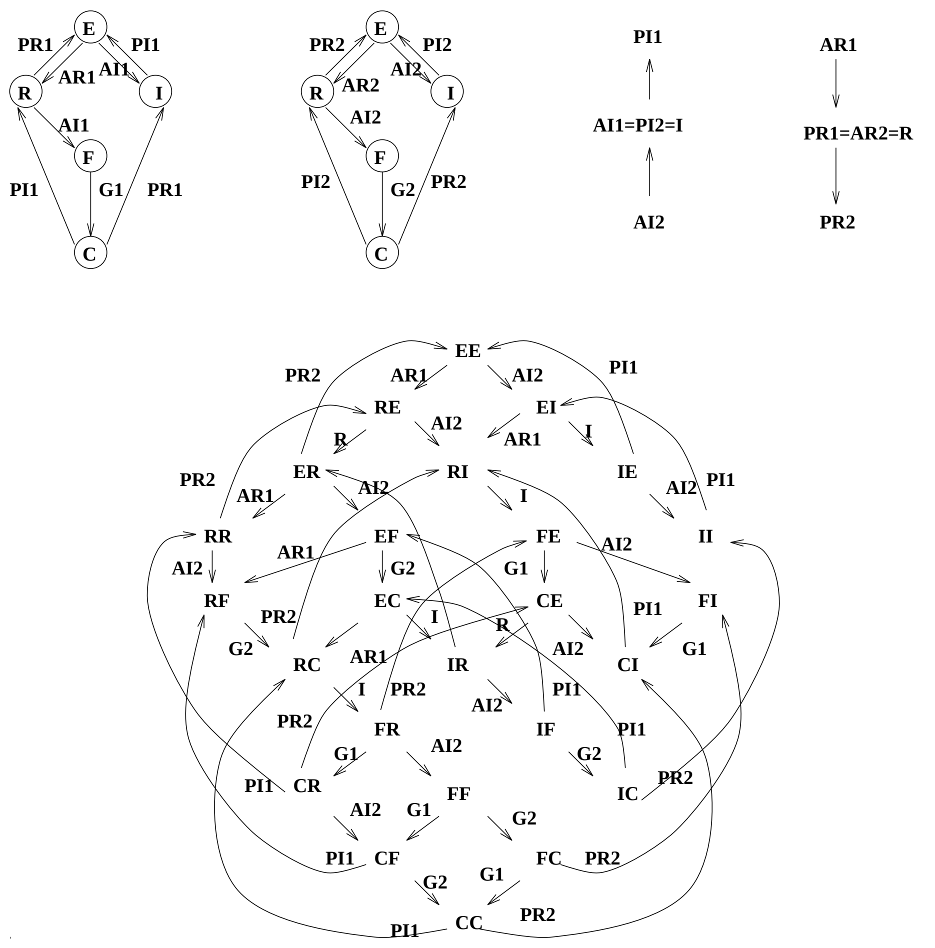
Figure 6: Composition of TSs of two adjacent stages

a composition of TSs of several stages and formally checked that the original requirements, outlined in Section 2, are satisfied.

The validity of such a check is intuitively clear from the following consideration. A simpler version of such a composition built for two adjacent stages, each modelled by the TS option shown in Figure 5,c (with AR deleted), is shown in Figure 6. This composition is a parallel composition of stage 1 and stage 2 with a "rendez-vous" type of synchronisation on the corresponding pairs of events, denoted as $AI1 = PI2 = I$ and $AR2 = PR1 = R$.

Although the composed TS may appear somewhat complicated, one can check the crucial synchronisation cases by examining groups of traces in it. For example, it clearly shows that the system is *deadlock-free*. The requirement of an instruction and result entering the pipe to *never miss* each other is seen from the following observation. Whenever both $AR1$ (result enters the pipe) and $AI2$ (instruction enters the pipe) have occurred, the system always passes through the states in which it performs garnering. It either happens in stage 1 (event $G1$) or in stage 2 (event $G2$).

Similar sort of analysis can be performed for the composition of TSs shown in Figure 5,b (with PR deleted).

Now, in order to produce PNs, both reduced TSs need a dummy event (ϵ) to be inserted in them, which brings no further semantical constraints. The corresponding quasi-elementary TSs are respectively shown in Figure 7, a and Figure 8, a.

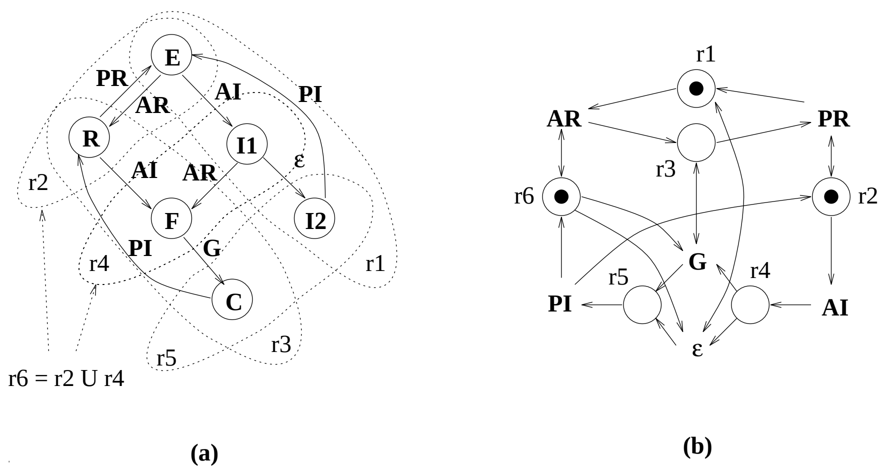

Figure 7: Petri net synthesis from asymmetric TS: (a) quasi-elementary version for TS shown in Figure 5, b; (b) Synthesised Petri net

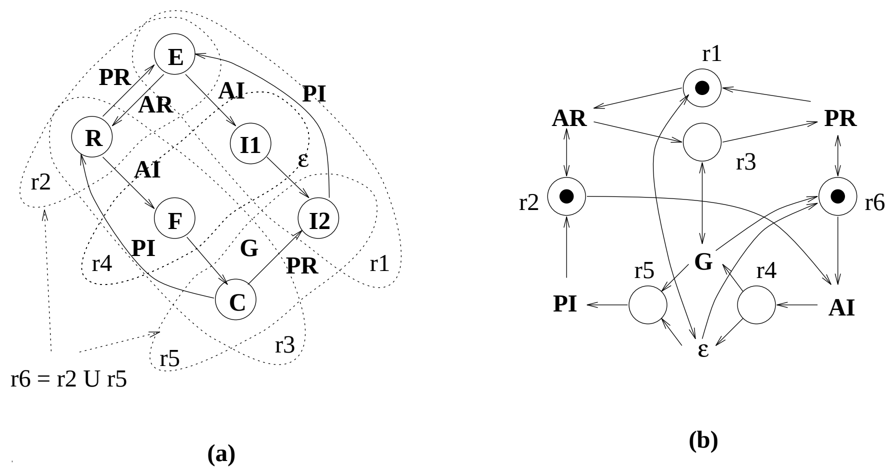

Figure 8: Petri net synthesis from asymmetric TS: (a) quasi-elementary version for TS shown in Figure 5, c; (b) Synthesised Petri net

The PNs derived from these two TSs are respectively shown in Figure 7, b and Figure 8, b. Note that in order to enforce the Excitation Closure in both these cases, we have to use non-minimal regions, denoted by $r6$, to become self-loop pre-regions for AR and PR, respectively. These self-loop pre-regions guarantee that AR and PR are not enabled in the states $I2$ and F in Figure 7, a and Figure 8, a, respectively.

We could, of course, derive inhibitor nets based only on minimal regions, using the fact that $r6$ is complementary to $r5$ in Figure 7, a, and $r6$ is complementary to $r4$ in Figure 8, a.

5 Circuit implementations for CFPP stage control

5.1 The overall approach

In this section, we first briefly review the general approach to circuit implementation of specifications with internal conflicts, originally presented in [6]. This approach would allow us to make an appropriate refinement of the PN models obtained in the previous section. The nets, as can be easily observed from their Reachability Graphs, exhibit conflicts in the form of disabling of some transitions by others.

More formally, a PN that reaches a marking m in which a pair of transitions $t1$ and $t2$ is enabled, and by firing one of them, say $t1$, a marking m' is reached such that $t2$ is not enabled, is called *non-persistent* with respect to $t2$.

For example, the net shown in Figure 8, b is non-persistent with respect to transitions AR and PR. Either of these can be disabled by firing AI.

Why is persistency crucial for circuit synthesis ?

As shown in [6], logic synthesis procedures, in particular those operating from the Signal Transition Graph (STG) refinements of Petri net models [8], can produce hazard-free implementations only for specifications without conflicts on non-input signals (so called *output-persistent* STGs). Obviously, the abstract events AR, PR, AI, PI in our specifications encapsulate transitions of both input and output signals – the former arrive in the stage while the latter are produced by the stage. Therefore, non-persistency with respect to, say, AR means that the logic synthesis techniques, such as those of [4], cannot be applied to deriving the logic for such output signals. The method described in [6] proposes to treat such signals separately, by “factoring them out” of the specification and associating them with standard arbitration components, such as a four-phase two-way *mutex* (ME) element or a two-phase *RGD-arbiter*, depending on whether a four-phase or two-phase circuit implementation is obtained [8].

The overall procedure [6] for implementing PNs and STGs with non-persistency with respect to non-input signals can be summarised as follows:

1. determine a set of non-input signals whose transitions make the PN non-persistent;
2. insert an appropriate set of *semaphore* actions, making semantic-preserving transformations at the PN level;
3. associate each semaphore with an appropriate ME element or an RGD-arbiter, depending on whether a four-phase or two-phase circuit is synthesised (if the semaphores are multi-way, use appropriate decompositions of multi-way mutex components to 2-way ME’s or 2-way RGD’s, respectively);
4. factor the semaphore implementations (the “mutex part”) from the circuit, making their outputs to be additional inputs to the circuit, which should now be output-persistent;

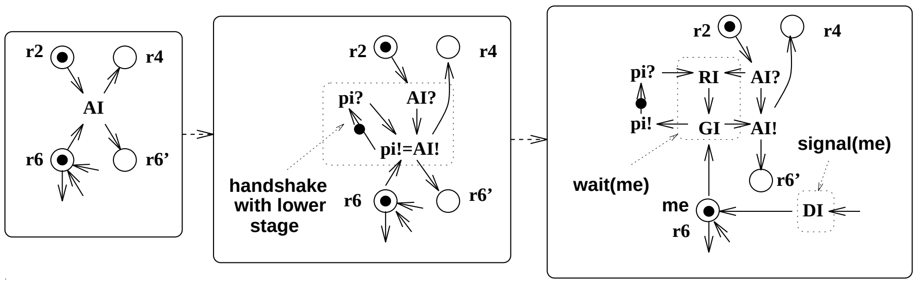

Figure 9: Action refinement

5. synthesise the "logical part" of the circuit by the existing (e.g., STG-based, if the specification has been refined to the event level of rising and falling edges of signals, alternatively use syntax-directed methods to derive logic from PNs);

6. combine the "mutex part" with the "logical part" by interconnecting the ME's or RGD's and the gate network through the set of request-acknowledgement handshakes.

5.2 Towards the circuit implementation of "symmetric solution"

We shall now illustrate how this technique is used to obtain a two-phase implementation for our "symmetric case", that is the PN shown in Figure 4, d.

First of all, we need to refine this net to reflect the idea of input and output signal actions in a stage. Figure 9 shows such a refinement for action *AI*; similar refinements can be obtained for the remaining three abstract actions *AR*, *PR* and *PI*.

Here, we first split *AI* into four events to represent two pairs of handshake signals. One pair is *AI*? and *AI*!, which is produced within the current stage. Here, *AI*? stands for a signal whose meaning is a query "I am ready to accept an instruction from the lower stage, can I accept it?"; the meaning of *AI*! is "You can accept an instruction from the lower stage". The other pair *pi*? and *pi*! models the handshake interface with the lower stage. Here, *pi*?, stands for a query "I am ready to pass an instruction, are you ready to receive it?" (which is an input signal to the current stage), and *pi*!, meaning "The instruction has been received (and latched [5] in a register) by me!" (which is an output signal from the current stage to the lower stage). It is obvious that events *AI*! and *pi*! can be produced simultaneously, hence, to avoid cluttering of multiple arcs, we use a single PN transition $pi! = AI!$ at this step.

Since the $pi! = AI!$ transition is now the cause of non-persistency (place $r6$ is a place through which the transition can be disabled by a corresponding transition in the results pipe model), we associate a semaphore denoted by me (to further become a two-way mutex element implemented by an RGD-arbiter) with place $r6$. The second step of refinement inserts explicit semaphore actions (wait(me) and signal(me)), protecting the $pi! = AI!$ transition. These actions are represented by the signal transitions of an RGD arbiter implementing this semaphore, namely *RI* (request from the instruction pipe), *GI* (grant to the instruction pipe), and *DI* (done from the instruction pipe).

Finally, when both actions *pi*! and *AI*! are protected by the me we can split them into two separate transitions assuming that they will further correspond in the circuit to two different wires forking out of a single grant output from the RGD-arbiter.

[5] We try to avoid describing the structure of the CFPP data path in this paper but in some cases it seems impossible to at least mention some actions on data path, such latching instructions and results within a CFPP stage.

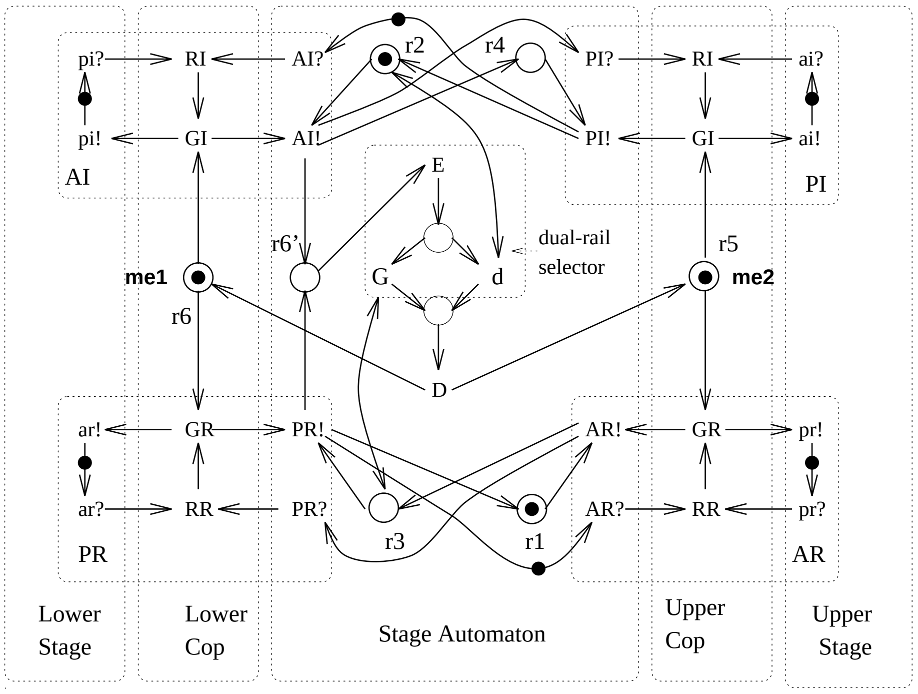

Figure 10: Petri net used for direct two-phase circuit implementation

We can now refine all four actions in a similar manner, that is followed by a trivial PN level transformation:

- adding two auxiliary transitions E (standing for "Execute") and D (meaning "Done");
- both "Done" events DI and DR, for both semaphores me1 and me2 can be combined into single "Done" (D) transition, to be implemented by one internal signal.

This transformation does not change the behaviour of the net with respect to its original semantics.

The resulting net is shown in Figure 10. It is easy to notice structural resemblance of this net to the organisation of the control, based on stage control circuits and inter-stage "cops", proposed in [2]. This is reflected in the dotted boxes.

We can now apply a direct transformation technique (similar to the one used in [8], which essentially adopted Patil's approach [13]) to obtain a two-phase circuit implementation:

- the mutex signal transitions are implemented by two RGD-arbiters; note that we may use a modified version of the RGD-arbiter, with a single "Done" signal [2] (sometimes called Sequencer);
- both pairs of transitions RI and RR can be implemented by C-elements;
- the "merging" place $r6'$ can be implemented by XOR;
- transition E produces an event-based signal to activate selection between G and d whereas its D counterpart is a simple fork after an XOR standing for a place which is input-incident to D;

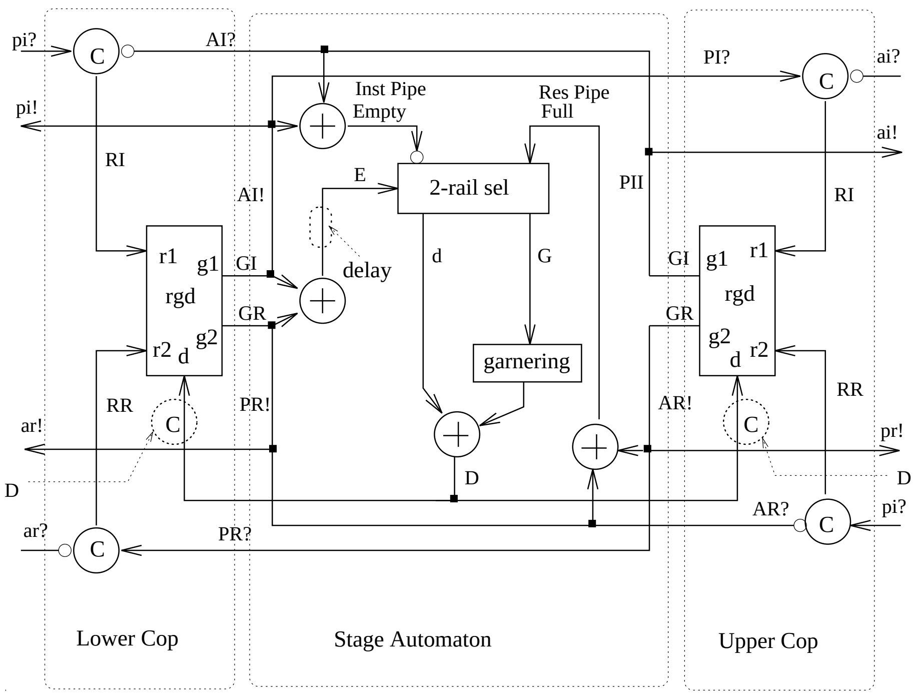

Figure 11: Two-phase circuit implementation of "symmetric" solution

- finally, G and d require a special circuit component, which is effectively a dual-rail Selector, with one event-based input E, two event-based outputs for G and d, and two level-based signals f and f, forming the boolean condition (dual-rail encoded) for this selector. The last two signals are logically "built" from the outputs from the RGD-arbiters. They have the meaning of marked places $r3$ (the results part is filled with an item) and $r2$ (the instruction part is empty).

The analysis of this net shows that the trickiest part of the circuit is the interface between the handshake signals of both pipes and the dual-rail selector. It can in fact be refined in a most straightforward way. Indeed, at the time when signal associated with transition E is produced, the marking of places $r1, r2, r3$ and $r4$ would either be $r1 = r2 = 0, r3 = r4 = 1$ or $r1 = r2 = 1, r3 = r4 = 0$. The former corresponds to the case of generating the "Garner" control signal, while the latter is the case of a "skip" signal. The skipping means that either instruction or result is passing through the stage without interaction with its counterpart. To implement these conditions in logic we can use for example two XOR's (one with inverted output) to produce level-based signals f and t, used to control the Selector. Each such XOR would stand for the boolean condition "the instruction (result) part is empty (filled with an item)".

The main circuit diagram is shown in Figure 11 while the internal structure of Selector is in Figure 12. Here, L is a Transparent Latch, whose generic logic equation is: $Q = DC + (D + \overline{C})Q$.

In this simple implementation, which is not purely speed-independent, we must guarantee, to avoid glitches in the Selector, that the delay with which signal E is applied to the Selector's input x is large enough compared to that of the XORs forming the dual-rail inputs f and t. If necessary, an extra delay element should be inserted in wire E to make sure that this signal does not arrive before the level-based control has reached its valid state.

This circuit, at such a modular level, looks very much like the one described in [2], except that the latter does not show the Full-Empty detector of the Stage Automaton while we "hide" the fact

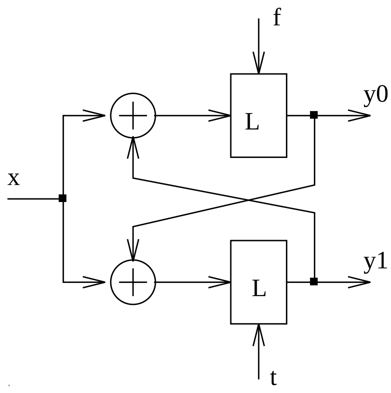

Figure 12: Circuit implementation for dual-rail Selector

the "Done" signal (input d) to each RGD-arbiter is in fact formed by joining two "Done's" of the two adjacent stages (as shown by dotted C-elements and extra connections in Figure 11). Plus, in this paper, this circuit has been obtained from the initial specification on a formal basis.

It would also be possible to make a different signalling expansion of the PN shown in Figure 4, d, using the four-phase approach. The circuit could then be built of ordinary four-phase ME elements and standard logic gates, whose boolean functions would be obtained from an automatic tool, such as SIS. We do not present this alternative here for the "symmetric" case but it will be shown for the "asymmetric" one.

5.3 Towards the circuit for "asymmetric solution"

We have two potential specifications to take up for further refinement and implementation, one is in Figure 7, b and the other in Figure 8, b. They basically produce the same effect on the results pipe, and differ mainly in the way the handshake in the instruction pipe is synchronised with garnering.

It is quite easy to notice however that executing each action AI or PI involves (see the AI refinement in the previous section) synchronisation of handshake signals of two adjacent stages, where the role of AI in the upper stage is similar to that of PI in the lower one. We can therefore conclude that both models would effectively yield the same performance – the critical cycle in the instruction pipe always involves an execution (garnering or skip) and two inter-stage transfers (note sequence $r2 \rightarrow AI \rightarrow r4 \rightarrow G|\epsilon \rightarrow r5 \rightarrow PI$).

What matters however is the way we synchronise two adjacent stages. Note that the mutually exclusive place $r2$ in the model of one stage can play the part of $p6$ in the adjacent stage. Hence one possible way of synchronising two stages could be to merge such places into one which will control the border between the two stages.

Such an approach is shown in Figure 13, where place labelled me2 is the merger of $r2$ and $r6$ for the composition of two stages of Figure 7, b. The meaning behind this model is as follows. As soon as an instruction enters the right hand side stage ("lower" stage in the terminology of [2]), having passed through semaphore place me1, it acquires the grant from the me2 semaphore, which controls the border between the stages. Only after both semaphores are held with the instruction pipe it can perform execution, that is either garnering or skipping the instruction, depending on the state of the results part. After completing the execution phase, the instruction pipe releases the me1 semaphore (token is returned back to place me2) and both instruction and result (or only instruction) can move further to their corresponding next stages.

Figure 14 shows a refinement of the above two-stage PN model, in which we separated actions

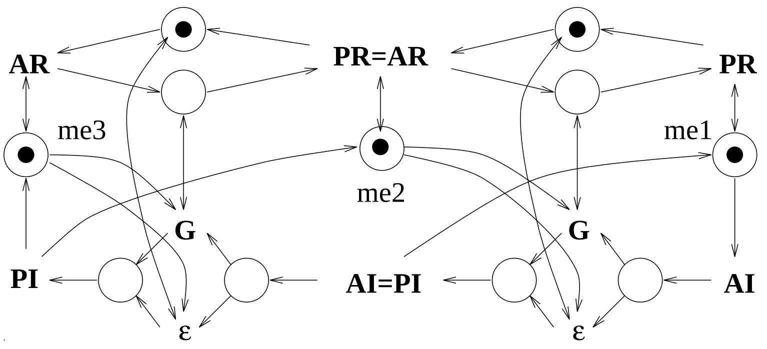

Figure 13: Model of two stages with merged "semaphore" places

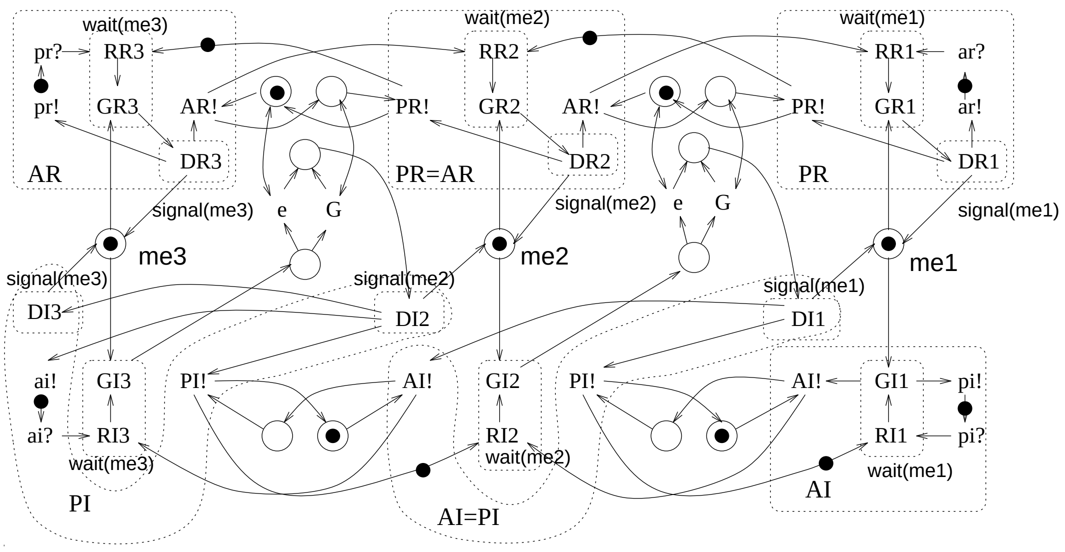

Figure 14: Refined model of two stages with merged "semaphore" places

on semaphores and handshake signal transitions to suit the two-phase signalling approach. This net can now be used for direct implementation by a two-phase circuit.

Figure 15 shows the corresponding circuit, which is "compiled" from the PN in the same way as in the "symmetric" solution. Note that due to asymmetry of the control organisation, the top part and the bottom part of the result and instruction pipes, respectively, are "non-standard". Similar sort of circuit solution, based on "asymmetrisation" of the CFPP control, has been recently presented by J. Ebergen [16].

It can be observed from the PN model that in this circuit two adjacent stages always perform execution (garnering or skipping) sequentially. That is, two instructions, in adjacent stages cannot be garnered in parallel. This constraint stems from the fact that the new execution in the lower stage can only begin if the upper stage has completed its execution and released the corresponding semaphore. This is an obvious disadvantage of our first approach to synchronise two adjacent stages by means of merging their mutual exclusion places $r2$ and $r6$ according to Figure 13. In such an organisation, the merger place, say me2, plays not only the role of mutual exclusion between the result and instruction pipes but also restricts concurrency between events in the instruction pipe.

It may appear possible to increase parallelism between the stages by allowing the upper stage to produce its acknowledgement to the lower stage before the execution in it begins, say after it

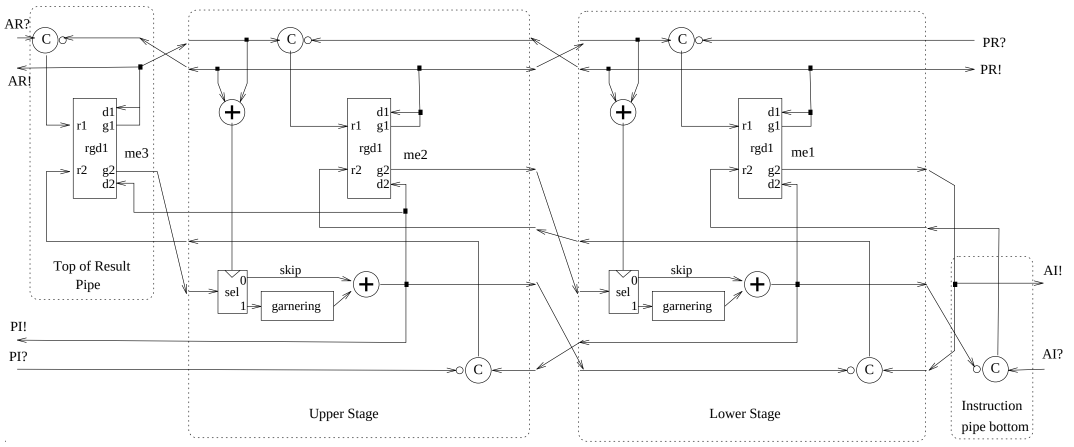

Figure 15: Two-phase circuit implementation of two adjacent stages

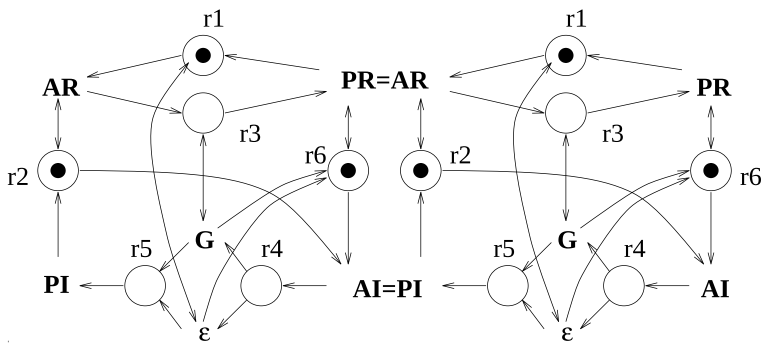

Figure 16: Model of two stages with unmerged “semaphore” places

has gained the grant from the next mutex element. But, in order to start the execution in the lower stage, we need to unlock the current mutex element, by also generating “Done” to it before the execution in the upper stage. This has its inevitable effect on unlocking the result pipe, which must not be allowed since the execution needs that pipe to be in stationary state.

We shall now try to avoid constraining concurrency by a different means, namely, by composing two stages preserving both mutex places $r2$ and $r6$ between them. The PN composition is shown in Figure 16. Here, as a building block we have taken the stage model of Figure 8, b.

It is easy to observe that this net allows execution to take place in both stages at the same time. Indeed the following simple sequence of events brings the net to a marking in which both places $r4$ have a token: $AI, \epsilon, AI = PI, AI$. Had the sequence $AR, PR = AR, AR$ preceded the previous sequence, we would have had both garnering actions G enabled in the above-mentioned marking.

This net is however slightly more difficult to implement. Note that events labelled $PR = AR$ and $AI = PI$ are atomic in the sense that they change the marking of both mutex places simultaneously (additionally, the instruction pipe's action $AI = PI$ also consumes a token from the next $r2$). First of all, we need to refine this net to such a form where the mutually exclusive places are associated with semaphore actions. Furthermore, we must be careful in such a refinement since, if we want to use standard 2-way mutex elements to implement semaphores, we need to split the above-indicated atomicity – such splitting often leads to a solution with a deadlock.

Luckily, we can observe in this net that place $r2$ is always decremented first in the instruction pipe. This pipe must therefore hold this token until it has seized a token from $r6$ and further

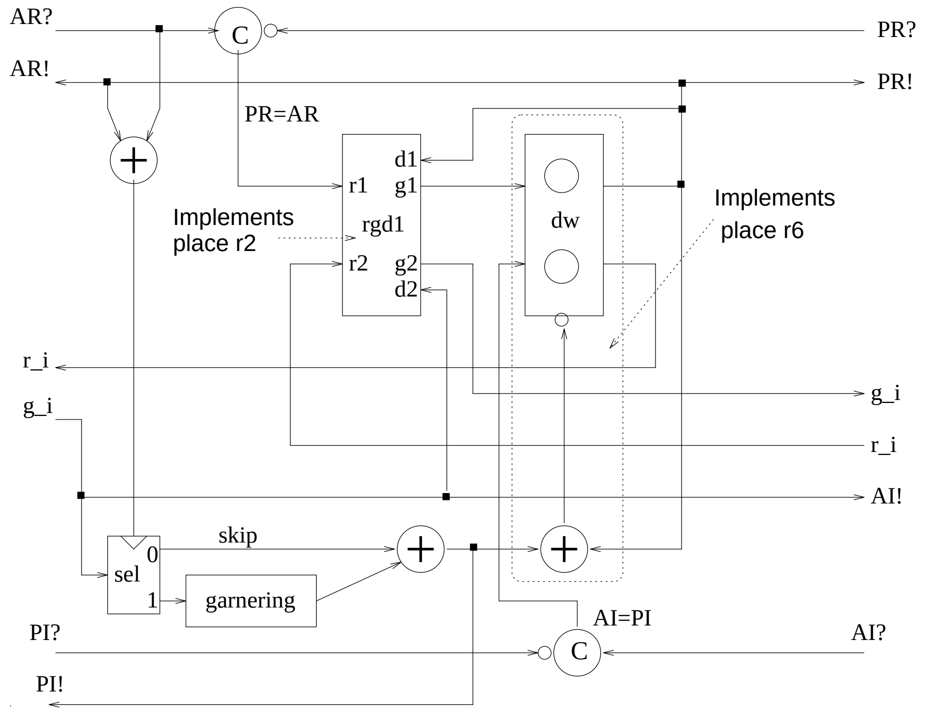

Figure 17: Two-phase circuit implementation for unmerged “semaphore” places

from the next $r2$. If we adopt the same order of acquisition, first $r2$ and then $r6$, for the result pipe (action $PR = AR$) we can guarantee that the net does not get into a deadlock. Indeed, the only place for which both pipes effectively arbitrate is $r2$. Place $r6$ thus plays the role of a passive synchroniser between the stages, so it need not be associated with a semaphore (and an arbiter in the circuit) at all.

For this stage of our report, we omit the phase of PN refinement as it seems now rather tedious to go through. The circuit (for a single stage inside the CFPP) obtained in a way similar to the previous solutions is shown in Figure 17.

Note that this circuit really benefits from the fact that the mutex place $r6$ is not an arbitration one. Instead of taking an extra RGD-arbiter, we can use a simpler block, a 2-by-1 Decision-Wait (originating from Join [15]), which is another standard two-phase circuit element (we also need an XOR to realise the “token-merging” functionality of place $r6$). Note also that, although signal $PI!$ is produced to the next stage only after the instruction execution, we generate both acknowledgement $AI!$ and a “Done” for the RGD-arbiter earlier, to enable the lower stage to process its following instruction.

Let us now look at an alternative way to implement our basic PN models. Instead of deriving a two-phase circuit directly from the refined PN description we shall try to make use of automatic tools, e.g. SIS, which can synthesise logic from an STG expansion of the PN [8]. It should be reminded that the ME elements must be factored out so that the synthesis tool will only need to produce boolean equations for “persistent” signals.

As an example, let us take the PN from Figure 8, b. In order to obtain an STG expansion of this net we need to represent explicitly all signal events in the form of falling and rising edges ($AI+, AI-, PI+, \ldots$). Additionally, we must insert semaphore actions, in the form of falling and rising edges on the request grant pairs to the appropriate ME elements. The result of such an

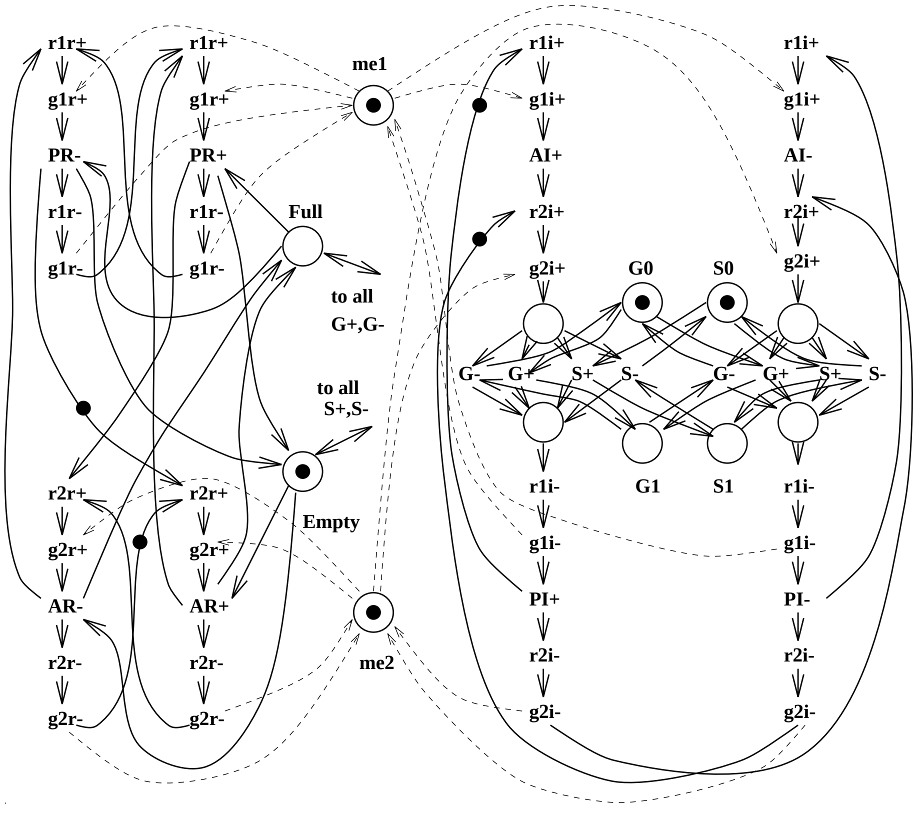

Figure 18: An STG expansion of PN shown in Figure 8, b

expansion is shown in Figure 18. This STG describes the behaviour of a single (internal) stage of the CFPP. Here, signal S represents the "skipping" case of the instruction execution; G still stands for the signal activating garnering.

Although it looks quite cluttered with arcs (some of which are only sketched for being quite obvious, while those connecting the mutex places are shown by dashed lines), this STG is still sufficiently clear to reflect its correspondence to the PN it originates from. To avoid further cluttering we do not split control events $AI+, \ldots$ into their actual handshake pairs $AI?+, AI!+, \ldots$, presuming that it would be rather clear to trace them in the circuit representation.

This STG involves 14 signals. The actual synthesised version was slightly bigger, 17 signals. It had, for example, a couple of internal signals that were included to assist unique state encoding. It was run through SIS on a fairly powerful workstation and produced results in a few minutes:

$$\begin{aligned}
r1i &= \overline{(G \oplus S \oplus PI)} \\
r2i &= (PI \oplus ai) \\
r1r &= PR \oplus d1r \\
r2r &= AR \oplus d2r \\
AR &= d2r\ g2r + AR(d2r + \overline{g2r}) \\
PR &= d1r\ g1r + PR(d1r + \overline{g1r}) \\
ai &= AI\ \overline{PI} + ai(AI + \overline{PI}) \\
PI &= ai\ \overline{g1i} + PI(g1i + ai)
\end{aligned}$$

$$\begin{aligned}
d1r &= AR\,\overline{g1r} + d1r(AR + g1r)\\
d2r &= \overline{PR}\,\overline{g2r} + d2r(\overline{PR} + g2r)\\
G &= g2i(AR \oplus PR)(ai \oplus S) + G((ai \oplus S) + \overline{(AR \oplus PR)} + \overline{g2i})\\
S &= g2i\,\overline{(AR \oplus PR)} + (ai \oplus G) + S((ai \oplus G) + (AR \oplus PR) + \overline{g2i})
\end{aligned}$$

Note that this STG, models only one stage and its logic implementation provides only a partial view of the entire circuit. It was believed that to have a complete logic, all synthesised by a tool, would give us a more trustworthy implementation rather than deriving only some parts of it through such reduced STG's with extra "logic glueing" done by hand. Indeed, in this asymmetric solution, one stage is, for example, responsible for setting a request to an ME element while the resetting is done by the adjacent stage. More complex STG's, describing three adjacent stages, were also attempted. Some of them were still within the limits of a signal count allowed by SIS (currently 32 variables are allowed for the SUN workstation version [17]) but the largest possible "approximation" of the circuit has been obtained for an STG with 25 signals. This STG has been for three stages but had other "reductions" compared to the one shown here.

The only difficulty in using the above equations for obtaining the final circuit is in the actual implementation of the signals setting the ME elements from the instruction pipe. In order to obtain this implementation we should bear in mind that the actual setting of the $r1i$, for example, is done by the previous stage of the instruction pipe. The current stage has control only over the resetting of the request signal $r1i$ in its ME element; it however also controls the setting of the request signal $r2i$ in the ME element of the following stage. Therefore, we can use the $r1i$ logic to form the resetting part of the ME request signal, while the setting part could be obtained from the $r2i$ logic. Such a "glueing" actions allows us to optimise the overall logic, exploiting also the separation between the request and acknowledge pairs of the handshakes AI and PI.

The circuit is shown in Figure 19. Some logical parts of it are represented as L (Transparent Latch) element, whose generic logic equation is: $Q = DC + (D + \overline{C})Q$, where D is a data input and C is a latching input. It should be pointed out that the decomposition of the complex gates for Full/Empty Selector into simpler elements can be easily traced from their equations. This decomposition however puts certain constraints on delays of these elements to avoid hazards.

6 Further work and conclusions

We have formally derived circuits for CFPP stage control from the initial state-based specification presented in [2]. This required combining two methodological approaches. One [7] is targetted at a Petri net model synthesised from the Transition System description. The other [6] synthesizing an asynchronous circuit from a Petri net description of its behaviour with internal conflicts.

Some interesting lessons have been learned from this exercise as to the scope and power of the state-of-the-art synthesis tools.

There are some stages in this derivation at which model transformations may involve participation of the designer. Firstly, it is the transformation of the Transition System to a quasi-elementary form, to allow the Petri net to represent labelled events by means of unique net transitions. The uniqueness requirement is crucial for those events that correspond to the transitions of output signals which are involved in behavioural conflicts. This is a requirement of the method in [6]. Another place of possible active involvement is the signalling expansion, that is the refinement of the Petri net into a Signal Transition Graph. A large number of alternative refinements can be

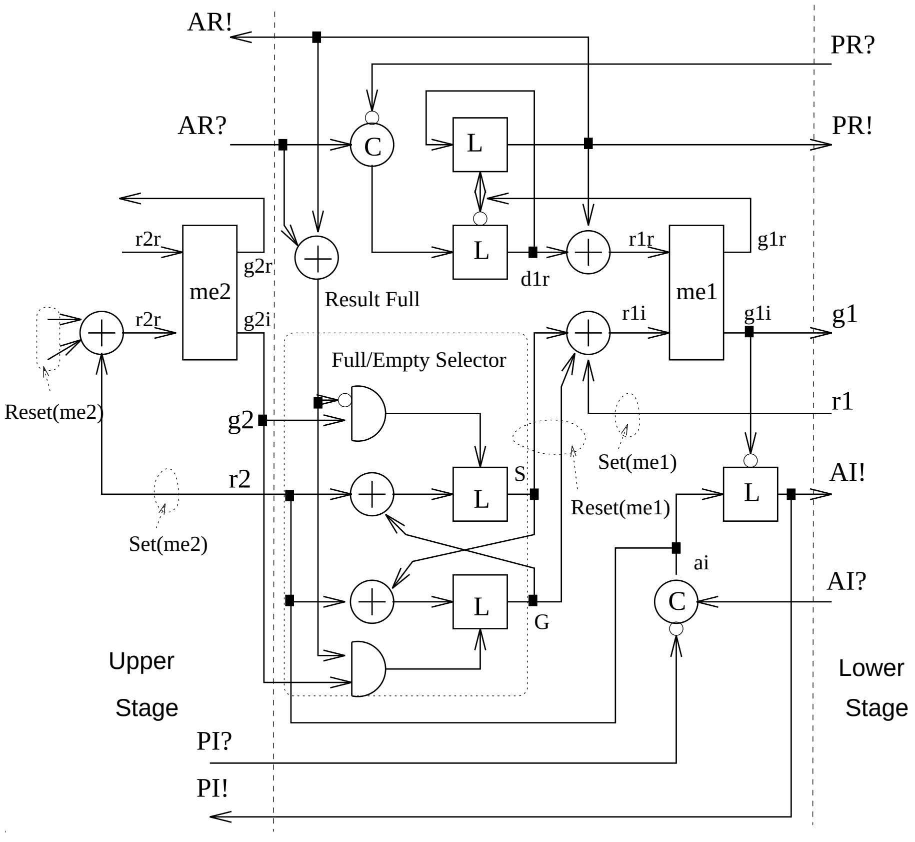

Figure 19: Circuit for PN shown in Figure 8, b, obtained through STG-based synthesis

built [8], all different in performance and size. The designer thus has to be able to make important decisions at the STG level. Due to the size constraints imposed by the synthesis tool, the designer may need to split the specification into parts and synthesise logic for them separately. Then, the final "glueing" is done at the logic implementation level. Here, an important role to be played is the verification at the circuit level as one can never guarantee that the dynamic behaviour of the composed circuit would be correct after such a glueing.

Note that verification is also needed at an earlier phase, to verify the legitimacy of transformations of the original specification, such as those reducing the set of possible execution sequences. We may also need to verify composition of Petri net models of individual modules. In this example, we had different alternatives to interconnect CFPP stages, to yield different performance and circuit size. The results of refining net models with abstract transitions into those with semaphore actions may also need checking for deadlock-freedom, especially if the structure of conflicts between net transitions involves several mutually shared places.

To summarise, we can state that both synthesis and verification steps are closely linked in this design process as some transformations are hardly mechanisable. Our asynchronous design tools should therefore provide an efficient interface between these steps, to allow the designer to interfere into this process at various stages.

In this report, we have not mentioned anything about actual evaluation of performance of the obtained circuits. Similarly, analysis of timing constraints for their hazard-free implementation (e.g., in simple logical gates) should not be missed out. Some recently proposed methods to solve these problems can be found for instance in [18, 19, 20].

Acknowledgements

The author would like to thank Jordi Cortadella, Mike Kishinevsky and Luciano Lavagno for fruitful collaboration on the topic of synthesising Petri nets from state-based descriptions. Many thanks to Luciano Lavagno for useful discussions about constructing a Petri net model of the CFPP example and his kind help with using SIS. I also highly appreciate help of Alex Semenov and his software tools in verifying the numerous Petri net models produced in the course of this work.

References

[1] S. Furber, P. Day, J.D.Garside, N.C. Paver, J.V. Woods. AMULET1: A micropipelines ARM. In *Proceedings of VLSI'93*, Grenoble, France, Sept. 1993, Best Paper Award.

[2] R.F. Sproull,I. Sutherland and C.E. Molnar. The Counterflow Pipeline Processor Architecture. *IEEE Design and Test of Computers*, Fall 1994, pp. 48 – 59.

[3] K. van Berkel, J. Kessels, M. Roncken, R. Saejis and F. Schalij The VLSI-programming language Tangram and its translation into handshake circuits In *Proc. EDAC'91*, pp. 384 – 389, 1991.

[4] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis University of California at Berkeley, UCB/ERL M92/41, May 1992.

[5] M. Kishinevsky, A. Kondratyev, A. Taubin, V. Varshavsky. Concurrent Hardware: The Theory and Practice of Self-Timed Design. John Wiley and Sons, London, 1993.

[6] J. Cortadella, L. Lavagno, P. Vanbekbergen and A.Yakovlev. Designing asynchronous circuits from behavioural specifications with internal conflicts. In *Proceedings of Int. Conf. on Adv. Res. in Asynch. Circ. and Syst.*, Salt Lake City, Utah, November 1994, pp. 106 – 115.

[7] J. Cortadella, M. Kishinevsky, L. Lavagno and A. Yakovlev. Synthesizing Petri Nets from State-Based Models Universitat Politecnica de Catalunya, RR 95/09 UPC/DAC, April, 1995.

[8] A. Yakovlev, A.M. Koelmans and L. Lavagno. High level modelling and design of asynchronous interface logic. *IEEE Design and Test of Computers*, Spring 1995, pp. 32 – 40.

[9] M. Nielsen, G. Rozenberg and P.S. Thiagarajan. Elementary transition systems *Theoretical Computer Science*, Vol. 96, pp. 3 - 33, 1992.

[10] L. Bernardinello, G. De Michelis, K. Petruni and S. Vigna. On Synchronic Structure of Transition Systems. Universita di Milano, 1994.

[11] R. Brayton et al. Logic Minimisation Algorithms for VLSI Synthesis. Kluwer Academic Publishers, Hingham, MA, 1984

[12] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, April, 1989, pp. 541 – 580.

[13] S.S. Patil and J.B. Dennis. The description and realization of digital systems. In *Proceedings of the IEEE COMPCOM*, 1972, pp. 223 – 226.

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

[15] R.M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-32(1), June 1974, pp. 21 – 33.

[16] J. Ebergen, Personal Communication, March 1995.

[17] L. Lavagno, Personal Communication, April 1995.

[18] H. Hulgaard and S.M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *Proceedings of Int. Conf. on Adv. Res. in Asynch. Circ. and Syst.*, Salt Lake City, Utah, November 1994, pp. 2 – 11.

[19] T.G. Rokicki. Representing and Modeling Digital Circuits. Ph.D. thesis, Stanford University, 1993.

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