Formal Methods in System Design, 12, 39–71 (1998)
© 1998 Kluwer Academic Publishers. Manufactured in The Netherlands.

Designing Control Logic for Counterflow Pipeline Processor Using Petri Nets

ALEXANDRE YAKOVLEV alex.yakovlev@ncl.ac.uk
Department of Computing Science, University of Newcastle upon Tyne, NE1 7RU, England

Received June 2, 1995; Revised January 28, 1997

Editor: E.M. Clarke

Abstract. This paper approaches the problem of synthesising an asynchronous control circuit for a stage of the Sproull Counterflow pipeline processor (CFPP) as an exercise in exploiting formal techniques available for Petri nets. We first synthesise a Petri net model of the CFPP stage control from its original "five-state-five-event" description due to Charles Molnar. Secondly, we implement that model 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 designing industrial strength circuits and developing design tools. Two recent processor design projects, the Amulet1 microprocessor [12] and Sproull's counterflow pipeline processor (CFPP) [30], 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 [1], supporting syntax-driven design from high-level programming specifications, and SIS [29] and FORCAGE [14], 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 as 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 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 [9]. 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 [30, 18], 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 [9] could be directly applied to it. The way from the specification to the circuit, as outlined in [30], 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 first draft version of our approach has been presented in [33]. Since the SCPP-A problem was posed in [18] there have been two other attempts presented in [15, 16]. Both of them are based on a process algebraic framework.

This paper tackles the problem from the standpoint provided by Petri nets and Signal Transition Graphs (STGs) [34]. The paper demonstrates the combined use of the following two major constituents:

1. Synthesis of a Petri net specification amenable to subsequent circuit implementation. This task includes two subtasks:
   - Behaviour-preserving transformations at the state-transition level, which are aimed at obtaining a state graph in such a form that can be converted into a Petri net.
   - Actual conversion of the state graph into a behaviourally equivalent Petri net [4, 5]; the net must satisfy the requirements of subsequent circuit synthesis [9].
2. Synthesis of a circuit in one of the two potential approaches:
   - The first one is a two-phase circuit consisting of special, micropipeline-like [31], elements. Such a circuit can often be obtained by a relatively straightforward conversion of the Petri net into an interconnection of macromodules, quite similar to a syntax-driven approach of TANGRAM.
   - The second approach, quite a challenge for today's synthesis tools, is to refine the net into a Signal Transition Graph (using the so-called "signalling expansion" [34]) 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 good motivation and provide guidance for further work on tools. For example, a new tool, called petrify, which is being developed by J. Cortadella on the ideas of [4, 5], supports synthesis of Petri nets from state graphs and

some behaviour-preserving transformations at the Petri net level. In its current status petrify also allows many options for speed-independent logic synthesis from STGs, including the so far most advanced method for resolving Complete State Coding problem [6, 7]. In fact, the recent version of petrify [8] has helped to obtain the Petri net model shown in Figure 10(c), which leads to the circuit shown in Figure 14. That net, synthesised originally by hand, had some redundant places and arcs. The four-phased logic implementation of the circuit has also been obtained by petrify.

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 provides the formal basis for the behaviour-preserving transformations. In particular, it 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. Sections 5 and 6 present implementations of the Petri net models of the CFPP stage control. Finally, Section 7 draws conclusions.

2. CFPP stage control circuit. Original description

For a complete description of the CFPP architecture we refer the reader to [30]. 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 stage 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 happening 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 state 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.

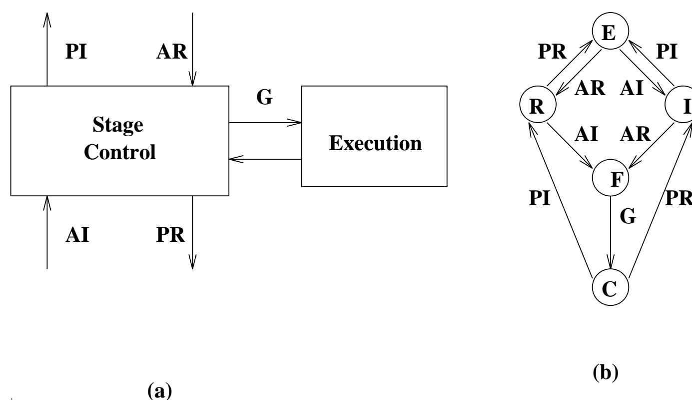

Figure 1. Counterflow pipeline stage control: (a) structural view, (b) state diagram

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 [30] 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 result may 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 instruction may arrive in the stage *or* instruction may arrive before result is allowed to leave the stage.

Before we proceed with the design, let us be slightly "suspicious" and check that the TS model indeed describes the behaviour which satisfies the original requirements imposed on the CFPP interstage synchronisation. To do this, we can build a composition of TSs of several stages and formally check that the original requirements, outlined in Section 2, are satisfied.

Such a check can be done for a simple version of the composition built for two adjacent stages, each modelled by the TS shown in Figure 1,b, is shown in Figure 2. This is a parallel FSM 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

Figure 2. Composition of TSs of two adjacent stages

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 $G2$). It is also clear that, the system discriminates neither the instruction nor the result pipes. For example, when the upper stage is full with instruction, the environment can still choose whether to remove this instruction (execute $PI1$) or push a datum into it (execute $AR1$). The latter possibility, although not obvious from the synchronisation requirement in Section 2 (which was in some sense a "weak" form of a progress property), is important to ensure that the system can "halt" an instruction in the upper stage of the pipe temporarily until it has been garnered with some data.

The presentation in [30] “jumped” from the 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 [9].

3. Formal background for CFPP control circuit design

In this section we introduce some theoretical background of the model transformations essential in performing our design task.

3.1. Basic outline of formal transformations

The brief outline of the basic idea for our model transformations, together with a simple example, is depicted in Figure 3. Here, the initial model requires two actions [2] a and b to proceed in parallel but only once, i.e. for a (or b) to occur again it must wait for the completion of b (a). The *circuit semantics* of the model, used in subsequent refinement, also assumes that actions a and b are started by the designed control circuit. The latter means that these actions can be refined into the so-called *active* handshakes [1]. In such a handshake the first transition (e.g., a rising edge) is produced on an output *request* signal, and it is acknowledged by the environment of the circuit with a transition on the *acknowledgement* wire.

The two transformations shown in part (a) are aimed at a Petri net description which is amenable to subsequent circuit implementation, which can be done by using either of the two approaches shown in part (b) of the same figure.

Transformation (1.1) is applied to a Transition System [3] which does not satisfy a *semi-elementarity* condition, defined later in this section. The latter is a necessary and sufficient condition for applying further transformation (1.2). To satisfy that condition we insert at stage (1.1) additional events into the model, and those auxiliary events can be regarded as *dummy* (sometimes also called “silent”[17]) actions. The correctness of this transformation will be taken in the sense of its *observational equivalence* (also known as *weak bisimilarity*) [17] between the original and resultant Transition Systems, which is sufficiently powerful for the purposes of asynchronous design. It is also formally defined below. In our example, an auxiliary event d allows to satisfy the semi-elementarity conditions. The new Transition System is observationally equivalent to the original one with respect to the set of events $\{a, b\}$.

Transformation (1.2) is based on the notion of regions in a Transition System, which are sets of states corresponding to places in the synthesised 1-safe Petri net (see Figure 3.(a)). If the Transition System satisfies the condition of semi-elementarity, the net synthesised from it generates the reachability graph which is isomorphic to the Transition System. Thus, due to the property of transformation (1.1), the Petri net should be observationally equivalent to the original description. Note that the event labels of transitions in the original Transition System are used as the (unique) labels of the net’s events. The 1-safe net model shown in Figure 3.(a) is observationally equivalent to the original description.

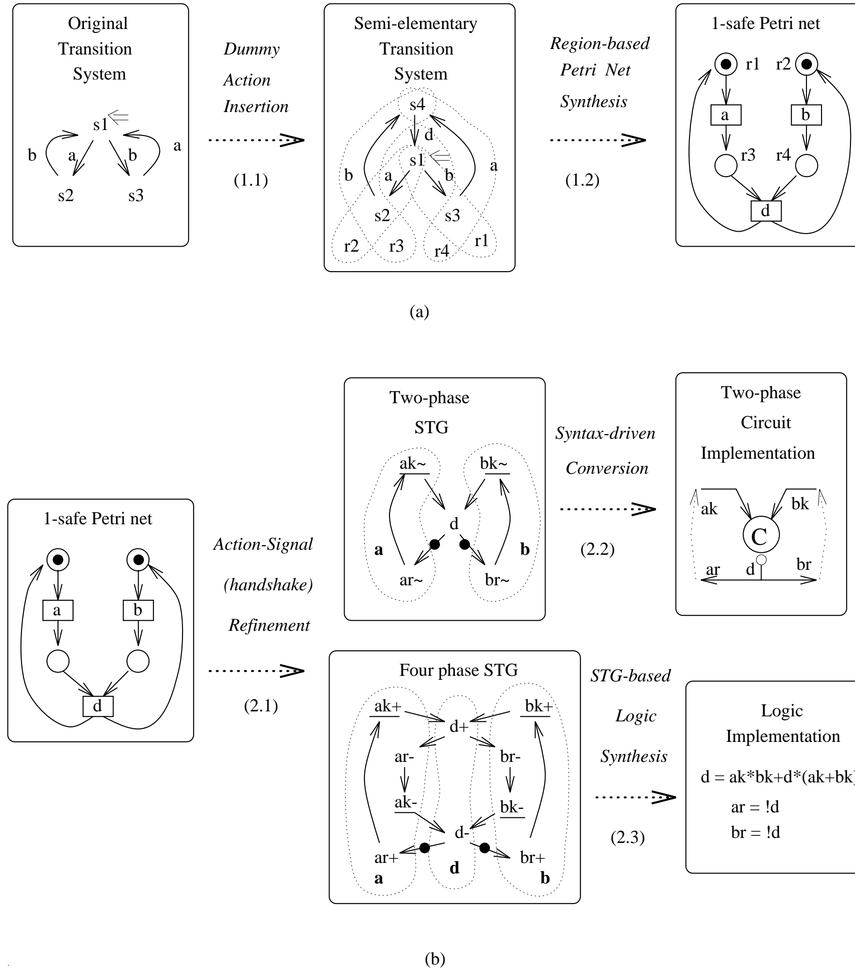

Figure 3. Outline of model transformations

Transformation (2.1) is basically an action refinement. It is however different from (1.1) since it involves associating an original event name with a set of events. Furthermore, it is performed at the Petri net level. In order to cast it into the notion of observational equivalence, we need to establish a mapping between the set of refined actions and the original actions. For every original action such a mapping should select a *critical* event from the refined set while other events must be regarded as silent actions. The idea of such refinements for labelled Petri nets has been defined in [35, 34]. The refinement can be done in two ways leading further to circuit implementations (2.2) and (2.3). Note that for the example shown in Figure 3 those implementations produce the same result, which is of course not true in general; an alternative (2.3) implementation is shown in Figure 4.

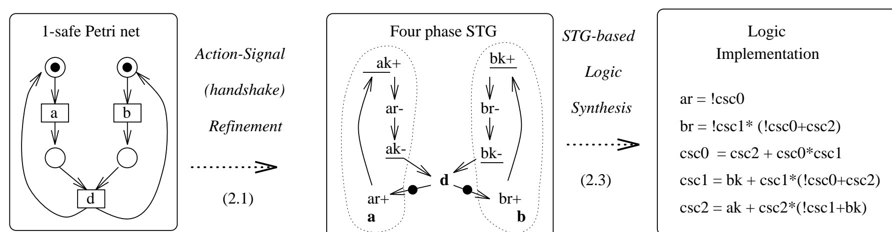

Figure 4. Alternative four-phase STG refinement

The (2.2) label is assigned to a two-phase circuit implementation, where the circuit is obtained by direct, syntax-based, conversion of Petri net fragments into corresponding macromodules in the style of [31] or [23]. The class of 1-safe *simple* [19] Petri nets is sufficient to perform such conversion [23]. The net, called a two-phase STG in Figure 3.(b), is obtained from the original net in the (2.1) transformation stage by means of: (i) expanding abstract events into pairs of handshake signals (*handshake expansion*) in a *two-phase protocol* (also known as a Non-Return-to-Zero, NRZ, protocol[4]) [31], and (ii) for resolving conflicts with *output signal non-persistency*, by inserting *semaphore actions* which are implemented with arbitration elements [9]. In our example, the circuit semantics of events a and b in the original model is such that they correspond to two *active* handshakes. Therefore, they are refined into two pairs of signal transitions $(ar\sim, ak\sim)$ (respectively, $(br\sim, bk\sim)$), standing for a request to execute action a (b) and an acknowledgement of its completion. The fact that the request part is leading in those handshakes (since they are both active) is reflected in the relative position of tokens – before $ar\sim$ and $br\sim$.

The (2.3) stage is concerned with synthesis of a logic gate implementation, which is called a four-phase implementation because it is synthesised from an STG in which signals are refined according to a *four-phase protocol* (also known as a Return-to-Zero, RZ, protocol[5]). Similar to (2.2), the (2.3) implementation also requires from the (2.1) refinement that abstract events are expanded into handshakes and explicit arbitration actions [9] are inserted. Unlike (2.2), the actual derivation of logic is however performed by means of logic synthesis from the STG, where one can (or rather must, due to complexity reasons) use automatic tools like SIS or petrify [29, 8], which access the logic minimisation package Espresso [3]. In our example, we refined both handshakes into an STG for its four-phase logic synthesis in the way that is not much different from the two-phase signalling. This is to benefit from the existence of the auxiliary event d, which can itself be interpreted as an extra state signal (not a handshake pair) and refined into a pair of transitions $d+$ and $d-$. The latter are used to help solving the *Complete State Coding* problem, which is a necessary condition for obtaining logic equations for the output signals. Alternatively, refining only a and b handshakes, we could completely rely on the synthesis tool (in this case, on petrify), which could solve both the state coding and logic synthesis issues. This is illustrated in Figure 4, where three additional state signals ($csc0, csc1$ and $csc2$) have been added for Complete State Coding.

On the whole, the stages (1.2) and (2.3) are those which are supported by the existing tools. Stage (2.2) seems to be easy to automate either, however, there are still no good

tools for compilation of control circuits from Petri nets (perhaps, the reason for that is exactly that this task is not seen today as theoretically challenging!) The stages (1.1) and (2.1) are fairly hard to automate. Like any other refinement tasks, either in hardware or software design, they are subject to intuition of the designer, and require use of verification tools – e.g., to check behavioural equivalence (bisimilarity), deadlock freedom, safety and consistency. One can however apply some correctness-preserving Petri net refinements based on correspondence between structural and behavioural properties of nets [19].

In the following subsections we shall define our formal objects, Transition Systems and Petri nets, shall more formally address the two major notions of behavioural equivalence that will be used in our transformations. One is bisimulation of Transition Systems – it underlies the (1.1) and (2.1) transformations. The other is the isomorphism-based relationship between semi-elementary Transition Systems and 1-safe Petri nets, which supports stage (1.2).

3.2. *Transition Systems and their behavioural equivalence*

Transition systems. A *transition system* (TS) is a quadruple $TS = (S, E, T, s_{in})$, where S is a non-empty set of states, E is a set (alphabet) of events, $T \subseteq S \times E \times S$ is the transition relation, and s_{in} is the initial state.

A TS is represented by a directed graph in which every arc connecting a pair of states is labelled with a name of an event. Such a labelled arc is called *transition*. One state is marked as the initial state. We assume that any TS satisfies the following *basic conditions* [21, 4, 5]:

A1. For every $(s, e, s') \in T, s \neq s'$, i.e., no transition may begin and end in the same state.

A2. For every $e \in E$ there are s, s' such that $(s, e, s') \in T$, i.e. every event must have some occurrence.

A3. For every $s \in S - \{s_{in}\}$ there are $(s_i, e_i, s_{i+1}) \in T$, for $i = 0, 1, \ldots, n$, such that $s_0 = s_{in}$ and $s_{n+1} = s$, i.e. every state is reachable from the initial state.

Two TS's $TS = (S_1, E_1, T_1, s_{in1})$ and $TS = (S_2, E_2, T_2, s_{in2})$ are *isomorphic* iff there exist a pair (h_S, h_E) of total bijective mappings: $h_S : S_1 \to S_2, h_E : E_1 \to E_2$ such that $(s, e, s') \in T_1 \Longleftrightarrow (h_S(s), h_E(e), h_S(s')) \in T_2$.

We will say that s' is *reachable* from s by a (possibly empty) sequence $\sigma \in E^*$ of events $e_i, i = 0, 1, \ldots, n$ if there is a (possibly empty) sequence of transitions $(s_i, e_i, s_{i+1}) \in T, s_0 = s$ and $s_{n+1} = s'$. This is denoted by $s \xrightarrow{\sigma} s'$. The sequence σ is then called *feasible* in state s. A special case of such a sequence is a *feasible event* in state s, i.e. $s \xrightarrow{e} s'$ iff $(s, e, s') \in T$.

The *projection* of a sequence of events $\sigma \in E^*$ feasible in $s \in S$ on an event alphabet E' is an event sequence $\sigma' \in (E \cap E')^*$ obtained from σ by deleting all symbols which are not in E'. Let E' be an alphabet of *observable* events. We then say that a sequence of events σ' is *observably feasible* in $s \in S$ iff σ' is the projection of a sequence $\sigma \in E^*$ on E' and σ is feasible in s. This is denoted by $s \xrightarrow{\hat{\sigma'}} s'$. Again, a special case is $s \xrightarrow{\hat{e}} s'$, which means that there exists a sequence of events $\sigma \in E^*$ feasible in s and exactly one of these events is labeled with e.

Behavioural equivalence. Consider two TS's $TS_1 = (S_1, E_1, T_1, s_{in1})$ and $TS_2 = (S_2, E_2, T_2, s_{in2})$. A *weak bisimulation* with respect to an event alphabet E is a binary relation $\approx_E \subseteq S_1 \times S_2$ such that $(s_1, s_2) \in \approx_E$ implies, for all $e \in E$, that:

- (i) $s_1 \stackrel{\hat{e}}{\longrightarrow} s_1'$ implies that there exist $s_2, s_2' \in S_2$, such that $s_2 \stackrel{\hat{e}}{\longrightarrow} s_2'$ and $(s_1', s_2') \in \approx_E$, and
- (ii) $s_2 \stackrel{\hat{e}}{\longrightarrow} s_2'$ implies that there exist $s_1, s_1' \in S_1$, such that $s_1 \stackrel{\hat{e}}{\longrightarrow} s_1'$ and $(s_1', s_2') \in \approx_E$.

The above TS's TS_1 and TS_2 are called *observationally equivalent* (or *weakly bisimilar*) with respect to their initial states and an event alphabet E iff there is a weak bisimulation $\approx_E$ and $(s_{in1}, s_{in2}) \in \approx_E$.

This definition of bisimulation is quite similar to the original one from [17], and will be identical to it if we consider sets $E1 \setminus E$ and $E2 \setminus E$ to consist of a single event, called *silent action*. Such a restriction would be quite appropriate for justification of transformation (1.1). However, for the (2.1) transformation, the above definition is slightly more convenient.

A simple example of a TS transformation which preserves observational equivalence with respect to the original set of actions $\{a, b\}$ is shown in Figure 3.(a). The initial states of the TS's are both labelled with $s1$.

3.3. Labelled Petri Nets and their behavioural equivalence

Labelled Petri nets. The target of specification synthesis at stages (1.1) and (1.2) is a labelled Petri net. We assume that the reader is familiar with the basic terminology of Petri nets [19]. Thus we give here only a brief outline of the most relevant issues.

A *Petri net* (PN) is quadruple $N = (P, E, F, m_0)$, where P is a finite set of *places*, E is finite set of *transitions* (or events), $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*, and $m_0 : P \to \{0, 1, \ldots\}$ is the *initial marking*. A PN is usually represented as a directed graph consisting of two types of vertices, circles for places and bars (or boxes) for transitions, and arcs, leading from circles to bars and from bars to circles, to show the flow relation [6]. The initial marking is usually depicted in the graph by means of tokens (black dots) put into places according to their number prescribed by function m_0.

An event is *enabled* in a marking m if all its input places are marked under m. An enabled event 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 (RG) of the net. The RG of a PN $N = (P, E, F, m_0)$ is a TS $RG(N) = (S_N, E, T_N, m_0)$, in which the set of states S_N is formed by all markings reachable from m_0, the set of events coincides with the set of events of N, the set of transitions T_N is formed by the transitions between markings (m, e, m') whenever e can fire under $m \in S_N$, and the initial state is identified with the initial marking.

A labelled PN is a PN in which every event $e \in E$ is labelled with a symbol, called *label*, from a given alphabet A, thus giving rise to a *labelling function* $\lambda : E \to A$. Hence a labelled PN is a triplet $LN = (N, A, \lambda)$. In the case of *unique labelling*, i.e., if λ is

bijective, each event in the net can be uniquely identified by its label. In such a case we can use the label as the event's name. In this paper, we shall be mostly working with the uniquely labelled events (with the exception of perhaps dummies unless we need to distinguish them). In addition to the TS $RG(N)$, the reachability graph of the underlying PN N, a labelled PN LN produces another TS $RG(LN) = (S_N, A, T_N, m_0)$, which is graphically the same TS as $RG(N)$ but its transitions are labelled with names from alphabet A. We shall call such a reachability graph the labelled reachability graph of a labelled PN LN. It is obvious that if λ is a *total unique labelling*, then $RG(N)$ and $RG(LN)$ are isomorphic to each other.

A PN is called *1-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 *(strongly) live* with respect to an event e if from any reachable marking m_1 it is possible to reach a marking m_2 under which e is enabled. A PN is called *live* if it is live with respect to all events. A PN is called *persistent* with respect to an event e if for any reachable marking m_1 under which this event is enabled we cannot reach another marking m_2 by firing another event e' and e is not enabled under m_2. A PN is called *persistent* if it is persistent with respect to all events. The property of 1-safeness is crucial for TS to PN and PN to two-phase circuit conversions. The property of liveness is not particularly critical for conversions but it helps to keep track of the effectiveness of all events and signals in the circuit, i.e., that they are not redundant in the modelled operational modes. Finally, persistency, especially persistency with respect to events modelling output signals of the circuit, is important because non-persistent events must be implemented by special arbitration elements (which contain analogue devices) to avoid hazards.

A special case of a labelled Petri net is a *Signal Transition Graph* (STG). An STG is a triplet $G = (N, Y, \lambda)$, where $N = (P, E, F, m_0)$ is a PN, Y is a nonempty set of binary signals, and $\lambda : E \rightarrow Y \times \{+, -, \sim\}$, where $y+(y-)$ stands for the rising (respectively, falling) edge of signal y (e.g., in the four-phase signalling), while $y\sim$ means either rising or falling edge of y (e.g., in the two-phase signalling). In other words, $y\sim$ means a transition of signal y regardless of the current state. Thus an STG is a PN whose events are labelled with the names of binary signal transitions.

Two PNs N_1 and N_2 are called *observationally equivalent* with respect to a set of events E iff their RGs $RG(N_1)$ and $RG(N_2)$ are observationally equivalent with respect to E and their initial markings. Similarly, two STGs G_1 and G_2 are *observationally equivalent* if their labelled RGs $RG(G_1)$ and $RG(G_2)$ are observationally equivalent with respect to a signal set Y and their initial markings.

For example, let us refer back to stage (2.1) in Figure 3. Consider first the two-phase refinement. Actions a and b have been refined into two pairs of handshake signal transitions $(ar\sim, ak\sim)$ (respectively, $(br\sim, bk\sim)$). Now, if we semantically identify $ar\sim$ with a and $br\sim$ with b, regarding all other events as auxiliary ones (dummies), we can easily detect that the refined STG is observationally equivalent to its PN origin, and hence equivalent to the original TS. The legitimacy of that semantic identification is however determined by our characterisation of the criticality of transitions $ar\sim$ and $br\sim$ in representing their high-level counterparts a and b. The decision about such characterisation is obviously made by the designer, who refines the model, and thus cannot be completely formalised.

Similarly, we can approach the question of correctness of the four-phase refinement, where we should make appropriate semantic identification between four-phase signal transitions and the original abstract events. Thus, if we identify $ar+$ with a, $br+$ with b and $d+$ with d, we can easily observe that the refined STG is equivalent to the original model.

In carrying out transformations for the CFPP model at the PN and STG levels, we shall be relying on the behavioural equivalences defined in this section. Wherever it is non-trivial we shall be adding some comments on our refinements justifying them from both the designer's (semantic identification of events at different levels of abstraction) and formal (explaining why a particular PN structural transformation satisfies observational equivalence) viewpoints.

3.4. From Transition Systems to Petri Nets

The basic intuitive idea behind the construction of a Petri net whose behaviour is equivalent [7] 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*, *co-region*) of an event e if r is exited by (entered by, internal for) e.

Figure 5 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, post-region for AI and a co-region for G. 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 [21] that in order to generate an elementary net whose reachability graph is isomorphic to a given TS, the TS must be *elementary*. Elementary nets are effectively a subclass of 1-safe Petri nets [8]. The "gap" between 1-safe nets and elementary nets is filled by *non-pure* nets, which allow a self-loop relation between places and transitions. Such nets and their TS's, called *semi-elementary*, have been studied in [25]. In this design exercise, we need to be able to deal with non-pure nets.

The *semi-elementarity conditions*, additional to the above three basic TS conditions, are as follows:

A4. State separation property, which requires that for any two different states there must exist a region which contains one of the states and does not include the other.

A5. Forward closure property, which requires that, for every state s and every event e, if the sets of pre-regions and co-regions of e are included in the set of regions such that

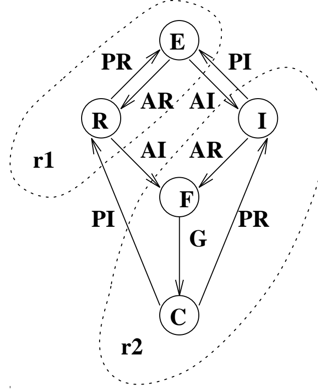

Figure 5. Illustration of regions

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 [25], for any semi-elementary TS TS there exists a 1-safe PN N such that: (1) each event in N is *uniquely* labelled with an event of TS; (2) the $RG(N)$ is isomorphic to TS.

The basic procedure to produce a PN from a semi-elementary TS is as follows:

1. For each event e an event 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-/co-regions and events, and between events and post-/co-regions.

A PN synthesised by this procedure is called a *saturated* net, since all regions are mapped into the corresponding places. A saturated net may have a lot of redundancy, i.e. some of its places may be removed without disturbing the isomorphism of reachability graphs. As shown in [2], 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 called a *minimal saturated* net. Even the latter can be redundant to produce the Reachability Graph isomorphic to the TS. The method described in [4] and implemented in the petrify tool 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 [3]). The semi-elementarity condition may sometimes require to use non-minimal regions as co-regions. Since enumeration of non-minimal regions is computationally hard, petrify applies some heuristics to optimise its search for co-region

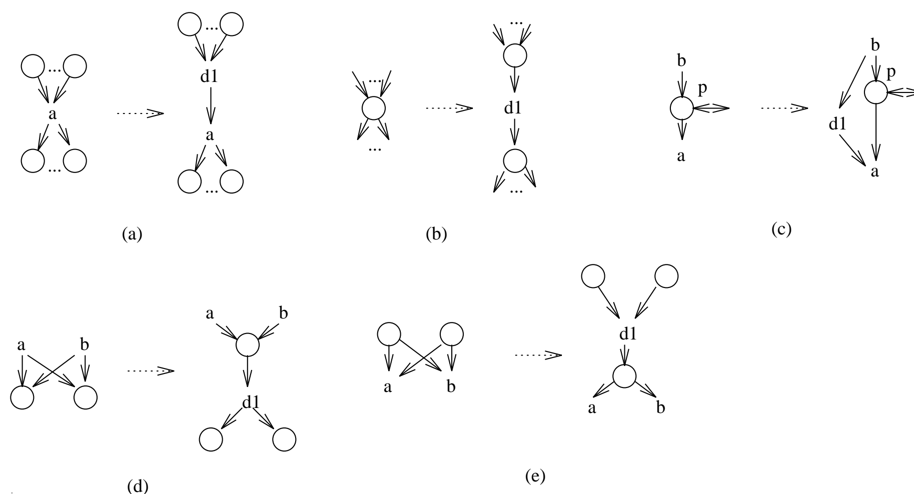

Figure 6. Petri net refinements preserving observational equivalence

candidates. Furthermore, note that petrify uses a slightly modified version of the semi-elementarity condition, called "Excitation Closure", based on excitation regions [4], which simplifies its checking in a symbolic state representation framework.

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 and co-regions of e is equal to $GER(e)$.

In our TS of Figure 5, the semi-elementarity 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 semi-elementary.

3.5. *Summary of Petri net refinements relevant for CFPP design*

Before we proceed with applying actual model transformations on the CFPP specification, let us consider some refinements that can be applied at the PN (or STG) levels, which preserve behavioural correctness of the original model in terms of observational equivalence. Refer to Figures 6,7 and 8. When talking about signal transition refinements, we will only consider the case of a two-phase signalling protocol for brevity.

Firstly, let us consider refinements which are insensitive to the semantics of the transformation. These refinements, shown in 6, can be applied to a 1-safe Petri net and are effectively structural transformations of the net which do not change its behavioural properties, such as safeness and liveness. They preserve observational equivalence with respect to the set of original event names. We omit here any formal proofs of this fact as the reader can easily find such justifications in the existing literature on PNs (e.g., see [19]). Note that the insertion of an auxiliary action $d1$ between a pair of original actions a and b shown

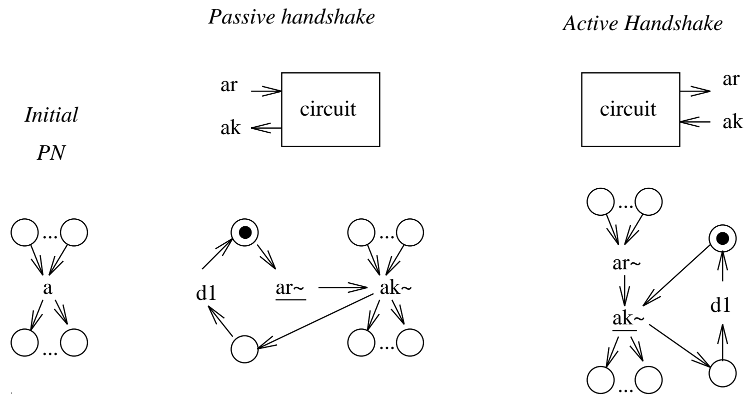

Figure 7. Passive and active handshake refinement of an abstract action

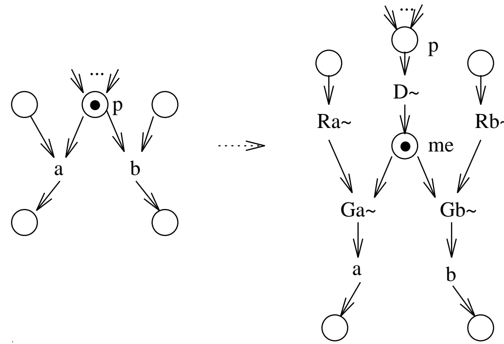

Figure 8. Refinement with explicit arbiter signals

in Figure 6(c) can sometimes be done without splitting place p beween a and b. This can be important when we need to preserve the fact that some condition associated with such place p must remain **true** between the firings of a and b.

Secondly, two major types of refinements with "semantic flavour" (they are driven by the process of circuit design) are handshake refinements and semaphore (or mutual exclusion) action insertion. The two main handshake refinements are shown in Figure 7. These are a "passive" and "active" handshakes [1]. An abstract action a on a port with a request-acknowledgement handshake (ar, ak) is typically refined with two events $ar\sim$ and $ak\sim$. For a passive (active) handshake the request $ar\sim$ is an input (output) event, and the acknowledgement $ak\sim$ is an output (input) event. We assume that whenever the environment (circuit) is willing to execute action a on a passive (active) handshake the circuit (environment) must be ready. This is reflected in the synchronisations shown in the figure. Note that, if we semantically identify the critical event $ak\sim$ with original action a (indeed, the acknowledgement $ak\sim$ actually determines that action a has been completed), then it should be clear that for a 1-safe Petri net, the net obtained after such refinements is

observationally equivalent due to the rules of the PN transformations shown in Figure 6. The role of dummy events $d1$ in Figure 7 is auxiliary; from the semantic point of view, they "close" the handshakes through the environment of the circuit, thereby preserving the model's liveness. Such a "condensed" model of the environment can be especially helpful when we need to connect two circuits together at the level of their Petri net models; here one party must be made active while the other is passive. A relevant example can be found further in Section 5.

An example of mutual exclusion action insertion is shown in Figure 8. This refinement, with a strong semantic motivation of circuit design, allows for the use of a particular type of arbiter, the so-called RGD arbiter (with request-grant-done interface) [31] with a single "Done" signal (it is sometimes called Sequencer) [30]. Again, this refinement is well backed up by the PN transformations of Figure 6, and preserves observational equivalence with respect to actions a and b (and, of course, the rest of the surrounding net's actions).

We could present similar semantical refinements for the four-phase signalling convention but we leave them out to avoid over-complication. Thus, at this point we are well-equipped with formal techniques for model transformation and we can proceed with our design.

4. Synthesis of a Petri net model for CFPP stage control

Our first step is to revisit the original TS model of the CFPP stage control and transform it to such a TS that would generate a PN using the above technique. As has been pointed out earlier, when discussing Figure 5, the original TS is not semi-elementary. The main obstacle in satisfying the semi-elementarity (Excitation Closure) condition comes with the event G, for which we do not have appropriate pre-regions and post-regions. We need to insert auxiliary (dummy) events into the original TS in such a way that the resulting TS is semi-elementary and bisilimar to the original TS. This would correspond to stage (1.1) in our classification of Section 3.1.

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). Our approach, eventually leading us to a circuit solution similar to the one found by C. Molnar [30], 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 9.

The new TS contains three dummy events, labelled i, r and c, and three transitions with split labelling of G. Splitting the labels into several *different* instances of a *semantically unique event name*, does not violate observational equivalence, and can always be applied in its labelled PN version. 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 10, 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 [21] (without use of co-regions) but it is a semi-elementary one since it satisfies the Excitation Closure condition for pre-regions and

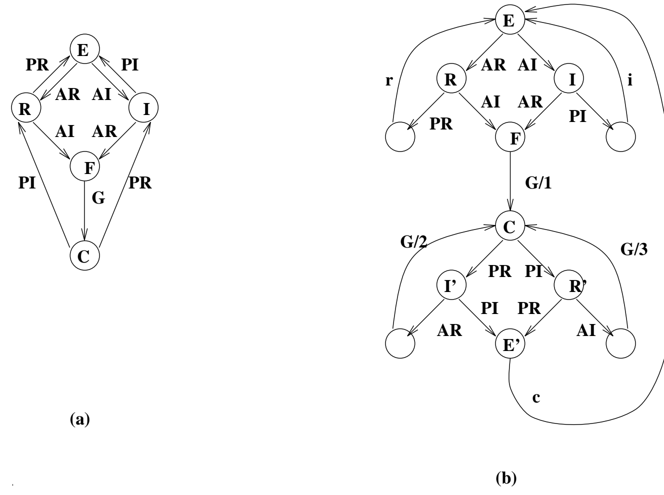

Figure 9. One way of separating the "diamonds": (a) original TS, (b) transformed TS

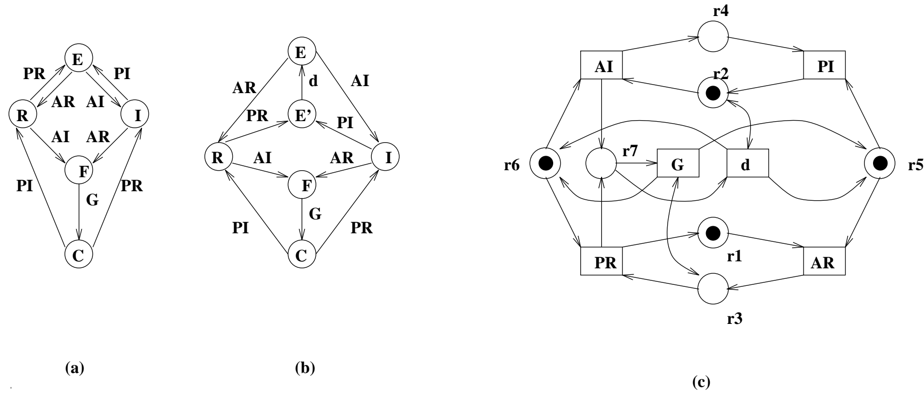

Figure 10. A better way to separate the "diamonds" and its synthesis result: (a) original TS, (b) transformed TS, (c) ordinary PN

co-regions. We can therefore proceed to stage (1.2) in our transformation process. For this we apply the procedure from Section 3.4. This procedure produces the net shown in Figure 10 (c). The regions giving rise to the places of this net are as follows: $r1 = \{E, I, E'\}, r2 = \{E, R, E'\}, r3 = \{R, F, C\}, r4 = \{I, F, C\}, r5 = \{E, I, C\}, r6 = \{E, R, C\}$ and $r7 = \{I, E', F\}$. Note that (r_1, r_3), (r_2, r_4) and $(r6, r7)$ are pairs of complementary regions (places) (for any such pair, the TS is always either in the first or second region). The reader may check the pre-regions, pre-regions and co-regions for all events by tracing them back from the net's arcs. Due to the presence of co-regions to events, and hence self-loop arcs, the resulting net is non-pure.

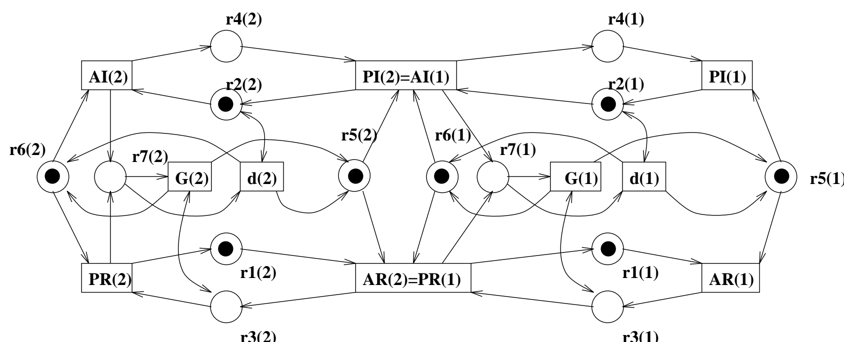

Figure 11. Connecting Petri net models of adjacent stages

The reader may also wish to construct the reachability graph for this net and verify its isomorphic conformance to the TS in Figure 10 (b). The latter is in its turn observationally equivalent to the original specification. This equivalence also guarantees the correctness of a construction of the entire control of the CFPP out of the PN models. For example, the parallel composition of two one stage models, shown in Figure 11, produces exactly the same behaviour as the composition at the TS level we had earlier in Figure 2. Note that unlike the TS composition, which suffered from state explosion, the size of the aggregate PN grows only linearly. It is important to note that the semantic identification of actions $PI(i+1) = AI(i)$ and $AR(i+1) = PR(i)$ of neighbouring stages plays a crucial role in our subsequent refinement of these actions at the handshake signal level.

We shall now use this net model to produce a circuit implementation.

5. Circuit implementation for CFPP stage control: a two-phase solution

In this section we concentrate on the realisation of stages (2.1) and (2.2) of our transformation strategy (Section 3.1), which will lead us to a two-phase circuit implementation of a CFPP stage.

1. Our first step is the handshake refinement of actions AI, PI, AR and PR, on which neighbouring stages of the CFPP synchronise (see Figure 11). Using the refinements shown in Figure 7 we transform the model of Figure 10(c) into the PN shown in Figure 12. Note that due to the direction of the flow of data path (instructions and results), the AI and AR actions are refined as passive handshakes, whereas PI and PR as active ones. With this choice we assume that it is always the environment who produces the first event on the instruction (AI) and result (AR) interfaces. A dual assignment of the passive and active roles would also be possible (but it seems less intuitive sematically). The crucial point here is that the (linking) handshake pairs (PI, AI) and (PR, AR) of the adjacent stages must be those of mutually complementary handshake types.

   When applying handshake refinement, we make an important designer's decision that the new (acknowledging) events $AIk\sim$, $PIk\sim$, $ARk\sim$ and $PRk\sim$ stand semantically

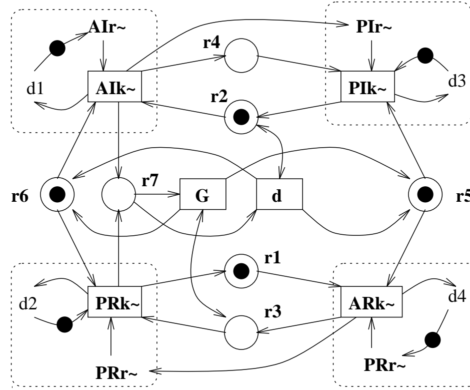

Figure 12. Handshake refinement for CFPP

for their respective abstract prototypes AI, PI, AR and PR. Indeed, from the semantics of the original model, e.g., action "Accept Instruction" is effectively accomplished when the $AIk\sim$ is generated. Since the handshake for AI is initiated by the environment (request to propagate the instruction is designated by event $AIr\sim$), the input condition $r6$ of event AI is inherited by $AIk\sim$. Similar reasoning is applied to the refinement of AR. For the active handshake of PI (action "Pass Instruction"), the request (event $PIr\sim$) is initiated by this stage, therefore this event is inserted in the way $d1$ is added in Figure 6(c). Action PR is also refined as an active handshake. The use of events $d1$, $d2$, $d3$ and $d4$ is purely auxiliary, they help to preserve liveness of the net model; from the semantic point of view, they "close" the handshakes through the environment.

Note that we would like to preserve the complementarity of pairs of places $(r2, r4)$ and $(r1, r3)$ because these places will be used further as indicators of the conditions that the instruction pipe is full ($r4$ is marked) or empty ($r2$ is marked) and that the result pipe is full ($r3$ is marked) or empty ($r1$ is marked). We would like these conditions to be toggled by events $AIk\sim$, $PIk\sim$, $ARk\sim$ and $PRk\sim$, likewise in the original net, where they are toggled by AI, PI, AR and PR. Thus, for the above-mentioned identification of event names, the net of Figure 12 is equivalent to that of Figure 10(b).

2. Our second step is the introduction of new events to resolve conflicts (non-persistency) between pairs of events ($AIk\sim$, $PRk\sim$) and ($PIk\sim$, $ARk\sim$) with respect to shared places $r6$ and $r5$. The need and a basic technique for such transformation at the PN level has been justified in [9]. Intuitively, it should be clear that if two events associated with output signals of the synthesised circuit are in behavioural conflict in the net model (there exists a marking under which both events are enabled but the firing of each of these events can disable the other event), we cannot build a hazard-free circuit out of purely logic based modules. In such cases, we must use special circuit elements called *arbiters* [9]. Such arbiters exist in both two-phase (*RGD-arbiter* with two 'Done" signals [31, 34] or one "Done" signal [30]) and four-phase (*Mutex element* [9]) design technologies. The type of the arbiter to be used in the implementation dictates the

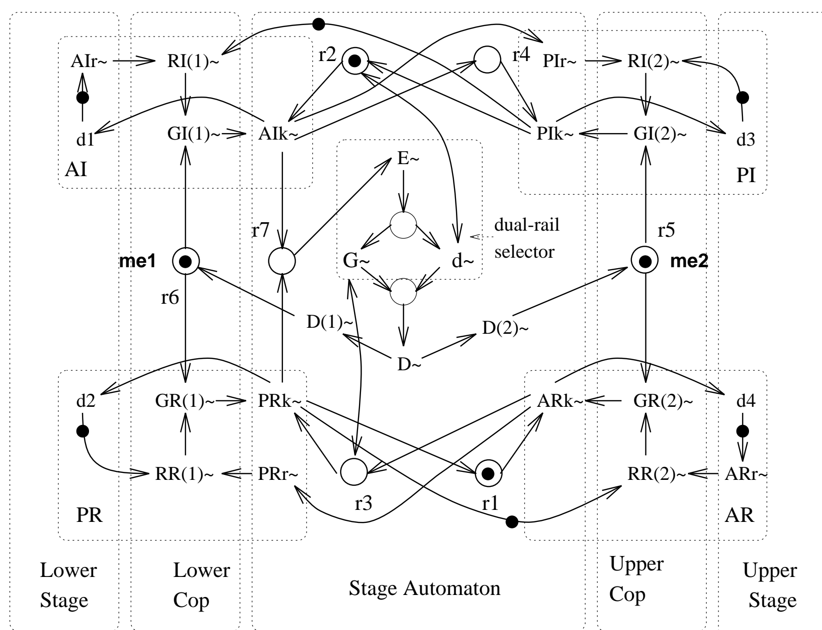

Figure 13. Full refinement of PN model for CFPP

action refinement. For this example, we are going to use an RGD-arbiter single "Done" signal, hence our refinement of the net in Figure 12 will proceed in accordance with the technique of Figure 8. The result of this refinement is shown in Figure 13.

In this refinement, when inserting request and grant events in front of $AIk\sim$, $PIk\sim$, $ARk\sim$ and $PRk\sim$, we applied the technique of Figure 6(c) to preserve the complementarity of pairs $(r2, r4)$ and $(r1, r3)$.

Additionally, in this net we have also used some of the PN refinements of Figure 6 in order to help ourselves in subsequent conversion of the net to a two-phase circuit (note for example the fragment of Figure 13 labelled "dual-rail selector"):

- (i) we added event $E\sim$ by means of splitting place $r7$ according to Figure 6(b);
- (ii) we applied Figure 6(d) to obtain a single output place for mutually exclusive events $G\sim$ and $d\sim$, and thus obtained an event $D\sim$;
- (iii) event $D\sim$ was then trivially forked into two separate "Done" events, $D(1)\sim$ for place $p6$ (also labelled as the mutex $me1$ condition) and $D(2)\sim$ for place $p5$ (also labelled as $me2$), in accordance with Figure 8.

The resulting net in Figure 13 bears much structural resemblance with the organisation of control based on stage control circuits and inter-stage "cops", proposed in [30]. This is reflected in the dotted boxes.

3. We can now apply a direct transformation technique (similar to the one used in [34], which essentially adopted Patil's approach [23]) to obtain a two-phase circuit implementation:

   - the mutex signal transitions are implemented by two RGD-arbiters with single "Done" [30] (sometimes called Sequencers [24]);
   - events $RI(1) \sim, RI(2) \sim, RR(1) \sim$ and $RR(2) \sim$ can be implemented by C-elements;
   - the "OR-joining" place $r7$ can be implemented by XOR;
   - transition $E\sim$ produces an event-based signal to activate selection between $G\sim$ and $d\sim$ whereas its $D\sim$ counterpart is a simple fork after an XOR standing for a place which is input-incident to $D\sim$;
   - $G\sim$ and $d\sim$ require a special circuit component, which is effectively a dual-rail Selector, with one event-based input $E\sim$, two event-based outputs for $G\sim$ and $d\sim$, and two level-based signals f and t, forming the boolean condition (dual-rail encoded) for this selector. The last two signals are logically built of 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);
   - finally, dummy events $d1$, $d2$, $d3$ and $d4$, which were introduced to "close" all four handshakes through the environment, are mapped into inverters (since the arcs outgoing from those events carry tokens). These inverters again are purely auxiliary – they model the environment as it is seen with respect to those four handshakes.

   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 the 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 14 while the internal structure of Selector is in Figure 15. 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.

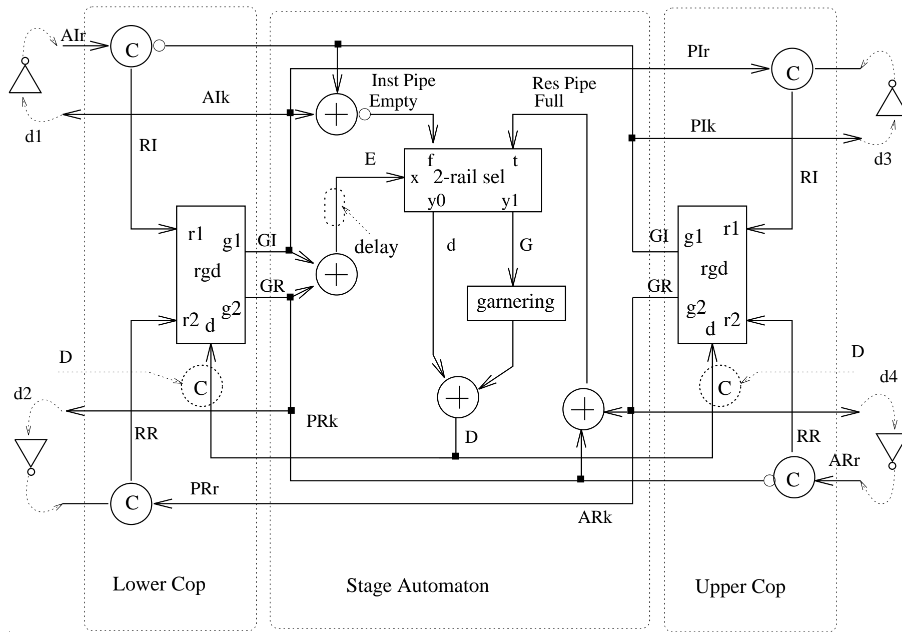

Figure 14. Two-phase circuit implementation of CFPP stage (with two neighbouring “cops”)

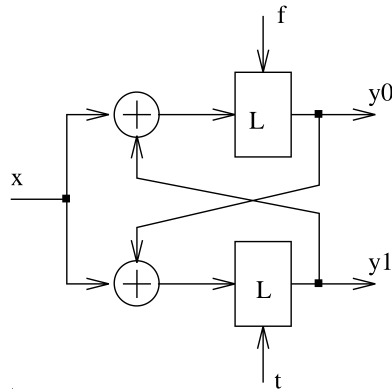

Figure 15. Circuit implementation for dual-rail Selector

This circuit, at such a modular level, looks very much like the one described in [30], except that the latter does not show the Full-Empty detector of the Stage Automaton while we “hide” the fact 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 14). The justification of these C-elements is provided by the transformation shown in Figure 16, which merges the mutex places of two adjacent stages.

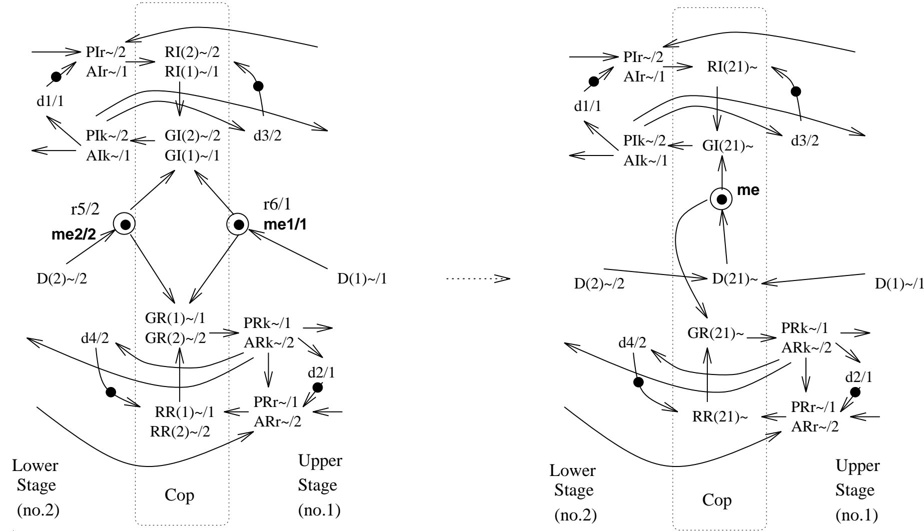

Figure 16. The effect of merging the mutex places of adjacent stages

In order to perform this transformation we should synchronise two neighbouring stages in the same way as we did in Figure 11. That is, thanks to the semantics of our design we can make pairs of events ($PIk\sim/2$,$AIk\sim/1$) and ($PRk\sim/1$, $ARk\sim/2$) identical. Here, the "/2" and "/1" subscripts are used for the "lower" and "upper" stages, respectively. We also identify the corresponding events of the two arbiters. Finally, the subsequent syntactic PN transformation is formally justified by the technique of merging two conflict places into one place shown in Figure 6(e).

At the intuitive level, the last transformation is also understandable since it should be sufficient to have only one arbitration unit ("cop") between two neighbouring stages. The events of this single arbiter are labelled with the subscripts 21, to designate that this arbiter is between stages 2 and 1. It should be clear now that the above-mentioned C-elements are obtained as the result of conversion of the new events $D(21)\sim$, which synchronise the "Done" events of the neighbouring stages.

We have thus derived a two-phase circuit which correctly implements the original specification. Indeed, if we now identify the transitions on signal wires AIk, PIk, ARk, PRk and G with names of actions AI, PI, AR, PR and G, then the behaviour of the obtained control circuit will be observationally equivalent to the TS shown in Figure 1(b).

In the next section we present a circuit solution obtained from the PN model in Figure 10 (c) using the four-phase signalling protocol.

6. Four-phase Implementation

In order to obtain a circuit by means of logic synthesis from its behavioural specification, we need to refine our PN specification in Figure 10(c) into an STG with four-phase signalling

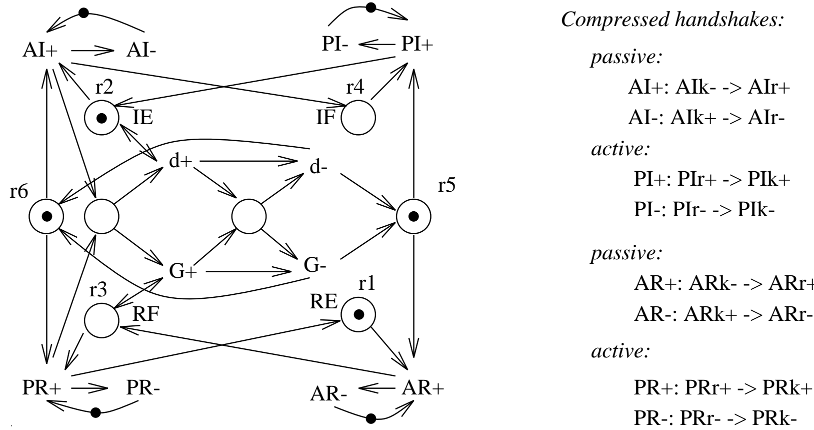

Figure 17. Four-phase refinement with ”compressed” handshakes

for the main four instruction and result pipe handshakes: AI, PI, AR and PR, as well as for the garner and skip control signals: G and d. The STG with such refinements is shown in Figure 17. For brevity, we have compressed each handshake pair into one signal, as shown in the right half of the figure. Here, the transitions of each signal model a pair of adjacent transitions of the associated handshake signals. This reduction is possible since, when we implement a handshake pair ar and ak, only one of these two signals is an output (ar for active handshake, ak for passive handshake), whose function is to be synthesised. The other signal, which is produced by the environment, can simply be seen as either a delayed version (ak, for active handshake) or inverted version (ar, for passive handshake) of the output.

This refinement is obtained by applying formal PN transformations from Figure 6. Its behavioural correctness can be traced by means of checking the observational equivalence of this STG and the original PN. The semantic identification can be done in the following way: $AI{+} \rightarrow AI$, $PI{+} \rightarrow PI$, $AR{+} \rightarrow AR$, $PR{+} \rightarrow PR$, $G{+} \rightarrow G$ and $d{+} \rightarrow d$.

Note that for easier checking we have preserved the names of places in this STG. In this net we have not yet inserted events for a mutual exclusion element to protect those settings and resettings. A more complete STG, in which request and grant signals for a *four-phase mutex element with enable* have been inserted, is shown in Figure 18. The description and a possible implementation of the mutex element is shown separately in Figure 19.

We have synthesised logic for the STG model in Figure 18 using petrify:

$$
\begin{aligned}
AI &= \overline{PI} \\
PI &= GI1 * \overline{GI2} + PI * (\overline{GI2} + GI1) = C(GI1,\ \overline{GI2}) \\
AR &= \overline{PR} \\
PR &= \overline{GR1} * GR2 + PR * (\overline{GR1} + GR2) = C(GR2,\ \overline{GR1}) \\
G &= PR * GI1 + GI1 * GR2 + PI * GR2
\end{aligned}
$$

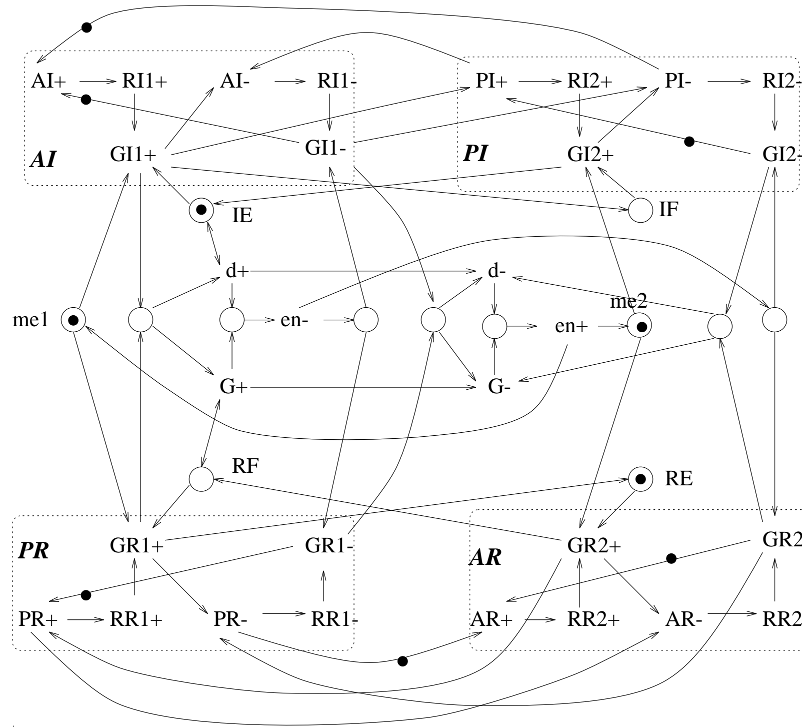

Figure 18. Four-phase STG refinement with mutex element signals

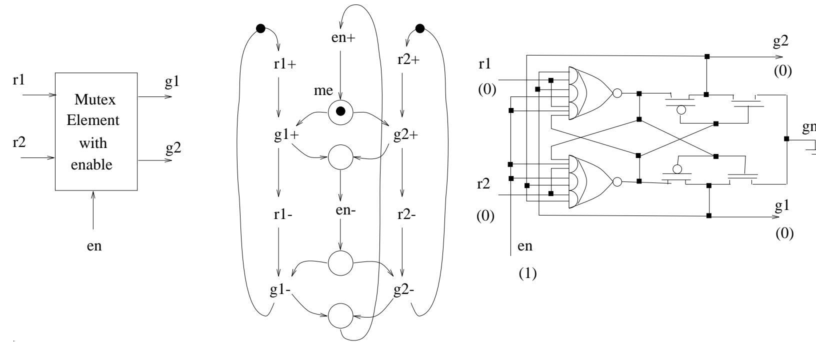

Figure 19. Mutual exclusion element with "enable" signal

$$
\begin{aligned}
d &= \overline{PI} * GR1 + \overline{PR} * GI2 + GI2 * GR1 \\
RI1 &= AI \\
RI2 &= PI \\
RR1 &= PR
\end{aligned}
$$

$$
\begin{aligned}
RR2 &= AR \\
GI1 &= RI1 * \overline{GR1} * en + GI1 * (en + RI1) \\
GI2 &= RI2 * \overline{GR2} * en + GI2 * (en + RI2) \\
GR1 &= RR1 * \overline{GI1} * en + GI1 * (en + RI1) \\
GI1 &= RI1 * \overline{GI1} * en + GI1 * (en + RI1) \\
en &= \overline{G} * \overline{d}
\end{aligned}
$$

The equations for the grant outputs of the mutex elements $GI1, GI2, GR1$ and $GR2$ correspond to the implementation shown in Figure 19, though petrify supplies them with a warning that these signals are not output-persistent. Due to the latter fact, we must use a special (analogue) CMOS transistor interconnection to resolve metastability.

The above equations effectively describe a CFPP stage as an "autonomous" circuit, which realises its handshakes for AI, PI, AR and PR in their compressed form. That is, depending on whether a particular handshake is active (PI and PR) or passive (AI and AR) the corresponding logic is either for the request (PIr and PRr) or acknowledgement (AIk and ARk) signal. Furthermore, we should bear in mind that, if we insert this implementation into the overall CFPP structure as an intermediate cell, we should merge corresponding pairs ($PI(i+1) = AI(i)$) and ($PR(i) = AR(i+1)$) of adjacent stages. Note that the numbering goes from the top stage down to the bottom, i.e. its ascending order coincides with the result pipe. Such a merge can be done at the logic equations level:

$$
\begin{aligned}
PI(i+1) &= GI(i+1) * \overline{GI(i)} * \overline{PI(i)} + PI(i+1) * (GI(i+1) + \overline{GI(i)} + \overline{PI(i)}) \\
&= C(GI(i+1),\ \overline{GI(i)},\ \overline{PI(i)}) \\
PR(i) &= GR(i) * \overline{GR(i+1)} * \overline{PR(i+1)} + PR(i) * (GR(i) + \overline{GR(i+1)} + \overline{PR(i+1)}) \\
&= C(GR(i),\ \overline{GR(i+1)},\ \overline{PR(i+1)})
\end{aligned}
$$

These are three input C-elements but we can notice that they have some redundancy (result of the merge) – input $\overline{GI(i)}$ in $PI(i+1)(= AI(i))$ and input $\overline{GR(i+1)}$ in $PR(i)(= AR(i+1))$. Indeed, for example, the value of $\overline{GI(i)}$ always changes before we have a corresponding transition on $\overline{PI(i)}$, which must be awaited by the C-element for $PI(i+1)(= AI(i))$ anyway. It is therefore possible to use:

$$
\begin{aligned}
PI(i+1) &= C(GI(i+1),\ \overline{PI(i)}) \\
PR(i) &= C(GR(i),\ \overline{PR(i+1)})
\end{aligned}
$$

The remaining functions (with corresponding subscripts), applicable for the circuit shown in Figure 20, can be written down easily from the above equations. This figure exemplifies a CFPP control circuit consisting of two stages. The Enable (en) signals for the mutex elements for adjacent stages can be built using C-elements. The justification for this is similar to the one we had for the two-phase implementation in Figure 16. In spite of the intuitive simplicity of this idea, its version for the four-phase case would be too cumbersome to show.

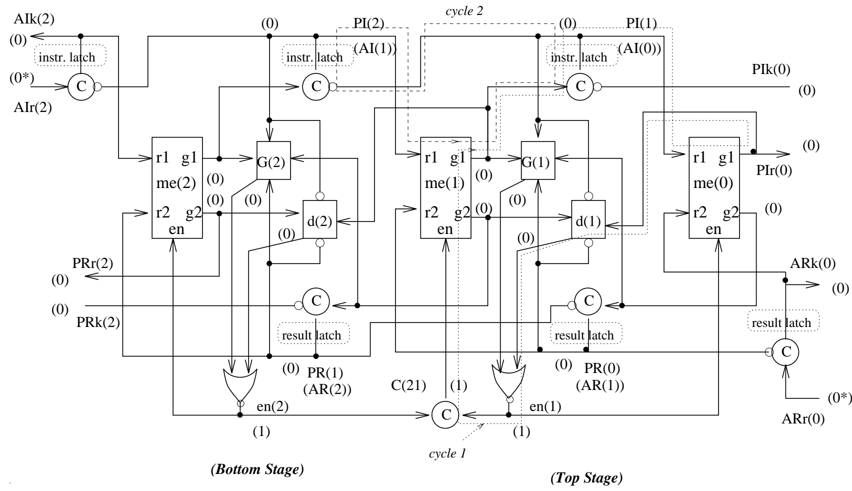

Figure 20. Four-phase circuit for a CFPP with two stages

We now conclude with a simple comparison of the two-phase and four-phase circuits. To make such a comparison let us use a similar interconnection of two stages in a two-phase implementation, shown in Figure 21. In both these circuits let us also show the place in which we are going to insert the hypothetical data path, i.e. latches for instructions and results. According to the semantics of our behavioural specification, such a place can be where we generate a request to pass the instruction (result) to the next stage. This is the outputs of the C-elements producing such requests. Thus, if we break the output wire of each such C-element, the latch can be inserted into this break, where the first end of the wire can be connected to a latch load (request) signal and the other end to the completion (acknowledgement) signal. If the latch does not produce a completion signal, we can insert an appropriate "scaling delay" element into that wire, which should be of sufficient value to emulate the delay of the latch. Let us assume for simplicity that for a four-phase implementation we can use a classical D-latch with a four-phase control signal. For a two-phase circuit, we may resort to event-based latches. Examples of such latches can be found in [31, 10, 11].

In both circuits we can identify *critical cycles*, i.e., cycles with the longest cumulative delay. Critical cycles can be found in an unfolded event graph (based on a Petri net unfolding [27]), in which signal transitions are annotated with delays of corresponding gates, macromodules or interconnections. To avoid dealing with processes with alternatives and arbitration, we shall only consider the performance for the case of propagating one type of data (say, instructions). This "restriction" should not affect the result of comparison since both circuits have a symmetric structure (between instruction and result pipes). They both produce the same functionality (synchronisation between the counterflow pipes) and their speed is determined by the propagation delays in the pipes. Let us assume for simplicity that the same types of elements in different stages have exactly the same delay. The critical cycle has to be found between a given pair of adjacent stages. Two candidates to be a critical

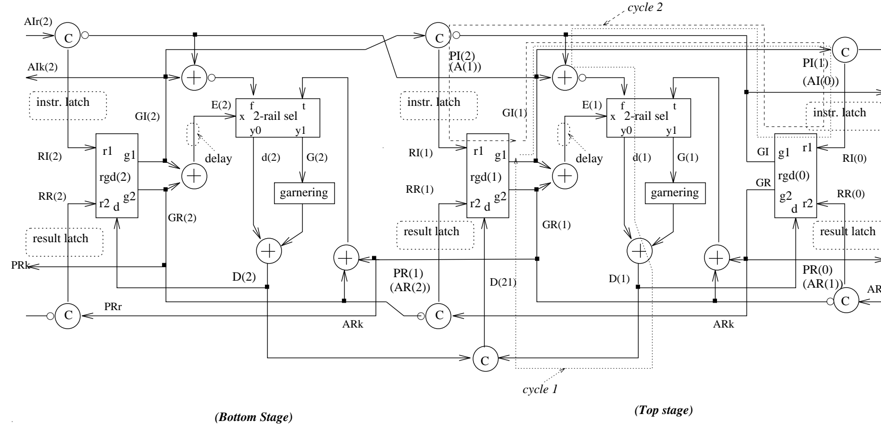

Figure 21. Two-phase circuit for a CFPP with two stages

cycle are shown with dotted and dashed lines in each of Figures 20 and 21. Which of those two is critical depends on the actual delay values. The cycle shown with a dashed line is dominated by the data path latch delays and arbiter delays (this cycle passes through two such latches and two arbiters), while the one with a dotted line goes predominantly through control logic (arbiter and pipe synchronisations elements).

Let us for example compare the cumulative delays of the "dashed" cycles.

The cycle in the two-phase circuit has the following cumulative delay:

$$T_{2ph} = 2t_{DL2} + t_{HS} + 2t_{RGD} + 2t_C,$$

where T_{DL2} is the delay of a two-phase data-latch, t_{HS} is the delay of a request-acknowledge "round-trip" between two stages, t_{RGD} is the delay of an RGD arbiter when issuing a grant without waiting (recall that we have assumed that instructions flow without "interference" from results), and t_C is the delay of a C-element. One of the known speed-independent implementations of an RGD arbiter [9] (too pessimistic though!) has the following delay:

$$t_{RGD} = 2t_L + t_{XOR} + t_{ME},$$

where t_L is the delay of a standard (transparent) D-latch (described by equation $Q = DC + (D + \overline{C})Q$, where D is data input and C is control input), t_{XOR} is the delay of an XOR gate, and t_{ME} is the delay of a standard two-way mutex element built on an SR flip-flop (e.g., two cross-coupled NANDs) and a transistor-based metastability detector [9]. Note, however, that the implementation of an RGD arbiter with a single "Done" signal includes an additional delay of logic to synchronise the outgoing "Grants" with the "Done", e.g., in the most optimistic case, at least the delay of a C-element. Then, since we have no arbitration condition, the delay of a mutex is effectively equal to that of a latch. Similarly, we can put $t_C = t_L$. Thus,

$$T_{2ph} = 2t_{DL2} + t_{HS} + 10t_L + 2t_{XOR}.$$

The corresponding cycle in the four-phase circuit has the following cumulative delay:

$$T_{4ph} = 2(2t_{DL4} + t_{HS} + t_{MEE} + 2t_C),$$

where t_{DL4} is the delay of a four-phase data-latch, t_{HS} is again the delay of the total handshake wire delay between stages, t_{MEE} is the delay of a mutex element with enable input, and t_C is the delay of a C-element. Note however that, due to the use of four-phase signalling, we have to pass along this cycle twice, to make our comparison with the two-phase design fair. Hence the factor of 2 in the above expression.

Let us compare these delays. Not being too pessimistic about the four-phase case, we can assume that the delay of a two-phase data-latch is twice that of a four-phase one. Since the standard mutex element is built out of slightly simpler gates, we can (very pessimistically) assume that $t_{MEE} = 1.5t_{ME}$. Now, assuming also $t_C = t_L$, we have

$$T_{4ph} = 2t_{DL2} + 2t_{HS} + 7t_L$$

.

We should therefore tradeoff the terms of the following difference:

$$T_{2ph} - T_{4ph} = 3t_L + 2t_{XOR} - t_{HS},$$

which will in most cases be in favour of the four-phase design. A similar sort of conclusion can be drawn if we examine the other two candidates for critical cycles (shown with dotted lines). Indeed, without simplifying the implementation of components in the two-phase circuit the latter will be slower and occupy more area than the four-phase one. This is no surprise since it is the result of compilation of the specification into a net of macromodules, whereas the four-phase logic is the product of logic synthesis with minimisation. In the two-phase solution, one can of course implement an RGD arbiter with one “Done” by using special techniques at the transistor level (e.g., a Propeller Arbiter by C. Molnar) or by sacrificing some of its speed-independence (there are less conservative designs of Sequencers [24]).

Finally, as a soothing remark to the two-phase circuit, we should mention that the above four-phase circuit has the so-called non-dense pipeline structure. This means that every two consecutive data items, either in the instruction or results pipe, must be separated by at least one “bubble” (no data). It is thus impossible to store two instructions in the two adjacent stages, since we need at least one stage to carry out the resetting phase of the four-phase protocol. This should not however be regarded as a shortcoming of the four-phase method – e.g., one can add extra control logic (state signals) and implement the resetting phase within the same stage (the reader interested in control of dense pipelines is referred to [32, 11]).

7. Discussion and conclusions

We have formally derived circuits for CFPP stage control from the initial state-based specification presented in [30]. This required us to follow: (1) transformations at the Transition System level and synthesis of a Petri net from a semi-elementary TS; (2) refinements at the Petri net level and synthesis of circuits from Petri nets and Signal Transition Graphs.

The transformations at stages (1) and (2) are backed up by the notion of observational equivalence, which guarantees behavioural correctness of the design process. At the same time, some semantic and even heuristic issues are involved in this process at various stages. Namely, at stage (1), we may need to insert auxiliary (dummy) events to make the original state graph a semi-elementary TS. At stage (2), doing our handshake signal refinement of the original actions we may "shuffle" the relative order of the resetting phases of signal [34]. Those different orderings may affect the Complete State Coding (CSC) property, and thus require an extra state signal insertion. These issues are closely related with the aspects of performance and area. For example, our four-phase signal refinement has been chosen rather simple in order to avoid solving the CSC problem – we simply "interleave" the resetting phase of the AI handshake in any (i-th) stage with the setting phase of AI in the next ($i-1$-th) stage (similar for the results pipe). This can be observed from the ordering $GI1+ \rightarrow PI+ \rightarrow AI - \ldots$ in the STG of Figure 18, bearing in mind that handshake PI in the given stage is semantically identical to handshake AI of the next stage. The other part of this interleaving is $GI1- \rightarrow PI- \rightarrow AI + \ldots$, which allows the current stage to latch the new instruction only if the latch in the next stage has been reset to 0 (i.e., the previous instruction has been moved at least one stage after the next one). The cost of such a simple option has been that our pipeline is non-dense.

Due to the size constraints imposed by the logic synthesis tools (e.g., petrify currently needs about 24 hrs to find an implementation, in complex gates, for an STG of up to 25 signals), 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. For example, in our four-phase design, we implemented logic for an "autonomous" stage (15 signals), and had to modify the equations when putting the stage as an intermediate one in the overall CFPP structure. In this case, it was a fairly trivial transformation. Often, however, this is not so obvious, and the designer may need to verify the composed circuit against the composite STG specification.

Note that verification may also be needed at an earlier phase, to verify the observational equivalence between TSs or between PNs if the transformations applied are not those which are correct by construction (Figure 6). The results of refining net models with abstract transitions into those with four-phase handshake and mutex may also need checking for signal transition consistency, behavioural equivalence and 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 paper, we have only briefly addressed performance analysis issues. 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 [13, 26, 20, 28].

Acknowledgments

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 are to Luciano Lavagno and Jordi Cortadella for useful discussions about constructing a Petri net model of the CFPP example and their kind help with SIS and petrify. I also highly appreciate help of Alex Semenov and his software tool PUNT (Petri net analysis based on unfoldings) in verifying the numerous Petri net models produced in the course of this work. Last but not least, many thanks to the three anonymous reviewers for their constructive comments, which helped improving the paper immensely.

The UK Engineering and Physical Science Research Council (EPSRC) partially supported this research through grants GR/J52327 and GR/K70175.

Notes

1. As was noted in [30], 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.
2. Unless specified otherwise, terms “action” and “event” are considered equivalent in this text.
3. The term “Transition System” is used as a synonym to “State Graph”. Only if it may cause confusion, we will be applying the latter term in a more specific sense than the former. Namely, a State Graph is a Transition System which has binary encoding. This follows the terminological tradition established in the asynchronous design community.
4. In such a protocol, both the rising and the falling edges of a signal have equal significance from the semantical point of view.
5. Here, the process control semantics of the rising and falling edges of a signal is different. E.g., only the rising edge can be significant, say, to stand for the fact that data in the data path is valid, while the other edge carries out only a “resetting” function.
6. 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 of tokens.
7. We even 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.
8. We say ”effectively” because formally elementary nets are defined in a slightly different way (see, e.g., [21]), but any elementary net can be converted into a behaviourally equivalent 1-safe net, by possibly adding complementary places [4].

References

1. 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*, 1991, pp. 384 – 389.
2. L. Bernardinello, G. De Michelis, K. Petruni and S. Vigna, “On Synchronic Structure of Transition Systems,” Universita di Milano, 1994.
3. R.K. Brayton, G.D. Hachtel, C.T. McMullen, A. Sangiovanni-Vincentelli, *Logic Minimisation Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Hingham, MA, 1984.
4. J. Cortadella, M. Kishinevsky, L. Lavagno and A. Yakovlev, “Synthesizing Petri Nets from State-Based Models,” Technical Report RR 95/09 UPC/DAC, Universitat Politecnica de Catalunya, April 1995.
5. J. Cortadella, M. Kishinevsky, L. Lavagno and A. Yakovlev, “Synthesizing Petri nets from state-based models,” pp. 164–171, November 1995.
6. J. Cortadella, A. Kondratyev, M. Kishinevsky, L. Lavagno and A. Yakovlev, “Complete state encoding based on theory of regions,” *Proc. Sec. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, Aizu, Japan, IEEE Comp. Soc. Press, March 1996, pp. 36–47.

7. J. Cortadella, A. Kondratyev, M. Kishinevsky, L. Lavagno and A. Yakovlev, "Methodology and tools for complete state coding in STG-based synthesis of asynchronous circuits," in *Proc. DAC'96*, Las Vegas, June 1996, pp. 63–66.
8. J. Cortadella, A. Kondratyev, M. Kishinevsky, L. Lavagno and A. Yakovlev. "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *Proc. Eleventh Conf. Design of Integrated Circuits and Systems (DCIS'96)*, Barcelona, November 1996, pp. 205–210.
9. J. Cortadella, L. Lavagno, P. Vanbekbergen and A.Yakovlev, "Designing asynchronous circuits from behavioural specifications with internal conflicts," *Proceedings of First Int. Symp. on Adv. Res. in Asynch. Circ. and Syst.*, Salt Lake City, Utah, IEEE Comp. Soc. Press, November 1994 ,pp. 106 – 115.
10. P.Day and J.V.Woods, "Investigation into micropipeline latch design styles," *IEEE Transactions on VLSI Systems*, 3:264-272, June 1995.
11. S. Furber and P. Day, "Four-Phase Micropipeline Latch Control Circuits," *IEEE Transactions on VLSI Systems*, 4:247 – 253, June 1996.
12. S. Furber, P. Day, J.D.Garside, N.C. Paver and J.V. Woods, "AMULET1: A micropipelines ARM," *Proceedings of VLSI'93*, Grenoble, France, September 1993, Best Paper Award.
13. H. Hulgaard and S.M. Burns, "Bounded delay timing analysis of a class of CSP programs with choice", *Proceedings of Int. Conf. on Adv. Res. in Asynch. Circ. and Syst.*, Salt Lake City, Utah, November 1994, pp. 2 – 11.
14. M. Kishinevsky, A. Kondratyev, A. Taubin and V. Varshavsky, *Concurrent Hardware: The Theory and Practice of Self-Timed Design*, John Wiley and Sons, London, 1993.
15. W.H.F.J. Körver and I.M. Nedelchev, "An Asynchronous Implementation of SCPP-A," Technical Report CSRG95-07, University of Surrey, Guildford, July 1995.
16. P.G. Lucassen and J.T. Udding, "On the Correctness of the Sproull Counterflow Pipeline Processor," in *Proceedings of the Sec. Int. Symp. on Adv. Res. in Asynch. Circ. and Syst.*, Aizu-Wakamatsu, Japan, IEEE Comp. Soc. Press, March 1996 ,pp. 112–120.
17. R. Milner, *Communication and Concurrency*, Prentics Hall International, Series in Computer Science, London, 1989.
18. C.E. Molnar and H.M. Schols, "The Design Problem SCPP-A," Technical Report TR-95-49, Sun Microsystems Laboratories, Mountain View, CA, December 1995.
19. T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, April 1989, pp. 541 – 580.
20. C.J.Myers and T.H.-Y. Meng, "Synthesis of timed asynchronous circuits," *IEEE Transactions on VLSI Systems*, 1: 106 –119, June 1993.
21. M. Nielsen, G. Rozenberg and P.S. Thiagarajan, "Elementary transition systems," *Theoretical Computer Science*, 96:3-33, 1992.
22. S.S. Patil, "Cellular Arrays for Asynchronous Control," in *Proc. 7th Annual Workshop on Microprogramming*, 1974 (also Computation Structures Group Memo. 122, Project MAC, MIT, April 1975).
23. S.S. Patil and J.B. Dennis, "The description and realization of digital systems," in *Proceedings of the IEEE COMPCOM*, 1972, pp. 223 – 226.
24. P. Patra and D.S. Fussell, "Efficient building blocks for delay insensitive circuits," in *Proceedings of First Int. Symp. on Adv. Res. in Asynch. Circ. and Syst.*, Salt Lake City, Utah, IEEE Comp. Soc. Press, pp. 196 – 205, November 1994.
25. M. Pietkiewicz-Koutny and A. Yakovlev, "Non-pure nets and their transition systems," Technical Report Series, no. 528, Department of Computing Science, University of Newcastle upon Tyne, September 1995.
26. T.G. Rokicki, "Representing and Modeling Digital Circuits," Ph.D. thesis, Stanford University, 1993.
27. A. Semenov and A. Yakovlev, "Combining partial orders and symbolic traversal for efficient verification of asynchronous circuits," *Proc. IFIP Int. Conference on Computer Hardware Description Languages, (CHDL'95)*, Chiba, Japan, August-September 1995, pp. 567–573.
28. A.Semenov and A. Yakovlev. "Verification of asynchronous circuits based on timed Petri net unfolding," *Proc. 33rd Design Automation Conference DAC'96*, Las Vegas, June 1996, pp. 59–63.
29. 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.
30. 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.
31. I. E. Sutherland, "Micropipelines," *Communications of the ACM*, 32:720-738, Turing Award Lecture, June 1989.

32. V. I. Varshavsky, M. A. Kishinevsky, V. B. Marakhovsky, V. A. Peschansky, L. Y. Rosenblum, A. R. Taubin and B. S. Tzirlin, *Self-timed Control of Concurrent Processes*, Kluwer Academic Publishers, Boston/Dordrecht/London, 1990, (Russian edition: 1986).
33. A. Yakovlev, “Designing Control Logic for Counterflow Pipeline Processor Using Petri nets,” Technical Report no. 522, Department of Computing Science, University of Newcastle upon Tyne, May 1995.
34. 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.
35. A. Yakovlev, A. Koelmans and L. Lavagno. “High Level Modelling and Design of Asynchronous Interface Logic,” Technical Report Series, no.460, University of Newcastle upon Tyne, Department of Computing Science, November 1993.