Innovations Syst Softw Eng (2005) 1: 25–40
DOI 10.1007/s11334-005-0002-4

ORIGINAL PAPER

Antonio Cerone · George J. Milne

Property verification of asynchronous systems

Received: 29 September 2004 / Accepted: 15 January 2005 / Published online: 11 March 2005

Abstract We demonstrate a new modelling technique that facilitates the description and the formal verification of timing properties of concurrent systems, such as asynchronous digital hardware. We utilise a process algebra and its associated automatic verification system and construct models and verification strategies using them. Utilising the hierarchical nature of our approach, these techniques may then be applied to larger systems, such as asynchronous circuits of commercial complexity. The modelling techniques introduced permit four distinct classes of objects, namely system components, assumed constraints on their behaviour, properties requiring proof and behaviour refinements, all to be modelled by a process. We illustrate this approach by modelling the necessary timing relationships required for the correct operation of asynchronous micropipline stages and then verifying that the resulting behaviour is correct. Finally, we demonstrate how the same models are used to make some observations about the performance and the timing properties of such designs.

Keywords Formal verification · asynchronous hardware · process algebra · constraint modelling

1 Introduction

The ability of a modelling formalism to accurately represent timing information is becoming increasingly significant for the design of concurrent systems, such as those involving communication protocols and asynchronous digital logic. Additional circuitry is usually required if a circuit that makes no assumptions about signal transition timing is to be designed to work correctly. In contrast, circuits that are designed utilising timing information may give superior performance; however, this requires the ability to accurately specify and verify the necessary timing properties to ensure that those asynchronous designs will behave correctly.

In this paper, we address the modelling of timing properties using the Circal process algebra [21,22,29]. Specifically, we illustrate new techniques for constructing processes that describe timing properties and timing relationships. The approach adopted here, presented in a preliminary form in previous papers [7,11], avoids the state-explosion problem found when modelling time by using sequences of ticks to model delays [2]. In the design of asynchronous circuits, for example, the absolute duration of the delay intervals are less important than the relationships between them. As an example, consider Sutherland's asynchronous micropipelines [28]. In a micropipeline, a major consideration for its correct operation is to ensure that the delay in the control paths is longer than the delay in the data path. Our ability to express such a relationship using an interval constraint technique avoids the state-explosion issue that is associated with explicit timing models [2,21].

This paper also illustrates how one class of object in our modelling formalism, namely a process, is used to model conceptually distinct objects that are required in the verification of concurrent systems where timing is significant, such as with asynchronous-logic designs. These objects are the system components, such as logic gates, the behavioural constraints, such as assumed timing relationships, the behavioural properties that require being proven and that specify the correctness of the design, and behaviour refinements, which introduce new actions to highlight the behavioural aspects relevant to the property to be verified. These distinct processes are manipulated by the Circal System [22,29], where they get composed by the concurrent composition operator and where the behavioural equivalence between processes is determined by an equivalence checker, the basis of the Circal automatic-verification technique.

Unlike other approaches, we do not produce an application-specific customised version of an existing process algebra, as Joseph and Udding [18] do with CSP [19], for

A. Cerone (✉)
International Institute for Software Technology
United Nations University, Macau SAR China
E-mail: antonio@iist.unu.edu
G. J. Milne
School of Computer Science and Software Engineering
The University of Western Australia, Perth, Australia
E-mail: george@csse.uwa.edu.au

example. Rather, we develop suitable modelling techniques that exploit the underlying primitives of the Circal process calculus to build appropriate models and proof techniques for the class of asynchronous systems under investigation.

Process algebras, such as Circal, CCS [23], CSP [19], LOTOS [4], the π-calculus [24] and DI Algebra [18] are powerful yet primitive. Their elegance results from the presence of only a limited number of primitive operators; that is their simplicity. But to utilise them practically requires the development of suitable modelling and verification methodologies that sit above the primitive constructs of the respective formalisms. This need for modelling and proof infrastructure can also be found in assertional formalisms, such as in the use of higher order logic for hardware specification and verification, as exemplified in the HOL system [17]. The research reported in this paper illustrates the type of techniques necessary for using a process algebra, such as Circal, to rigorously analyse timing relationships such as found in asynchronous circuits.

This paper introduces techniques for modelling timing relationships in process algebra and demonstrates the appropriateness of such techniques using two micropipeline examples. It also introduces a technique for determining performance by establishing the concurrency of activity among system components rather than by using absolute numeric values. Properties relating the simultaneity of activity can be represented as processes and these performance properties can be verified in the same way as behavioural correctness properties.

The Circal process algebra has distinctive features that allow the integration of correctness, timing and performance properties [7]. The constraint-based modelling [30] methodology made available in Circal [22] supports both a natural representation of timing constraints [9,7] and a verification procedure that is entirely performed within the process algebra framework [6,11]. The use of sets of simultaneous actions within guards, a distinctive feature of the Circal formalism, supports the characterisation of simultaneity and sequentiality between components, which is essential in the definition of performance properties.

Formal methods have tended to concentrate on the verification of the behavioural correctness of software and/or hardware systems. However, it may be argued that verification should also include the rigorous analysis of the performance of a system. This is especially the case for those application domains, such as hardware systems, where performance plays a key role in the choice between alternative technologies.

Two major modelling approaches that have been used for both verification and performance analysis are timed Petri nets and timed process algebra [14]. A standard method to introduce performance analysis into these two formalisms is to associate time with the actions of the process algebra and the transitions (or equivalently the residence time of the places) of the Petri net. These times could be either deterministic or stochastic; in the analysis of the former, we can use max-plus algebra [1], while the latter utilises stochastic Petri nets and stochastic process algebra. Performance analysis is then based on a derived Markov chain.

Adding time in this way to both Petri net and process algebra models increases the complexity of the analytical procedure compared with the untimed case. For example, Markov chain analysis is particularly restricted by the state-explosion problem. Some Petri net methods allow the abstraction of time from the model where it is not significant, by having zero time transitions, but most stochastic process algebras do not allow abstraction with zero time transitions.

In previous papers [7,11], we introduced our approach to the integrated verification of correctness, timing and performance properties in concurrent systems, using the Circal process algebra and its mechanisation, the Circal System [22,29]. In this paper, we give a thorough presentation of our specification and verification methodology as applied to distinct micropipeline designs. In our performance analysis, we do not model explicit timing properties but rather determine performance characteristics in a more abstract fashion, in terms of the degree of parallelism achieved among the system components, so leading to a quantitative evaluation of the throughput of the system.

In Sect. 2, we introduce the Circal process algebra. In Sect. 3, we discuss our approach to modelling time using the notion of timing intervals. In Sect. 4, we formally specify the behaviour of an asynchronous micropipeline design as well as construct a description of its implementation using the Circal composition operator. The specific environmental constraints and assumptions that are applied to each micropipeline stage are also modelled as processes. In Sect. 5, we show how to formally verify the correctness of a micropipeline stage. Then we analyse the performance features of both designs under investigation and verify that the more complex design improves micropipeline performance, as claimed by Furber and Day [15]. Finally, we show how to formally verify that specific timing requirements are necessary for the correct operation of the micropipeline.

2 The process algebra

Process algebras are mathematical formalisms for describing systems of interacting finite-state machines (FSMs). The interaction is given by synchronising the transitions that occur in different FSMs. This can be done in several ways, which differentiate the many process algebras that appear in the literature [4,19,22,23].

2.1 A hierarchy of processes

Every process algebra has one (or more) *parallel composition operator(s)*, a *hiding* operator and a *relabelling* operator. The combination of these operators allows for the structure of a system to be modelled as a hierarchy of abstraction levels, as shown in Fig. 1a.

Every box represents a process and is decomposed into the components at the lower level to which it is connected. For

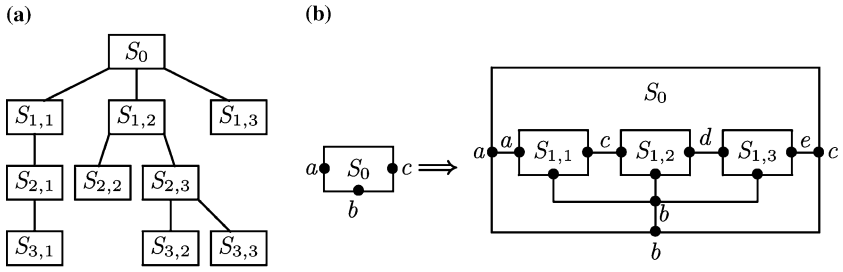

Fig. 1 **a** Hierarchy of system components; **b** graphical representation of parallel composition with hiding and relabelling

example, process $S_{1,2}$ consists of two components: process $S_{2,2}$ and process $S_{2,3}$. Only the boxes that are leaves of the hierarchy, not necessarily at the lowest level, explicitly encapsulate behaviours ($S_{1,3}$, $S_{2,2}$, $S_{3,1}$, $S_{3,2}$ and $S_{3,3}$). In the following, we will call such leaves *behavioural processes*. Every process interacts with other processes through communication ports. Interaction between processes occurs through the actions that are associated with the ports. In Circal [22], CSP [19] and LOTOS [4], a communication channel connects all the ports that are labelled with the same action. In CCS [23], actions are coupled in complementary pairs (input and output actions) and a directed communication channel connects the two ports that are labelled with complementary actions. In our approach, we adopt the communication paradigm of Circal, CSP and LOTOS. Rather than formally introducing one of these process algebraic languages, we will present all necessary concepts in an intuitive graphical fashion.

If we look at the hierarchy given in Fig. 1a from a bottom-up point of view, every process, apart from the root, is embedded within the parent process by composing it in parallel with other processes; by hiding some of the actions that are used for interaction and possibly by relabelling other actions. For example, $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$ are embedded within S_0, as shown in Fig. 1b. Communication ports are represented as bullets on the outline of the box. Communication channels are represented by lines connecting two ports, when only two ports are involved in the communication, or by a bullet connected with all the ports involved in the communication, when the involved ports are more than two. The action associated with a port is written next to the port, if that port is not involved in any communication at the given abstraction level, next to the bullet or to any line describing the communication in which the given port is involved, otherwise. The embedding of a set of processes within the next abstraction level is represented by a box surrounding the set of processes, with new bullets (with the corresponding actions, which may be relabelled, written next to them) on its boundary to represent all the ports that are not hidden after the composition. These new bullets are connected by lines to any of the corresponding internal bullets. For example, in Fig. 1b, $S_{1,1}$ and $S_{1,2}$ communicate through the channel labelled by action c, $S_{1,2}$ and $S_{1,3}$ through the channel labelled by action d and $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$ through the three-way channel labelled by action b. Actions c and d are hidden in S_0; a and b are visible at the next abstraction level as ports of S_0; e is relabelled with c as a port of S_0.

In this way, the box that embeds a set of processes represents the *interface* of the composite process. Every process has a *sort*, which is the set of action names that label the ports on the box that embeds its components. For example, in Fig. 1b, S_0 and $S_{1,1}$ have sort $\{a, b, c\}$, $S_{1,2}$ has sort $\{b, c, d\}$ and $S_{1,3}$ has sort $\{b, d, e\}$. For a behavioural process, its sort (or interface) must contain at least all the actions that occur in the embedded behaviour.

2.2 Process behaviour

The parallel composition of behavioural processes may be expanded into a global behaviour. Every state of the global behaviour is given by the product of component states, one for each component. The precise semantics of parallel composition depend on the specific process algebra involved. In this paper, we consider the approach adopted by the Circal process calculus [22], where every transition between states is labelled with a (possibly empty) set of actions.

Given a set of processes and a set of transitions, one for each process, the transitions of the set *may synchronise* if and only if, for each action that belongs to the label of at least one transition, the action does not occur in the label of a transition of the given set, then it does not belong to the sort of the process to which such a transition belongs; here, causally independent actions are synchronised. If the transitions of the set may synchronise, then some of these transitions *must synchronise* if and only if there is at least one action in the intersection of their labels; here, identical actions from distinct component processes synchronise. When transitions of different processes synchronise, the label of the transition of the composite process is the union of the labels of all components.

For instance, if we compose $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$ in Fig. 2a–c, then the transition labelled by $\{a, c\}$ from $A_{1,1}$ to $A_{1,1}$ in $S_{1,1}$, the transition labelled by $\{c, d\}$ from $A_{1,2}$ to $B_{1,2}$ in $S_{1,2}$, and the transition labelled by $\{d\}$ from $A_{1,3}$ to $B_{1,3}$ in $S_{1,3}$, may synchronise. The corresponding transition of the composite process, which is represented in Fig. 2d, is from

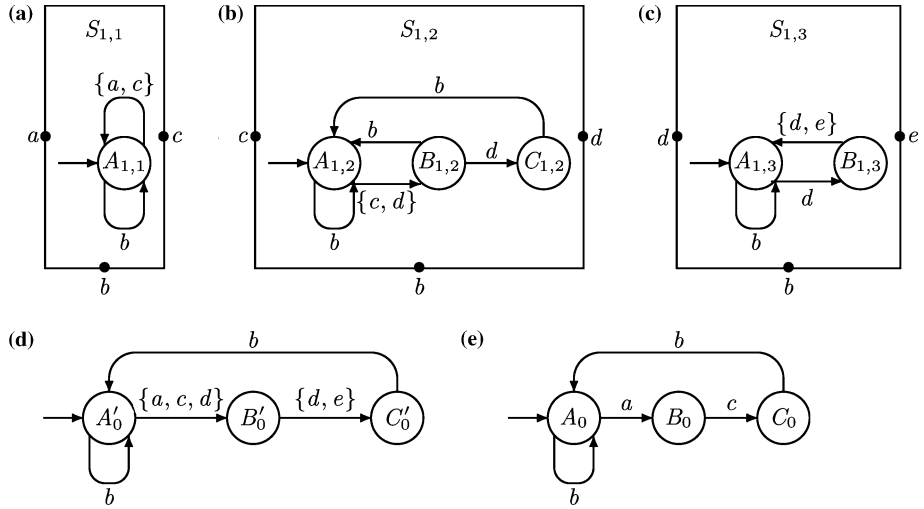

Fig. 2 **a–c** Interfaces and behaviours of $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$; **d** behaviour of the parallel composition of $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$; **e** behaviour of S_0

$A_{1,1} \times A_{1,2} \times A_{1,3}$ to $A_{1,1} \times B_{1,2} \times B_{1,3}$ and is labelled by $\{a, c, d\} = \{a, c\} \cup \{c, d\} \cup \{d\}$. Notice that, in Fig. 2a–c, we have represented the sets that consist of a single action by writing just the action name without brackets.

The behaviour of the composition of $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$ is given in Fig. 2d. Here, $A'_0 = A_{1,1} \times A_{1,2} \times A_{1,3}$, $B'_0 = A_{1,1} \times B_{1,2} \times B_{1,3}$ and $C'_0 = A_{1,1} \times C_{1,2} \times A_{1,3}$. Notice that the transition labelled by $\{a, c\}$ from $A_{1,1}$ to $A_{1,1}$ in $S_{1,1}$ must synchronise with the transition labelled by $\{c, d\}$ from $A_{1,2}$ to $B_{1,2}$ in $S_{1,2}$. Analogously, the transition labelled by $\{c, d\}$ from $A_{1,2}$ to $B_{1,2}$ in $S_{1,2}$ must synchronise with the transition labelled by $\{d\}$ from $A_{1,3}$ to $B_{1,3}$ in $S_{1,3}$.

After hiding c and d and relabelling e with c, we obtain the behaviour of S_0, which is given in Fig. 2e.

2.3 Constraint-based modelling in Circal

In this section, we give a brief description of the Circal operators and their semantics and refer the reader to [2,21,22, 25] for further explanations. The syntax of Circal processes is summarised by the following BNF expressions, where Γ is a guard (consisting of single action `a` or the set of actions in M), P is a process, D a process definition, $\mathcal{A}$ is the set of possible actions and `a`, `b` $\in \mathcal{A}$ and I is a process variable:

$$M ::= \texttt{a} \mid \texttt{a}\ M$$
$$\Gamma ::= \texttt{a} \mid \texttt{(}M\texttt{)}$$
$$P ::= /\backslash \mid \Gamma P \mid P \texttt{ + } P \mid P \texttt{ \& } P \mid I \mid P \texttt{ * } P \mid P \texttt{ - } M \mid P\texttt{(}M\texttt{)} \mid P \texttt{ [a/b]}$$
$$D ::= I \leftarrow P$$

This syntax is a simplified form of the syntax implemented in the Circal System [29].

Each Circal process has a *sort* associated with it, which specifies the set of actions (ports) through which it may interact with other processes. Every sort will be a nonempty subset of $\mathcal{A}$, the collection of all available actions.

The $/\backslash$ constant represents a process that can participate in no communication. This is a process that has terminated or deadlocked.

In ΓP, the P process may be guarded by sets of simultaneously occurring actions. This is a key feature of Circal, which greatly enriches the modelling potential of the algebra in contrast with process algebras such as CSP [19], CCS [23] and LOTOS [4], which only permit a single action to occur at one computation instant.

A name can be given to a Circal process with the *definition* operator ($\leftarrow$). Recursive process definitions, such as $P \leftarrow \Gamma P$, are permitted.

The + operator defines an *external choice*, which is decided by the environment where the process is executed, whereas the & operator defines an *internal choice*, which is decided autonomously by the process itself without any influence from its environment. Internal choices appear to an external observer as *nondeterminism*.

Given processes P and Q, the term $P * Q$ represents the process that can perform the actions of the subterms P and Q together (*composition*). Any synchronisation that can be made between two terms, due to some atomic action being common to the sorts of both subterms, must be made; otherwise, the actions of the subterms may occur asynchronously.

The terms $P - M$ and P `[a/b]` define *abstraction* and *relabelling*, respectively, in the usual way.

Using the textual syntax of Circal, the behaviour described in Fig. 2d is defined as follows:

$$A'_0 \leftarrow (a\ c\ d)\ B'_0 \ +\ b\ A'_0$$
$$B'_0 \leftarrow (d\ e)\ C'_0$$
$$C'_0 \leftarrow b\ A'_0$$

The composition operator of Circal provides synchronisation among an arbitrary number of processes without the removal of the synchronising events in the resultant behaviour.

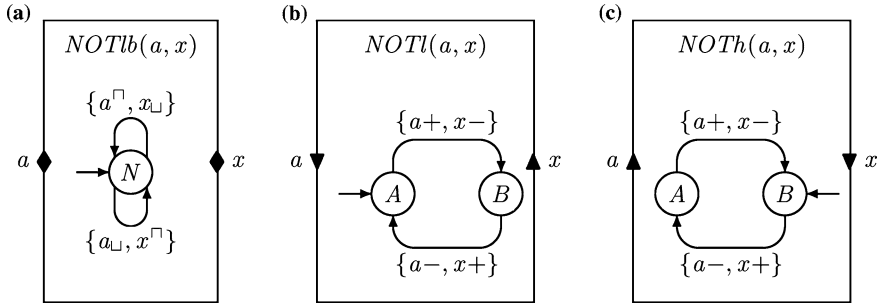

Fig. 3 **a** Level-based NOT gate; **b** transition-based NOT gate with input port initially low; **c** transition-based NOT gate with input port initially high

This particular nature of the composition operator is exploited by the *constraint-based* modelling methodology, which has been used in several application domains, such as communication protocols [5,6], safety-critical systems and asynchronous hardware [8].

When a process S of sort L is composed with a process C of sort L', such that $L \cap L' \neq \emptyset$, that part of the behaviour of S whose restriction to $L \cap L'$ is not consistent with the behaviour of C does not appear in the behaviour of $S * C$. This is equivalent to saying that C *constrains* S. As an example, consider the process P of sort $\{a, b, c\}$ and the process C of sort $\{a, b, d\}$ defined as follows:

$$P \leftarrow a\ P\ +\ b\ P\ +\ (a\ b)\ P\ +\ c\ P$$

$$C \leftarrow (a\ b)\ C\ +\ d\ C$$

The intersection of the sorts is $\{a, b\}$ and C constrains P to always perform a and b simultaneously.

Term $P\ (M)$ defines a *process-generation function*, which can be instantiated as an actual process by replacing formal parameters in M with actual parameters.

2.4 Process models of distinct system artefacts

In our verification methodology, we utilise the core modelling object, namely a *process*, to model four quite distinct artefacts. The first of these is when we model the *physical components* of the system under investigation.

The second artefact that we model by a process, or processes, are *assumptions* on the behaviour of the system. These assumptions usually relate to context or environmental restrictions and generally simplify the system behaviour when composed with it using the composition operator.

The third artefact that is modelled by a process is the specific *property*, which we want to verify as holding in the system and which is used to describe the notion of system correctness.

The fourth artefact that is also modelled by a process is a *refinement* of part of the behaviour of another process. By composing a given process with a refinement process, we extend the behaviour of the given process. Together with the hiding operator, this allows the definition of a new *view* of the system. In this paper, we will consider only a single type of refinement, namely a *time-interval refinement*. Processes may also model different artefacts at the same time; we will see processes that are both refinements and assumptions.

2.5 Modelling circuit components

There are two main approaches in modelling digital hardware in a process algebra or in any other event-based specification formalism. The voltage level transitions in a digital system may be modelled by focussing either on the actual value of the voltage level or just on the changes of the level [22].

In the first approach, called *level-based* modelling, the value of the voltage level is modelled by an event. We use a Boolean port a to model a high voltage level, $a^{\sqcap}$, and a low voltage level, $a_{\sqcup}$. Such ports have Boolean values and are denoted by a solid diamond, ♦.

For example, a NOT gate with input a and output x is given in the level-based modelling technique by the $NOTlb$ process generation function defined in Fig. 3a. Such a model is an immediate translation of the truth table that gives the values of the output for every combination of the inputs. The level of the output depends directly on the levels of the inputs; therefore, we can model the NOT gate using a process consisting of just one state as the model is only ever in a single state. The levels of the inputs and output are initialised by the environment with which the gate interacts.

In the alternative approach, called *transition-based* modelling, the change of the voltage level is modelled by an event. We use a Boolean port a to model a change from low to high voltage (event $a+$) and a change from high to low voltage level (event $a-$). Such ports have Boolean values and are denoted by a solid triangle. The value can be low, that is, the port is ready to generate a rising signal $(+)$, or high, that is, the port is ready to generate a falling signal $(-)$. The changes of the level of the output depend not only on the changes of the levels of the inputs but also on the current levels of those inputs that possibly do not change. Therefore, the levels of the inputs must be encoded within a state. Consequently, Boolean ports must be initialised with values consistent with the initial state. A port that is initially high is denoted by ▲; a port that is initially low is denoted by ▼. The use of ▲ or ▼ provides a structural notation to constrain the behaviour and

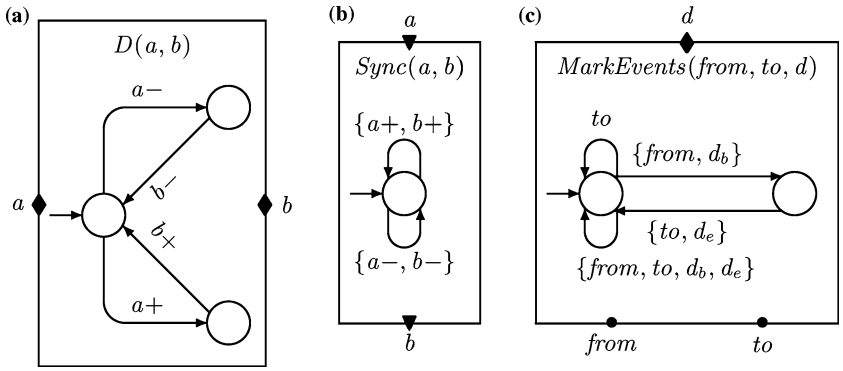

Fig. 4 **a** Process generation function that models a delay; **b** *s* Process generation function that models simultaneous signals; **c** time-interval refinement $markevents(from, to, d)$

improves readability. For example, a NOT gate with input a, initially low, and output x is given in the transition-based modelling technique by the $NOTl$ process-generation function defined in Fig. 3b, whereas a NOT gate with input a, initially high, and output x is given by the $NOTh$ process-generation function defined in Fig. 3c.

If we replace ▼ with ▲ in port a in Fig. 3b, we introduce a constraint that is not consistent with the given behaviour. In such a situation, the process cannot evolve and the resultant behaviour is a deadlock. If, instead, we replace ▼ with ♦ in Fig. 3b, we do not modify the behaviour of $NOTl(a, x)$ because the constraint to be receptive to only rising edges, which is modelled by ▼, is already implicit in the model of $NOTl(a, x)$. Therefore, when ▲ and ▼ are consistent with the behaviour, they can always be replaced by ♦, without affecting the semantics.

3 Modelling time

Time can be incorporated in Circal by adding an event representing the passage of time. This event can be introduced as a global tick event [22] or as a local tick event [6]. In such a model, circuit delays are represented as processes of which the duration is related to a certain number of ticks [2, 22]. However, there are problems with this approach when modelling delays in asynchronous circuits. First, the length of each delay must be prespecified. Second, the unit of resolution of delays is related to the number of tick events and thus modelling at a high timing resolution is associated with a further state explosion. Third, the properties verified for a specific example are limited to the resolution of the actual delay models used.

In the design of bounded delay asynchronous circuits, the actual delays in the circuits are less important than the relationships between delays. In a micropipeline, the major consideration for correct operation of the circuits is to ensure that the delay in the control path is not shorter than the delay in the data path. Thus, it is natural to express delays as inequalities. In Sect. 3.1, we present a delay model for introducing time in asynchronous hardware [8, 9]. The constraint-based method introduced in Sect. 2.3 avoids the state explosion associated with tick event-based delays and uses the natural expression of the relative length of delays. In Sect. 3.2, we present how to model *time interval refinements*. In Sect. 3.3, we show how timing constraints are specified in such a model. Other timing models suitable for specifying more complex timing constraints on an explicit representation of time have been presented in a previous paper [9].

3.1 The delay model

We consider a model of asynchronous circuits where there is no delay inside gates, but all delays are represented using special delay components. Such a component has an input port on the left and an output port on the right and may be modelled by the behaviour in Fig. 4a.

The $D(a, b)$ process-generation function defines an arbitrary delay between the in and out signals. In fact, an arbitrary number of external actions may occur between input $a+$ and output $b+$ or between input $a-$ and output $b-$.

This approach simplifies gate specification by separating the description of the functional behaviour from timing issues. A gate with delay can therefore be modelled by an undelayed gate whose output is connected to a delay component as shown in Fig. 5a.

Notice that the output port of the gate and the input port of the delay do not appear in the graphical representation. Such a representation visually associates the delay with the gate.

Moreover, a delay component can also model a delayed wire connecting the output of a gate to the input of another gate as in Fig. 5b. This is correct under the assumption that the maximum propagation time through the wire is not greater than the minimum time between two successive transitions at any pin in the circuit. That is, only one signal can travel at any time on the wire. In our delay model, a delayed wire is undistinguishable from a delayed gate: it is just a matter of interpretation. The absence of ports on the graphical representation of a delayed wire highlights such an interpretation.

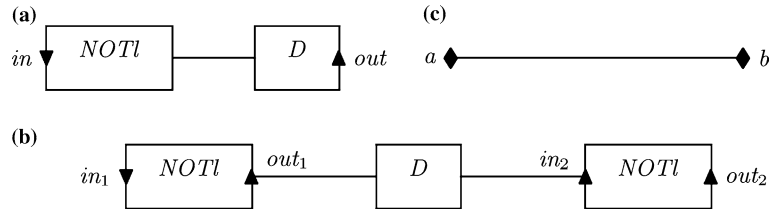

Fig. 5 **a** Delayed gate; **b** Delayed wire; **c** Undelayed wire

To model delayless wires, we use the $Sync(a, b)$ process-generation function whose behaviour is modelled in Fig. 4(b). Here, an input $a+$ is always simultaneous with an output $b+$ and an input $a-$ is always simultaneous with an output $b-$. An undelayed wire is represented as a line connecting two ports having different names, as shown in Fig. 5(c).

3.2 Time-interval refinements

In previous work [22], delays are entirely introduced at the circuit level and become part of the circuit implementation. To simplify description, we choose to introduce just arbitrary delays at the circuit level and define time relationships between them at a higher, more abstract descriptive level [8, 9]. Arbitrary delays, specified by instantiating the D process generation function defined in Fig. 4a, are marked by pairs of abstract events. Such abstract events are associated with the physical events occurring in the circuit through the composition of the process that specifies the circuit with refinement processes. The timing constraints are then specified at a higher level and their constraining power is propagated through the composition to the circuit level as described in the next section. Abstract events are introduced by refinement processes. A pure refinement process should take into account all the possible sequential and simultaneous occurrences of the physical events to be marked. For example, process-generation function $MarkEvent(from, to, d)$ defined in Fig. 4c marks the beginning of the time interval between an occurrence of physical action $from$ and an occurrence of physical action to, with abstract action d_b, and the end of the interval, with abstract action d_e. We have used the usual ♦ symbol, labelled by d, to denote the two possible values d_b and d_e.

All refinements defined by process generation function $MarkEvents$ do not introduce any constraints in the physical system. However, refinement processes are often associated with assumptions, which are constraints enforced by the environment. If this is the case, the refinement process does not model those sequential and simultaneous combinations of physical events that cannot occur under the given assumptions. Therefore, a single process can model both a refinement and an assumption, as anticipated in Sect. 2.4.

Process-generation function $MI_1(from, to, mark)$ described in Fig. 6a marks with the new $mark$ abstract action the time interval between physical actions $from$ and to. Action $mark$ models a generic point in the time interval between $from$ and to. For this reason, it is called *time-interval refinement*. Notice that MI_1 works also as an assumption because it implicitly forces $from$ and to always to occur in sequence.

Process generation function MI_1 marks the intervals generated by each sequence consisting of occurrences of $from$ and to. However, it might be useful to mark only some occurrences of the same physical action. For example, process-generation function MI_2 in Fig. 6b marks intervals between $from$ and to starting from the second occurrence of $from$ and to.

3.3 Timing constraints

We have said that, in asynchronous hardware, it is natural to express the relationships between the delays as inequalities. The delays that are compared are associated with different paths; for example, the control path and the data path in a micropipeline. The assumption that the delay in the control path is not shorter than the delay in the data path is modelled through the process-generation function shown in Fig. 7a, where we have represented arrows connecting the same states but labelled with different sets of actions by drawing just one link with all the labels vertically stacked. This process-generation function is based on the assumption that the start of any occurrence of the shorter delay (s_b) is always simultaneous with the start of an occurrence of the longer delay (l_b). Under such an assumption, the N initial state changes to the D state only when s_b occurs simultaneously with any of l_b (start of an occurrence of the longer delay) or l_b and l_e (start of an occurrence of the longer delay simultaneously with the end of its previous occurrence). In the D state, the process waits for s_e (end of the current occurrence of the shorter delay).

Using the time-interval refinements presented in Sect. 3.2, timing constraints can be modelled in terms of processes performing only abstract events. Timing constraints establish a time relationship among the abstract events, which is then propagated by the time-interval refinements down to the physical events and reduces the size of the system model. Figure 7b shows a physical system S, where the time occurring between events s_1 and s_2 is constrained to be not less than the time occurring between events s_3 and s_4. Time-interval refinement $MarkEvents(s_1, s_2, p)$ marks the occurrence of s_1 with p_b and the occurrence of s_2 with p_e; time-interval refinement $MarkEvents(s_3, s_4, q)$ marks the occurrence of s_3 with q_b and the occurrence of s_4 with q_e. Then timing constraint *NotLessThan(p,q)* constrains the

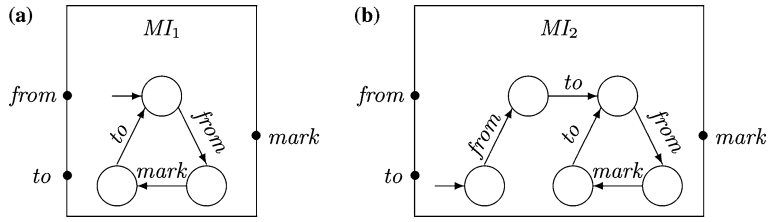

Fig. 6 **a** Interval refinement process MI_1; **b** Interval refinement process MI_2

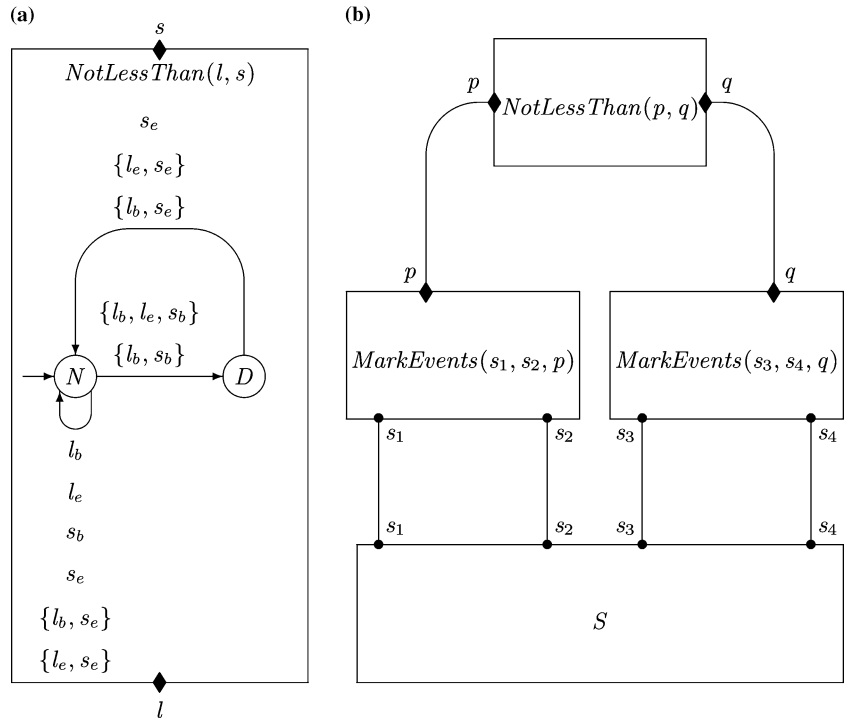

Fig. 7 **a** Timing constraint $NotLessThan(p, q)$, **b** High-level modelling of timing constraints

duration modelled by abstract event p to be not less than the duration modelled by abstract event q.

4 Asynchronous micropipelines

To demonstrate our new timing verification technique, we utilise an asynchronous micropipeline circuit whose design is related to the AMULET2 asynchronous RISC processor [16], which has known performance advantages over existing embedded processors.

In a RISC processor, the instruction pipeline is composed of logic stages and latches. Progress through a synchronous pipeline is managed by the clock; once the logic has completed evaluation, all the latches are clocked at the same time, simultaneously moving all instructions to the next pipeline stage. In an asynchronous micropipelined processor, the evaluation of a pipeline stage is governed by local interactions with its neighbours using a request acknowledge handshaking protocol [15,16]. It is possible for one stage to be evaluating while, at the same time, a stage further on is transferring an instruction to its neighbour. Thus, whilst the performance of a synchronous pipeline is governed entirely by the clock rate that can be achieved with a particular logic design, the performance of an asynchronous pipeline depends instead on the design of the handshaking control for each stage. In particular, if the asynchronous logic pipeline is to be as fast as the synchronous one it must be possible for all logic stage evaluations to overlap, as in the synchronous case.

Here, we present a detailed analysis of the correctness, timing and performance of two distinct micropipelines whose design is influenced by that of the AMULET2 asynchronous RISC processor [15]. In both designs, the control logic consists of a four-phase handshaking protocol with active rising signals. We simplify the datapath by treating it as a sequence of latches without interleaved logic stages, such as found in *real* asynchronous micropipelines. This simplification, where the micropipeline reduces to a FIFO queue, helps in the presentation of our modelling and verification

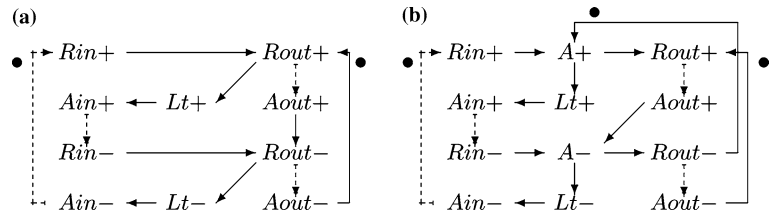

Fig. 8 **a** STG of simple four-phase latch control circuit; **b** STG of semidecoupled four-phase latch control circuit

methodology. An earlier presentation of this approach was given in previous papers [7,11].

4.1 Specification

The specifications of the single stages of the two latch control circuits we are going to model are given by the signal transition graphs (STGs) [12] in Fig. 8, where:

- the dashed arrows denote the orderings that must be maintained by the environment (*assumptions* about the environment);
- the solid arrows denote the orderings that must be ensured by the circuit itself (*properties* of the circuit);
- a solid circle is attached to an arrow in order to denote that the target of such an arrow is initially enabled to occur.

Rin and $Rout$ define the input and output *request signals*. Ain and $Aout$ define the input and output *acknowledgement signals*. The Lt *latch control signal* causes the data latch to be open when low ($Lt-$) and closed when high ($Lt+$). The STGs in Fig. 8 show that, when input data are available ($Rin+$), the latch may close ($Lt+$) and then the input may be acknowledged ($Ain+$); when the output data have been acknowledged ($Aout+$), the latch may open again ($Lt-$). It is the responsibility of the environment to ensure that

- an input acknowledgement signal ($Ain+$) will eventually reset the input request ($Rin-$);
- the reset of the input acknowledgement ($Ain-$) will be eventually followed by a new input request signal ($Rin+$);
- an output request signal ($Rout+$) will be eventually followed by an output acknowledgement ($Aout+$);
- the reset of the output request ($Rout-$) will be eventually followed by the reset of the output acknowledgement ($Aout-$).

These assumptions about the environment are denoted by dashed arrows in Fig. 8. Notice that the assumptions must be the same for both STGs.

4.2 Implementation

The STGs in Fig. 8 can be implemented as circuits using informal or semiformal synthesis techniques. A possible implementation of the STG in Fig. 8a is given in Fig. 9a and a possible implementation of the STG in Fig. 8(b) is given in Fig. 10a [15]. In the implementation in Fig. 9a, we have used a conventional Muller C-gate; in the implementation in Fig. 9a, we have used two asymmetric Muller C-gates. The methodology for implementing Muller C-gate in Circal has been presented in previous work [10].

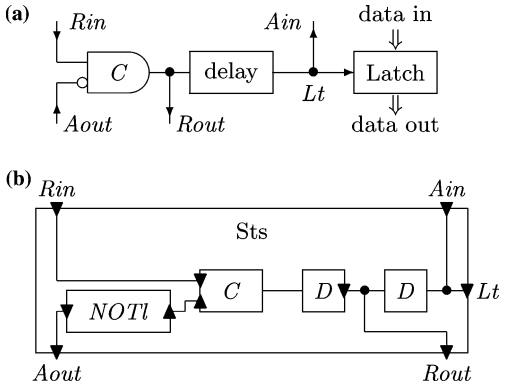

Fig. 9 **a** Simple four-phase latch control circuit; **b** Process algebra model of the simple four-phase latch control circuit

Once the gates have been defined as the basic components, the whole circuit is specified by composing in parallel the processes that define the gates, with additional processes to define the delays in the gates or on the wires. For example, the circuit in Fig. 9a is represented in the process algebra by the Sts process defined in Fig. 9b and the circuit in Fig. 10a is represented by the Std process defined in Fig. 10b.

5 Verification

5.1 Property verification

A correctness concept that can be readily characterised in Circal is the behavioural equivalence $P \cong Q$ between two processes, which is implemented by the Circal System [2].

However, in performing formal verification, equivalence is often too strong a property requiring proof. For certain systems, verifying their correctness consists of determining that certain properties hold, where these properties do not constitute a complete specification. Concurrent systems frequently require us to determine that certain time-dependent properties, known as temporal properties, hold. Here we consider a subclass of temporal properties called *safety properties*, which assert that "anything bad will never happen".

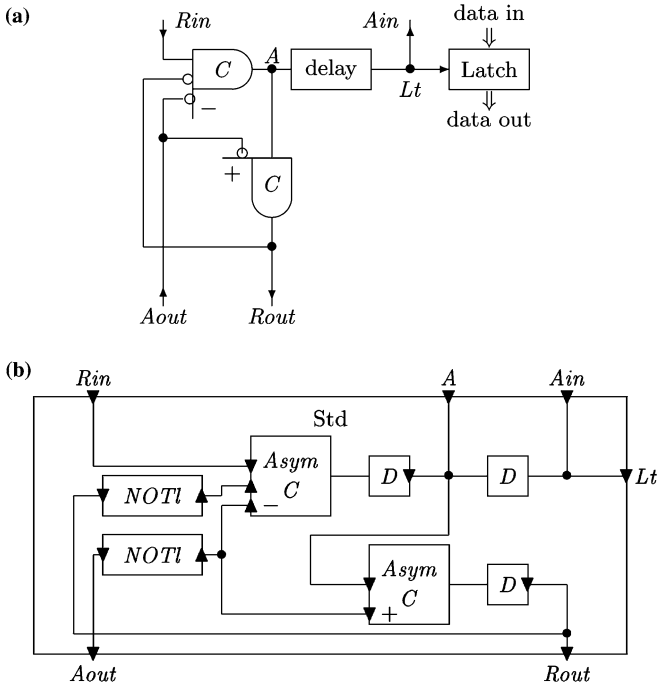

Fig. 10 **a** Semidecoupled four-phase latch control circuit; **b** Process algebra model of the semidecoupled four-phase latch control circuit

The constraint-based modelling technique presented in Sect. 2.3 supports a clear characterisation of safety properties. Let S be a process of sort L and P be a process of sort $L' \subseteq L$. Let us suppose that S models a physical system, then P can model a safety property that might or might not hold in S. If P constrains S, then the property represented by P is not implicitly modelled in S; on the other hand, if P does not constrain S, then the property represented by P is implicitly modelled in S, that is, the system satisfies the property. Therefore, the verification methodology consists of checking whether or not the process P that represents the safety property to be verified constrains the process S that represents the system.

The key point of the methodology is how to check whether or not one process constrains another. This can be done by an appropriate combination of parallel composition and the equivalence-checking procedure, which is available in the Circal System. In order to constrain a process S with another process P, we just need to compose S and P in parallel. Thus, if the constraint expressed by P is already implicitly modelled in S, then the parallel composition of S and P must be equivalent to S itself. Using the Circal syntax introduced in Sect. 2.3, we denote the parallel composition by $*$ and the equivalence checking by $\cong$. Then, in order to verify that the safety property modelled by process P holds on the system modelled by process S, we need to check the following equivalence:

$$S * P \cong S \tag{1}$$

When the safety property only holds under the assumptions defined by a process A, equivalence (1) becomes as follows:

$$A * S * P \cong A * S \tag{2}$$

In this case, the property is verified for the constrained system $A * S$. In (2), A can be any assumption, including a timing constraint, and P any property, including a performance property. In this way, the verification schema given by (2) integrates correctness, timing and performance verification [7]. Therefore, properties are expressed in the same formalism as the model, an approach also adopted by formalisms based on propositional, first-order logic and higher order logic [17], but which is not usually achievable in most process algebra-based verification environments.

5.2 Correctness verification

The two STGs defined in Fig. 8 can be modelled in our process algebra framework by defining a process for every single relationship between pairs of signals connected by an arrow and then by composing all these processes together. An arrow between signals pre and $post$ without a solid circle attached is defined by the AC process-generation function represented in Fig. 11a. Signal $post$ is not initially enabled. Therefore, it must always occur either simultaneously or after an occurrence of signal pre. An arrow between signals pre and $post$ with the solid circle attached is defined by the IC process-generation function represented in Fig. 11b. Signal $post$ is initially enabled. Therefore, after the first occurrence of $post$, every further occurrence of $post$ must always occur either simultaneously or after an occurrence of signal pre. The dashed arrows in Fig. 8 are represented in terms of processes in the same way as the solid arrows. However, these processes play different roles in the specification: processes that correspond to dashed arrows are *assumptions*, whereas processes that correspond to solid arrows are *properties*. The properties expressed by the STGs in Fig. 8 are *safety* properties; they assert that, for all possible executions, the defined ordering of signals must not be violated. Such properties can be verified using the verification schema given by (2) in Sect. 5.1.

Let us apply the schema given by (2) to the correctness verification of the circuit given in Fig. 9a, which is modelled in Circal by the Sts process defined in Fig. 9b. We have to define processes S, A and P in terms of the circuit model in Fig. 9b and of the instantiations of the AC and IC process generation functions that model the arrows of the STG in Fig. 8a.

Process S is given by the Sts circuit model in Fig. 9b. Process A is given by composing in parallel the instantiations of AC and IC that represent the assumptions as shown in Fig. 12a.

Here, we use a $+$ to denote a port that is receptive only to rising edges and a $-$ to denote a port that is receptive only to falling edges. If in (2) we replace S by the Sts process, we can automatically verify using the Circal System that the equivalence is true, that is the single stage modelled by Sts meets

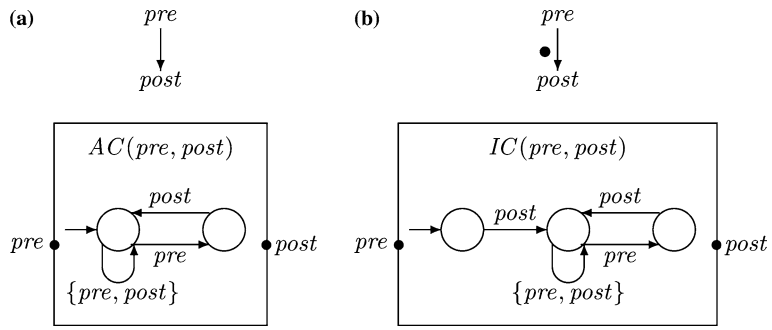

Fig. 11 **a** Process algebra model of an STG *arrow* without *solid circle*; **b** Process algebra model of an STG *arrow* with a *solid circle* attached

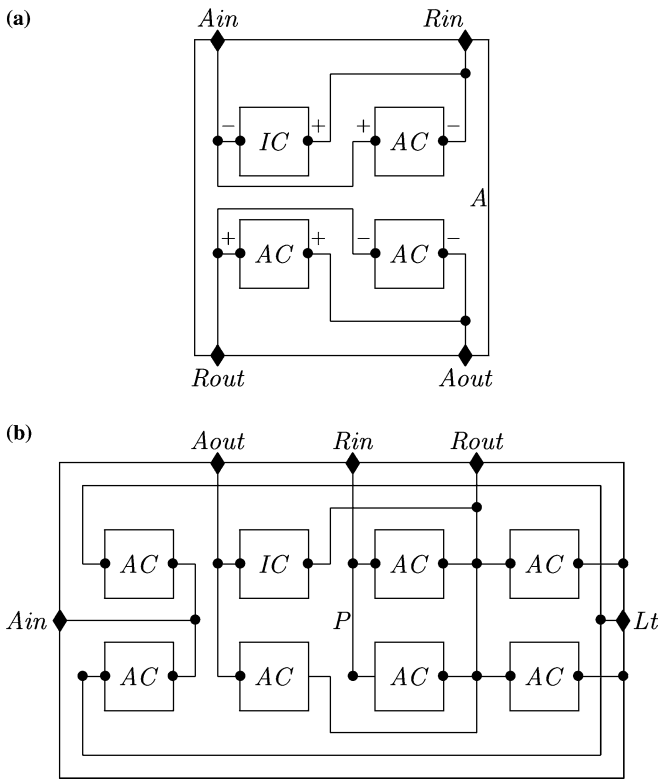

Fig. 12 **a** Process *A* modelling the assumptions; **b** Process *P* modelling the properties of *Sts*

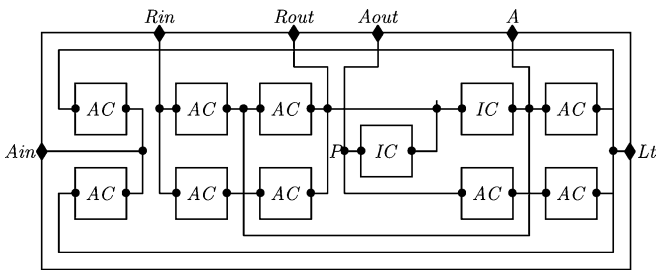

Fig. 13 Process *P* modelling the properties of *Std*

the properties modelled by P under the assumptions modelled by A. Therefore, the implementation given in Fig. 9a is correct with respect to the specification given in Fig. 8a.

In order to verify the correctness of the circuit given in Fig. 10a, we have to define S as the circuit given in Fig. 10b, and A and P from the STG in Fig. 8b. We notice that A is the same as before and P is shown in Fig. 13.

Analogously, we can use the Circal System to automatically verify that the implementation given in Fig. 10(b) is correct with respect to the specification given in Fig. 8(b).

5.3 Performance analysis

In the previous section, we have seen how to automatically verify that the circuits defined in Fig. 9a and in Fig. 10a operate correctly. Whilst they are both correct with respect to the corresponding STG specifications, they show different performance characteristics.

Several stages of our asynchronous micropipeline controller may be connected in series, as shown in Fig. 14c.

Here, St_i, $i = 1, 2, 3$ is the ith control stage, which can be instantiated either by Sts (given in Fig. 9b) or by Std–A_i (Std given in Fig. 10b with A_i hidden).

The latches corresponding to the controller form a FIFO of registers. The maximum potential parallelism for such a FIFO occurs when all the latches are full at the same time. Whether or not such a potential parallelism is effectively attained depends on the handshake control protocol.

We define *throughput* as the number of data items that can be passed through the pipeline per complete *handshake cycle*. This definition is justified by the practical observation that asynchronous pipeline throughput is limited by the elapsed time for one handshake cycle in the control stages. A handshake cycle for the ith stage is the sequence of events from Rin_i+ to Ain_i-. Previous studies have shown that there is a direct relationship between this throughput and the number of pipeline stages that can be full at a particular time [31].

We notice that, in the STG in Fig. 8a, $Aout_i$ must be low (and therefore the next latch empty, as in the system seen in Fig. 14c) before Lt_i can go high (and this latch becomes full). This is not the case for the STG in Fig. 8(b), where

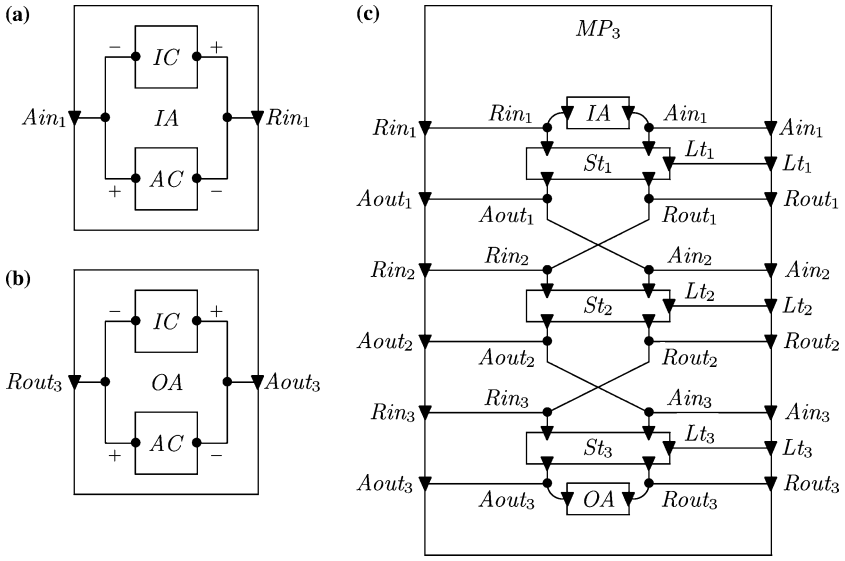

Fig. 14 **a** Process IA modelling the assumptions on the input; **b** Process OA modelling the assumptions on the output; **c** A 3-stage micropipeline controller MP_3

the input and output side of the latch control stage are partly decoupled. A consequence of this decoupling is that a falling signal $Aout_i-$ that acknowledges the falling signal $Rout_i-$ in a given handshake cycle is concurrent with a rising signal Lt_i+ in the next handshake cycle. In this section, we analyse the implication of this decoupling on the performance of the micropipeline.

The stages of the micropipeline are connected together, as in Fig. 14c. The IA process, defined in Fig. 14a, models the assumptions on the input; the OA process, defined in Fig. 14b, models the assumptions on the output. In a previous paper [7], we have shown how to automatically verify that the input assumptions on the first stage and the output assumptions on the last stage imply input and output assumptions on the intermediate stages.

We can carry out the automated analysis of the performance of the micropipeline using the same methodology that we have used in the previous section for the correctness proof [7]. Now, the safety property to be used will catch some performance aspects of the system rather than just the ordering of event occurrences. We want to express the performance in terms of throughput. We then need a way to characterise when a stage of the micropipeline is full. The ith stage starts to be full when Lt_i goes high. At this point, the data in the corresponding buffer is latched. It cannot be overwritten by data coming from the $(i-1)$th stage and is propagated on to the $(i+1)$th stage. When this data is latched in the $(i+1)$th stage, Ain_{i+1} goes high and, as a result, $Aout_i$ goes high. At this point, because the current data is latched in the $(i+1)$th stage, the buffer of the ith stage can be overwritten. Therefore, the time interval where the ith stage is full starts at Lt_i+ and ends at $Aout_i+$.

We instantiate the MI_1 process-generation function given in Fig. 6a to mark with the new $full_i$ abstract action the time interval between action Lt_i+ and action $Aout_i+$. When

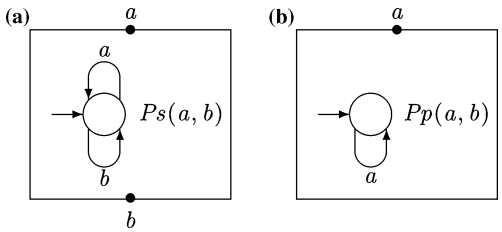

Fig. 15 **a** Property process generation function Ps; **b** Property process generation function Pp

instantiations of MI_1 are composed with a stage of the micropipeline, as shown in Fig. 16, we obtain a refinement of the micropipeline where some time intervals are highlighted by the abstract actions.

As we said in Sect. 3.2, $MI_1(Lt_i+, full_i, Aout_i+)$ works not only as a refinement but also as an assumption. In fact, it implicitly forces Lt_i+ and $Aout_i+$ always to occur in sequence. This assumption is acceptable if the delay inserted between $Rout_i$ (in the simple circuit) or A_i (in the semidecoupled circuit) and Lt_i is the same for every stage. If this is the case, in the simple circuit, the time interval between $Rout_i+$ and $Aout_i+$ is greater than the time interval between $Rout_i+$ and Lt_i+. In fact, the former consists of the internal delay of the C-gate in the $(i+1)$th stage plus the delay inserted between $Rout_{i+1}$ and Lt_{i+1}, whereas the latter consists of just the delay inserted between $Rout_i$ and Lt_i. Analogously, in the semidecoupled circuit, the time interval between A_i+ and $Aout_i+$ is greater than the time interval between A_i+ and Lt_i+.

After refining the three-stage controller as shown in Fig. 16, the resultant $TR_{1,2}$ process may be readily analysed to catch performance properties. The $Ps(a, b)$ process-generation function given in Fig. 15a models the property

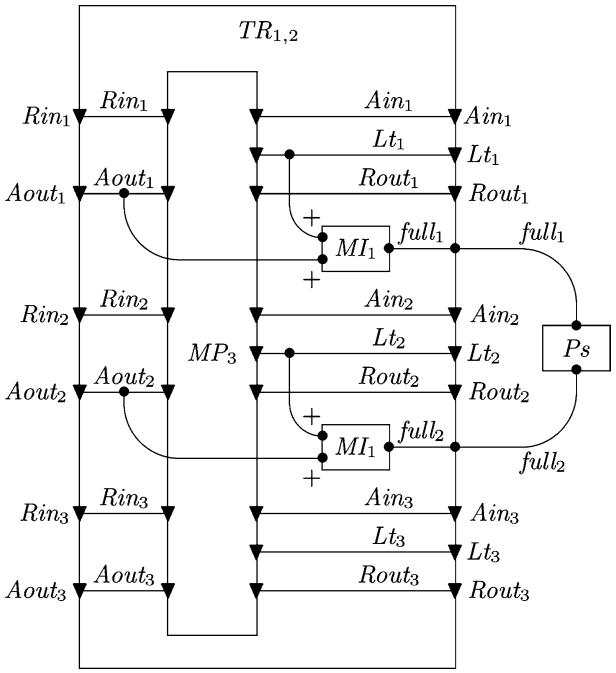

Fig. 16 Analysis of property $Ps(full_1, full_2)$

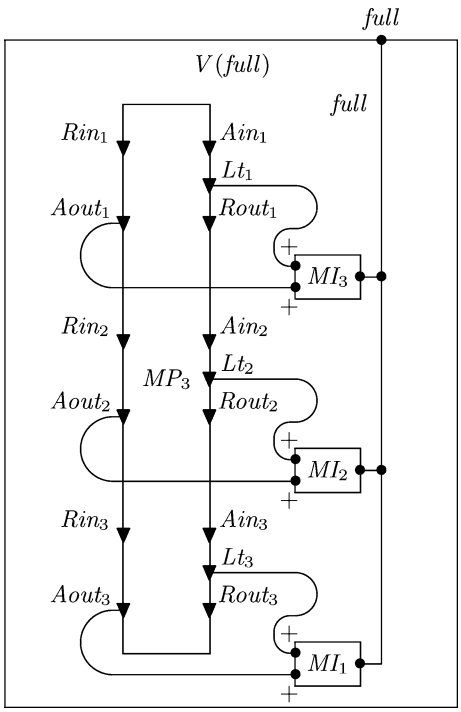

Fig. 17 Abstract view $V(full)$ of a 3-stage controller for the analysis of property $Pp(full)$

that actions a and b cannot synchronise. This means that a and b can never occur simultaneously. We can compose $Ps(full_1, full_2)$ with the $TR_{1,2}$ process, as in Fig. 16, and check the equivalence

$$TR_{1,2} * Ps(full_1, full_2) \cong TR_{1,2} \tag{3}$$

This is an instantiation of equivalence (1) in Sect. 5.2. If equivalence (3) holds, then $full_1$ and $full_2$ never occur simultaneously in $TR_{1,2}$. This means that the time interval between Lt_1+ and $Aout_1+$ and that between Lt_2+ and $Aout_2+$ are disjoint. Therefore, it means that the first two stages can never be full at the same time.

With the Circal System, we automatically verify that using the simple control circuit as the basis of an asynchronous design, at best, alternate stages can be occupied at the same time. In fact, the Ps property, which relates to concurrent activity amongst *multiple* stages, holds for pairs of adjacent stages, but not for pairs of alternate stages. Therefore, the degree of parallelism achieved is only 50% of the potential parallelism. This is equivalent to a throughput not greater than 50% [31].

If we check property Ps on the micropipeline in Fig. 10, the property does not hold for any pair of stages. This proves that adjacent stages may be occupied at the same time. So no upper bound of 50% is given to the throughput. However, we would like to know how many stages can be occupied at the same time. To achieve this, we need to use the view $V(full)$ in Fig. 17, in which the i-th stage is refined using the MI_{n-i} process, with n indicating the number of stages in the micropipeline. The generic MI_i process starts marking the time intervals from the i-th occurrence. MI_1 is defined in Fig. 6a and MI_2 is defined in Fig. 6b. MI_3 may be defined analogously. In this way we start marking intervals only when the flow of data reaches the output of the micropipeline, that is, when all stages may be occupied.

We want to verify whether this potential for full parallelism among all stages can be effectively achieved. Because $full$ belongs to the sort of every instantiation of MI_i, $i = 1, 2, 3$, in Fig. 17, then, following the composition rules introduced in Sect. 2.2, all stages are full at the same time iff the $full$ action visibly occurs in the behaviour of $V(full)$. In particular, if there is an execution such that all stages are full simultaneously in every handshake cycle, then the $V(full)$ view is equivalent to the $Pp(full)$ process defined in Fig. 15d, which models an infinite sequence of $full$ actions. Using the Circal System, we can prove that

$$V(full) \cong Pp(full) \tag{4}$$

and verify that a micropipeline of semidecoupled latch control circuits has a possible execution such that all stages are full simultaneously in every handshake cycle. Therefore, the semidecoupled latch control circuit shows a possible throughput of 100%. Whether such a performace is effectively attained depends on the environment, the surrounding circuitry, in which the micropipeline operates. In a simple FIFO, the maximum performance may be reached, but if the micropipeline incorporates processing logic, there may be a performance degradation [15].

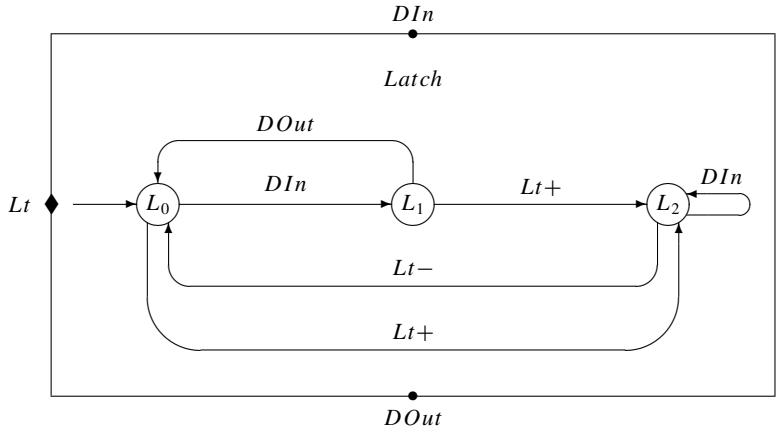

Fig. 18 Model of a latch

5.4 Timing analysis

In this section, we consider a full micropipeline stage consisting of the simple control circuit and the latch component.

We model the flow of data through the stage by an input action DIn and an output action $DOut$. We interpret an occurrence of the DIn action as a change of the input of the latch and an occurrence of the $DOut$ action as a change of the output of the latch. Process $Latch$ is modelled as in Fig. 18.

The latch is initially open (Lt is low) in state L_0 and waiting either for input on the DIn port, which takes to state L_1, or for action $Lt+$, which closes the latch and takes to state L_2. In L_1, either the input is propagated to the output, which results in action $DOut$, which takes it back to the initial state L_0, or the latch is closed through action $Lt+$, which takes it to state L_2. We assume that the propagation time of the data through the latch is less than the time between two consecutive changes of the input data. Thus, no action DIn can occur in L_1. In state L_2, the latch is closed and no input can be propagated to the output. Therefore, all input changes DIn occurring in L_2 cannot be stored in the buffer element and will not be followed by an output change $DOut$.

If, after the occurrence of an input change, we have reached L_2 directly from L_0, then the input change has been propagated to the output before the latch closes. This is modelled by the sequence $DIn\ DOut\ Lt+$. Notice that the latch can also close an arbitrary number of times even in the absence of input changes. This is modelled by a sequence $Lt-\ Lt+$.

If we have reached L_2 from L_1, then the latch has been closed before the input change has been propagated to the output. This is modelled by the sequence $DIn\ Lt+$, which cannot be followed by $DOut$. Therefore, in L_2, only two actions can occur: DIn, which can occur an arbitrary number of times so modelling further input changes, which are not stored and will never result in corresponding output changes, and $Lt-$, which opens the latches and takes it back to the initial state L_0.

We consider the version of control circuit given by the Stn process, shown in Fig. 19, where there is only a delay on the C-element, and such a delay has an identical length for every stage. In this way, the delay that matches the evaluation time disappears and, consequently, we have to assume a null evaluation delay in the data flow, i.e. no processing logic between stages.

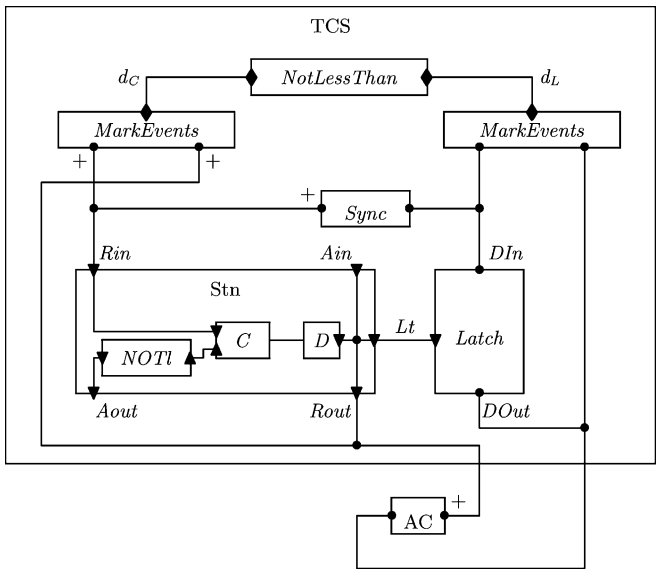

Fig. 19 Process algebra model of the simple four-phase latch control circuit with matching delay

We want to verify that the C-gate delay must be no shorter than the latch data-in to data-out delay for the correct operation of the circuit [15]. As the correct operation of the circuit, we intend now that the output request $Rout+$ is sent to the next stage only if the data output $DOut$ has already been propagated to the next stage. Such a property is modelled as process $AC(DOut, Rout+)$, instantiation of the process-generation function defined in Fig. 11a.

The verification is performed by using the technique presented in Sect. 3.3. As shown in Fig. 19, the Stn process is composed with the $Latch$ process to obtain a full micropipeline stage. The C-gate delay is defined in terms of abstract event d_C using the $MarkEvents(Rin+, Rout+, d_C)$ time-interval refinement, while the latch data-in to data-out delay is

defined in terms of abstract event d_L using the $MarkEvents(DIn, DOut, d_L)$ time-interval refinement. The two delays are then correlated using process *NotLessThan* (d_C, d_L), which constrains the system to satisfy the condition that the C-gate delay is no shorter than the latch data-in to data-out delay. The composite process TCS, which also includes process $Sync$ to enforce the simultaneity of $Rin+$ and DIn, meets the property modelled by process $AC(DOut, Rout+)$, as verified by checking the equivalence

$$TCS \ * \ AC(DOut, Rout+) \cong TCS,$$

where

$$\begin{aligned} TCS \leftarrow\ & Sts \ * \ Latch \ * \ Sync(Rin+, DIn) \ * \\ & MarkEvents(Rin+, Rout+, d_C) \ * \\ & MarkEvents(DIn, DOut, d_L) \ * \\ & NotLessThan(d_C, d_L). \end{aligned}$$

If we remove the $NotLessThan(d_C, d_L)$ timing constraint from process TCS, then the equivalence does not hold any more. This proves that the timing constraint is a sufficient condition for the correct propagation of the data flow through a single stage.

6 Discussion

In this paper, we have presented in detail a process algebra-based approach for the integrated verification of correctness, performance and timing in concurrent systems. The verification procedure is entirely performed within the Circal process algebra, without any recourse to other formalisms, such as modal or temporal logic, stochastic models and dense time models. Both correctness and performance properties as well as timing assumptions are captured in the same verification framework and are proven automatically. The automatic checking mechanism is based on both the underlying semantics [25] and a notion of testing equivalence [2]. If the automatic equivalence check fails to match, the Circal System [29] gives a diagnostic in terms of one event trace up to the point where the mismatch has been found [22]. The capability of expressing both model and properties within the same formalism is common in axiomatic frameworks [17] but not in process algebra-based frameworks. Some process algebras do not support this approach due to their use of a binary composition operator [23]. We exploit the multiway feature of Circal composition in a manner analogous to logical conjunction. Our approach is related to Roscoe's approach using CSP [27].

The approach has been applied to two four-phase asynchronous micropipelines. Both micropipelines have been proven to be correct using the automatic verification facility in the Circal System, but here we are able to demonstrate that they have different performace properties. The performance of the circuit in Fig. 9a shows a throughput not greater than 50%. The performance of the semidecoupled circuit in Fig. 9c shows a possible throughput of 100%, but such a performace is effectively attained only in the appropriate operational context; that is, it depends on the other circuit components connected to it in its immediate, surrounding environment.

It is interesting to notice that the two performance properties that we have analysed in Sect. 5 can be seen as two temporal properties that belong to two distinct classes of properties requiring proof. For instance, the property that "adjacent stages can never be occupied at the same time" belongs to the class of safety properties that assert that "for any possible execution, something is true at any time". This class is represented in branching time temporal logic by instantiations of the formula $\forall G\alpha$. In our methodology, this class of properties is verified by checking whether or not the process that models the property constrains the system.

It is interesting to compare synchronous versus asynchronous logic with respect to the verifiability. From the point of view of correctness, the verification of the design of a synchronous pipeline has been taken by most published work as meaning that the logic stages perform the correct functions and the control state machine is correct. The behaviour of the clock is typically abstracted away by an assumption that the clock rate will be set in such a way as to ensure that the slowest stage has time to complete in both the data path and the state-machine implementation. As a consequence of this abstraction, the so-called set up and hold times of latches are also ignored in the formal verification even though meeting these constraints is required for correct operation. Thus, the verification of synchronous circuits without consideration of the clock timing issue is incomplete. In an asynchronous setting, because every transition on every gate in the control path could be significant, verification must be much more complete to be useful at all.

In a micropipeline, there is also a requirement that the control circuit has a greater delay than the logic in the data path. This requirement allows the circuit designer to avoid the dual rail area overhead in the data path [15, 16]. It also means that the data path of the micropipeline can be identical with that of a synchronous design. Thus, the verification of correct operation of a micropipeline reduces to verification of the correct operation of the data path (using similar methods as are currently used for synchronous circuits), verification of the handshaking control path and verification of the timing relationship between data path and control path.

A final remark is that our performance analysis is based on a qualitative notion of performance, that is, the degree of parallelism of the system components. Such an abstract notion does not provide a complete performance evaluation of the system under analysis. We have seen that the increase of parallelism is associated with an increase in the size and complexity of the handshake control circuit. This might lead to a longer delay in the control circuit and, consequently, to a lower system performance.

Acknowledgements We would like to thank Steve Furber for detailed answers to our questions and Graeme Smith for helpful comments on this work. This work has been supported in part by the Australian Research Council, in part by a University of Queensland grant, in part by UNU-IIST and in part by Sun Microsystems Laboratories, USA.

References

1. Bacelli F et al. (1992) Synchronisation and linearity—algebra for discrete event systems. Wiley, New York
2. Bailey A, McCaskill GA, Milne GJ (1994) An exercise in the automatic verification of asynchronous designs. Formal Methods Syst Des 4(3):213–242
3. Birtwistle G, Davis A (eds) (1995) Asynchronous digital circuit design. Springer, Berlin Heidelberg New York
4. Bolognesi T, Brinksma E (1987) Introduction to the ISO specification language LOTOS. *Comput Netw ISDN Syst* 14(1):25–59
5. Cerone A, Cowie AJ, Milne GJ, Moseley PA (1996) Description and verification of a time-sensitive protocol. Technical report CIS-96-009, University of South Australia, School of Computer and Information Science, Adelaide, Australia
6. Cerone A, Cowie AJ, Milne GJ, Moseley PA (1997) Modelling a time-dependent protocol using the Circal Process Algebra. Lecture Notes in Computer Science, Vol 1201. Springer, Berlin Heidelberg New York, pp 124–138
7. Cerone A, Kearney DA, Milne GJ (1998) Integrating the verification of timing, performance and correctness properties of concurrent systems. In: Proceedings of the international conference on application of concurrency to system design (CSD'98), IEEE Comp Soc Press, pp 109–119
8. Cerone A, Kearney DA, Milne GJ (1997) Verifying bounded delay asynchronous circuits using time relationship Constraints. Technical Report CIS-97-012, University of South Australia, School of Computer and Information Science, Adelaide, Australia
9. Cerone A, Milne GJ (1997) Specification of timing constraints within the circal process algebra. In: Proceedings of AMAST'97, Lecture Notes in Computer Science, Vol. 1349. Springer, Berlin Heidelberg New York, pp 108–122
10. Cerone A, Milne GJ (1999) Modelling a subclass of CMOS circuits using a process algebra. In: Proceedings of the 6th annual Australasian conference on parallel and real-time systems (PART'99). Springer, Berlin Heidelberg New York, pp 386–397
11. Cerone A, Milne GJ (2000) A Methodology for the formal analysis of asynchronous micropipelines. In: Proceedings of FMCAD 2000, Lecture Notes in Computer Science, Vol 1954. Springer, Berlin Heidelberg New York, pp 246–262
12. Chu TA, Leung CKC, Wanuga TS (1985) A design methodology for concurrent VLSI systems. In: Proceedings of ICDD, pp 407–410
13. Dill DL (1989) Trace theory for automatic verification of speed independent circuits. MIT, Cambridge
14. Donatelli S, Hillston J, Ribaudo M (1995) Comparison of performance evaluation process algebra and generalized stochastic Petri nets. In: Proceedings of the 6th international work on Petri nets and performance models. IEEE Comp Soc Press
15. Furber SB, Day P (1996) Four-phase micropipeline latch control circuit. IEEE Trans Very Large Scale Integration (VLSI) Syst 4(2):247–253
16. Furber SB, Lin J (1996) Dynamic logic in four-phase micropipelines. In: Proceedings of the 2nd international symposium on advanced research in asynchronous circuits and systems. IEEE Comp Soc Press
17. Gordon MJC, Melham TF (1993) An introduction to HOL—a theorem proving environment for higher order logic. Cambridge University Press
18. Joseph MB, Udding JT (1990) An algebra for delay-insensitive circuits. In: Proceedings of the workshop on computer-aided verification
19. Hoare CAR (1985) Communication sequential processes. International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey
20. Mead C, Conway L (1980) Introduction to VLSI systems. Addison-Wesley, Menlo Park
21. Milne GJ (1991) The formal description and verification of hardware timing. IEEE Trans Comput 40(7):811–826
22. Milne GJ (1994) Formal specification and verification of digital systems. McGraw-Hill, New York
23. Milner R (1984) Communication and concurrency. International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey
24. Milner R, Parrow J, Walker D (1992) A calculus of mobile processes, part I and II. Inf Comput 100(1):1–40, 41–77
25. Moller F (1989) The semantics of Circal. Technical Report HDV-3-89, University of Strathclyde, Department of Computer Science, Glasgow
26. Molnar CE, Fang TP, Rosenberger FU (1985) Synthesis of delay-insensitive modules. In: Proceedings of the 1985 Chapel Hill conference on advanced research in VLSI, pp 67–86
27. Roscoe AW (1997) The theory and practice of concurrency. Prentice Hall, Englewood Cliffs, New Jersey
28. Sutherland IE (1989) Micropipelines. Com of ACM 32(6):720–738
29. UWA–CSSE (2005) The Circal System, Web page. Available via http://www.csse.uwa.edu.au/FormalSpecification/CircalSystem/
30. Vissers CA, Scollo G, van Sinderen M, Brinksma E (1991) Specification styles in distributed systems design and verification. Theor Comput Sci 89:179–206
31. Williams T (1992) Analyzing and improving the latency and throughput performance on self-timed pipelines and rings. In: Proceedings of the IEEE international symposium on circuit and systems. IEEE Comp Soc Press, New York