The Verification of Asynchronous Circuits using CCS

Graham Clark and George Taylor*

Technical Report ECS–LFCS–97–369
Department of Computer Science
University of Edinburgh

October 30, 1997

Abstract: Recent work in the design of asynchronous micropipeline latch controllers suggests that the two-phase protocol may have benefits over the four-phase protocol in terms of speed, energy consumption and not least simplicity. Empirically, circuit behaviour may be analysed using the conventional technique of SPICE simulation. However, in order to gain a high level of confidence in the correctness of these circuits, CCS was used to model several asynchronous latch controllers. This report gives details of the modelling decisions made, and the properties verified for the latch controllers.

1 Introduction

As the demand for functionality continues to increase in the world of VLSI technology, designers are finding it more and more difficult to distribute clock signals, as more circuitry is packed onto silicon. For example, statistics show that 30% of the power consumption of the power-hungry DEC Alpha [1] processor is used in driving the clock signal, which is distributed to each area of the circuit that requires it. Such a globally clocked circuit would be unacceptable for controlling low-power devices such as mobile telephones and laptop computers, which are increasingly in demand.

These concerns are characteristic of synchronous circuits, circuits which operate in lock-step, regulated by a global clock. In order to address these, designers are turning to *asynchronous circuits*, circuits which perform their calculations without being clocked globally. Potential benefits of asynchronous circuits are:

- the complexity of interfaces between circuit modules are simplified as complicated timing constraints underlying synchronous systems (e.g. setup time, hold time) are often no longer required. This eases specification, allows increased design reuse and aids synthesis from high-level languages [2].
- By their nature asynchronous circuits only use power when performing useful work. The global clock present in synchronous designs runs continuously, is the most rapidly changing signal present, and is distributed across a large area, making it undesirable for low-power operation.

*Department of Electrical Engineering, University of Edinburgh

- Since there is no global clock there are no clock distribution problems, in particular that of clock skew.
- In a synchronous design the maximum clock frequency is chosen such that the chip operates correctly under worst case conditions, e.g. high temperature, low power-supply voltage, worst case fabrication process parameters. Asynchronous circuits need not require this (usually large) safety margin, and so exhibit average case rather than worst case performance.

However, there are some disadvantages, such as the overhead incurred by the handshaking protocols, and the extra complexity of the design task [2].

A useful mechanism for increasing the speed at which processors operate is the *pipeline*. This allows the individual stages of a multi-stage process to operate on different items of data at the same time. The asynchronous counterpart of the synchronous pipeline is the *micropipeline*, presented by Sutherland [3]. The transfer of data between micropipeline elements is handled by a circuit called a *micropipeline latch controller*. Obviously it is in the interest of chip designers that these operate quickly, and with the growing concerns over power, should be efficient in their use of energy. To date, the asynchronous community has given support to the *four-phase protocol* for transferring data between micropipeline elements, for reasons such as speed and power consumption [4]. However a recent study by Taylor [5, 6] suggests that latch controllers designed for use with the conceptually simpler *two-phase protocol* may have benefits to offer in terms of both power and speed for linear micropipelines.

Taylor's simplified latch controller circuit has been exercised in typical operating parameters using the well-known circuit simulation tool, SPICE. However the limitations of simulation are well known, and there are interesting and desirable properties that SPICE cannot exhibit conclusively. Therefore, the authors decided to embark on a modelling case study, and make use of formal methods to verify some desirable properties. This report describes the modelling decisions required in order to capture the requisite behaviour of the circuit, and the results proven.

The rest of the report is structured as follows. Section 2 gives some background to the relevant aspects of asynchronous circuits; micropipelines are briefly introduced in order to motivate the discussion of the protocols used. Furthermore, CCS is described, along with its associated temporal logic and tool support. Section 3 describes the issues behind modelling asynchronous circuits in CCS, and the different modelling styles attempted. This leads onto Section 4 which contains details of the CCS models of three asynchronous latch controllers, and the behavioural analysis that was performed. Relevant related work in the area of asynchronous modelling and verification is discussed in Section 5, and compared to our modelling approach. Finally, Section 6 draws some conclusions.

2 Background

In this section, we detail the background material necessary to motivate our modelling study. This includes some material on micropipelines, which were the motivating influence for the circuit implementation by Taylor, and some information on the building blocks for asynchronous circuits, such as the Muller C-element.

Pipelines have proven to be a beneficial mechanism for increasing the speed at which, for example, CPUs operate. Typically, a CPU may take several steps to execute an instruction—this will involve an instruction decode step, and perhaps the fetching of data from memory. Instead of requiring that one instruction complete before the next begins, the result of each stage of the execution of an instruction can be stored in a pipeline, conceptually a FIFO queue. Thus, in theory, and typically in practice, several instructions can be currently under processing in the pipeline. The throughput of the CPU will increase correspondingly. In [3], Sutherland introduces the asynchronous micropipeline. A diagram of a canonical micropipeline is given in Figure 1.

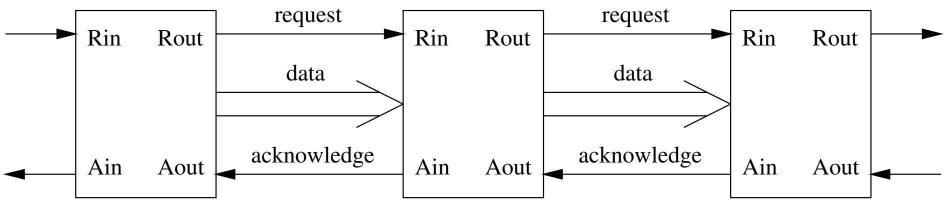

Figure 1: Canonical micropipeline

It is important to note that there is no clock signal to regulate the flow of data from one stage to the next. Each stage in the pipeline can request that the next stage accept new data, and acknowledge receipt of data from the previous stage—each stage can act whenever its immediate environment allows it.

An important feature of micropipelines is their *elasticity*. This means that the amount of data in the pipeline can vary over time. Due to their elasticity, such pipelines would be able to withstand brief increases in the data rate without blocking, and thus throughput could be increased.

2.1 Transition Signalling

A concept employed in micropipeline design that fundamentally differs from that used in conventional circuit design is that of *transition signalling*. Conventional circuit design identifies signals as either "logic high", for example 5 volts, or "logic low", for example 0 volts. Transition signalling observes the transition between logic levels instead, that is the transition of a signal from high to low, or from low to high. Both transitions are identified, and are called *events*. In electrical engineering terms, this means that all responses to transition signals are *edge triggered*. Transition signalling avoids assigning a meaning to the high or low state of control signals; it is only the transition between these states that is recognised. Transition signalling may provide advantages in both speed of operation and simplicity of design [3].

2.2 The Bundled Data Interface

There are several ways in which a sender and receiver may communicate data in the absence of a synchronising clock. When transition signalling is used, there will typically be many data wires between the two, and the sender and receiver may use the *bundled data interface protocol*. This entails an additional two wires between the two units, which here are called *request* and *acknowledge*, as shown in Figure 2.

The data wires carry conventional logic signals, i.e. high or low. When the sender wishes to communicate data, it places it on the data wires, and after that initiates a transition on the request wire.

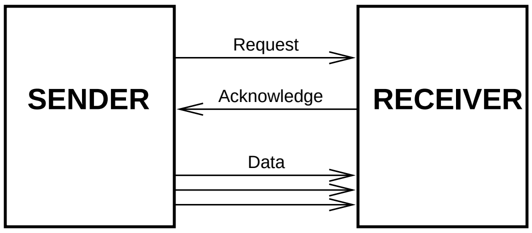

Figure 2: Bundled Data Interface

This may be either a rising transition or a falling transition, depending on the current logic level of the control wire. On receipt of this transition, the receiver knows there is valid data available. When the receiver has taken the data, it will effect a transition on the acknowledge wire. In this way, data passes between the two, and the sender may now proceed and begin the process again when ready. Note that the time taken to communicate data is not fixed.

This scheme is called a bundled data interface because the data and the request wire must be considered a bundle. It must be the case that the data reaches the receiver before the request does in order that the receiver is at no risk of reading the data wires too soon, before the new information appears.

There are alternative schemes to the bundled data interface; see for example *fully delay-insensitive* methods [2].

2.3 Transition Signalling Logic

Asynchronous circuits built using transition signalling typically employ different building blocks than conventional circuits. The design of conventional circuits makes extensive use of such familiar logic gates as AND and NOT. However when the state of a wire is unimportant, and instead events are used as control signals, new modules become more useful in the construction of asynchronous circuits. These form various logical combinations of events. Two pervasive building blocks are shown in Figure 3.

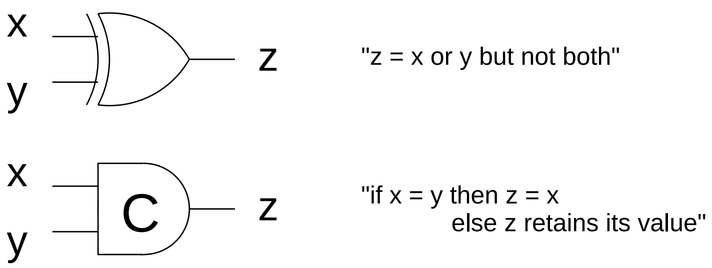

Figure 3: The XOR and the Muller C-element

It can be seen that the Muller C-element (henceforth called the C-element) acts as the AND element for events, and the XOR gate acts as the OR element for events.

The C-element is more complex than a traditional logic element because it keeps state. From Figure 3, the behaviour of the C-element is to hold the output at its current state if the inputs are at different logic levels, and to set the output equal to the input if they are at identical logic levels. Consider the

C-element with both inputs low. The output will follow the inputs, and will be low too. An event on either input causes no immediate change on the output, since the inputs will now differ. However an event on the other input will mean that both signals are now at the same logic level—high—causing the output to go high, and thus an event to be generated. In this way, an event is only generated at the output after an event at both its inputs. The XOR is simpler, and can be seen to perform the OR operation on events by a similar analysis.

There are other commonly used event logic elements, for example Arbiter and Call [3]. Those used in the modelling study will be introduced as they are required in this report. Several elements are used as building blocks in the control circuitry for micropipelines. The latch controller circuits described in Section 4 make use of several different components.

2.4 Pragmatics—Two- and Four-Phase Protocols

The micropipelines discussed by Sutherland, and in fact many implementations, such as the AMULET1 processor [7], make use of *two-phase* control circuits. The two-phase protocol is illustrated in Figure 4.

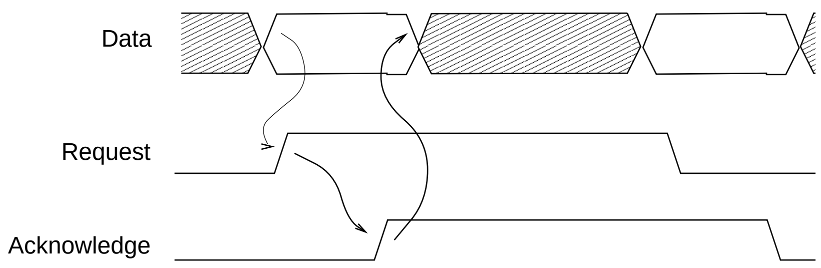

Figure 4: The two-phase protocol

This protocol is event based, and thus rising edges are not distinguished from falling edges. The operation of the communication is as described in Section 2.2. Note that for the second bundle of data, the events are falling edges in contrast with the rising edges for the first. The protocol is called two-phase because only two-phases of operation are distinguished; the sender's active phase, and the receiver's active phase. This scheme has an appealing simplicity, and furthermore may claim to be "low-power" since no control transitions are redundant.

Unfortunately however, certain electronic components are not edge-triggered. For example, latches, gates capable of storing one bit of data, are *level-sensitive*. Sutherland uses a special latch with two event-driven inputs, called a *capture-pass latch*. However, these are expensive in terms of area. Since a latch holds one bit of data, this increase in area would become significant in applications with wide datapaths. In order to use conventional latches effectively, the four-phase protocol was adopted. This is illustrated in Figure 5.

In the four-phase protocol, only one direction of signal transition is significant—arbitrarily, the rising edge is chosen here. The other edge is inactive, and is only required to place the circuit in a state where it can generate another event. A falling edge is part of the recovery phase. Use of this protocol gives practical benefits; four-phase control circuits can feature fewer transistors than two-phase circuits, and previous results show that they outperform their two-phase counterparts in terms of speed, area

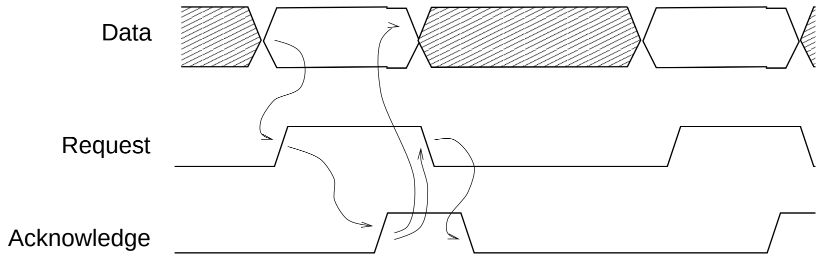

Figure 5: The four-phase protocol

and power [8]. However, conceptually the protocol is wasteful—the "return to zero" transition has no meaning.

This completes the background to asynchronous circuit design. Next, the modelling methodology is described, the compositional approach of process algebra.

2.5 CCS

Milner's Calculus of Communicating Systems (CCS [9]) provides a formal semantic basis for reasoning about concurrent systems. An example of a *process algebra*, systems described in CCS are expressions of the calculus. Larger processes can be built compositionally from smaller ones by using the few basic constructors provided by CCS, for example non-deterministic choice and parallel composition. Processes can be thought of as possessing a set of *ports* through which they can perform *actions* and thus communicate with other processes. By this mechanism they evolve into new processes. This calculus has been shown to be useful for modelling in real-life situations, for instance a part of a nuclear reactor protocol [10]. The syntax of CCS processes is given below.

Prefix: if P is an agent, then $\alpha.P$ is an agent that can perform an α action, evolving into P.

Actions fall into three categories: *input* actions denoted by a, *output* actions denoted by $\overline{\mathsf{a}}$, and a distinguished action, τ, that represents the silent action, or action produced by internal communication.

Definition: if Q is a process and $P \stackrel{\text{def}}{=} Q$, then P is a process that can only behave in exactly the same fashion as Q. This constructor must be used to create recursive definitions.

Summation/Choice: if P and Q are processes, then $P + Q$ is a process that non-deterministically chooses to behave like either P or Q.

Composition: if P and Q are processes, then $P|Q$ is a process that can behave like P and Q, acting independently of each other, or can evolve further by a communication between P and Q if they possess complementary input and output ports. This means that if P could perform the input action a, and Q could perform the complementary output action $\overline{\mathsf{a}}$, $P|Q$ could perform either a, $\overline{\mathsf{a}}$, or the silent action τ resulting from an internal communication between processes P and Q.

Restriction: if P is a process, and L is a set of actions (which cannot contain the silent action τ), then $P\backslash L$ is the process that can behave exactly the same as P, except that it cannot perform any actions contained in L.

Relabelling: if P is a process and $\mathtt{a}$ and $\mathtt{b}$ are actions, then $P[\mathtt{b}/\mathtt{a}]$ is the process that behaves exactly the same as P except that if P can perform $\mathtt{a}$ or $\overline{\mathtt{a}}$ then $P[\mathtt{b}/\mathtt{a}]$ can perform $\mathtt{b}$ or $\overline{\mathtt{b}}$ respectively. This is useful when several similar 'copies' of the same process are required, where only action names should differ.

2.6 Expressing Properties with Modal and Temporal Logic

Hennessy-Milner logic [11], and its temporal extension, the modal mu-calculus [12] are modal logics which can be used to express properties of labelled transition systems. The semantics of a CCS process leads to a labelled transition system, and therefore these logics are well-suited to reasoning about CCS processes. The syntax of mu-calculus formulae is given in Figure 6.

$$\Phi ::= Z \mid \mathtt{tt} \mid \mathtt{ff} \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [\mathtt{a}]\Phi \mid \langle\mathtt{a}\rangle\Phi \mid \nu Z.\,\Phi \mid \mu Z.\,\Phi$$

Figure 6: Syntax of the modal mu-calculus

Here, Z ranges over a set of variables and $\mathtt{a}$ over actions. The semantics of most of these kinds of formulae can be given intuitively. Firstly, all states of a process satisfy $\mathtt{tt}$, and no states satisfy $\mathtt{ff}$. The set of states that satisfies $\Phi_1 \wedge \Phi_2$ is simply the set of states that satisfies both Φ_1 and Φ_2. The set of states that satisfies $[\mathtt{a}]\Phi$ is the set of states such that if P is a member then *for all* processes Q such that P can perform an $\mathtt{a}$ to become Q, Q is in the set that satisfies Φ, i.e. all states such that each can do an $\mathtt{a}$-transition to some state Q that satisfies Φ. The set of states that satisfies $\langle\mathtt{a}\rangle\Phi$ is the set of states such that if P is a member then *there exists* a process Q such that P can perform an $\mathtt{a}$ to become Q, and Q is in the set of states that satisfies Φ.

Hardest to give an intuitive meaning are recursive expressions of the form $\nu Z.\,\Phi$ and $\mu Z.\,\Phi$. An operational way to look at their interpretation is given in Figure 7.

$$\begin{array}{lcll} \nu Z.\,\Phi & \equiv & P^0 \wedge P^1 \wedge P^2 \wedge \cdots, & \text{where } P^0 = \mathtt{tt} \text{ and } P^{i+1} = P[P^i/Z] \\ \mu Z.\,\Phi & \equiv & P^0 \vee P^1 \vee P^2 \vee \cdots, & \text{where } P^0 = \mathtt{ff} \text{ and } P^{i+1} = P[P^i/Z] \end{array}$$

Figure 7: Operational interpretation of fixed point expressions

The mu-calculus can be extended to handle "weak" modalities. This allows it to express properties of systems with respect to their observational behaviour. If a process P satisfies a proposition of the form $\langle\langle a\rangle\rangle\Phi$ then it simply means that there exists a process Q such that P can become Q by some sequence of actions which must include $\mathtt{a}$ only once, but may also include arbitrarily (but finitely) many τ actions. Therefore, to an outside observer, P may become Q by the observable action $\mathtt{a}$. A process P satisfies a formula of the form $[\![\mathtt{a}]\!]\,\Phi$ if for all sequences of actions consisting of one $\mathtt{a}$ action, and arbitrarily (but finitely) many τ actions, that lead to a process Q, Q satisfies Φ. The theory of observational equivalence was employed to determine the correctness of the latch controller models. This is discussed in Section 3.4.2.

Now that the relevant background material to asynchronous circuits has been described, and the modelling theory has been introduced, the next section discusses the various approaches that were taken to modelling circuits.

3 The Study—Modelling Circuits

In this section, we explain the motivation for the modelling study. Asynchronous latch controller circuits studied are introduced, and the various methods that we attempted to use to model these circuits are discussed, illustrating why the final style was adopted. Following this, the models for the circuits themselves are described, and the properties and features we discovered and confirmed are detailed.

3.1 Properties of Asynchronous Circuits

As described in Section 1, the behaviour of circuits at a low level may be determined by the conventional process of SPICE simulation. SPICE models of the lowest level components such as transistors can be made to be extremely accurate, and even with no circuit layout, results are usually a good indication of properties such as power consumption and speed. However asynchronous circuits require additional considerations due to the fact that there is no global clock to ensure everything proceeds in synchrony.

The circuits we will study are designed to communicate via the bundled data interface, and this handshaking based protocol brings obligations. Consider a sender that wishes to make new data available to a receiver. Obeying the protocol, it will set up the new data on the data wires, and then generate an event on the request line. If the propagation time from sender to receiver along the data wires is longer than the time along the request line, the receiver will risk reading old and incorrect data.

In presenting these latch controller circuits, we make the assumption that the interfaces to preceding and subsequent elements in the micropipeline do not suffer from the above problem. However it is necessary to consider what are called *race conditions* within the internals of the circuit itself. If not completely eliminated, they certainly should not offer the potential for what may be considered non-determinism, such that the circuit may behave differently depending on the propagation delay of one (or more) signals.

An example of a race condition is given in Figure 8. The numbers at the gate inputs and output denote the initial conditions. If the event on X arrives at the C-element before the event on Y, an event will be generated on the output. If they arrive in the opposite order, no event will be generated. Clearly this would not be a desirable situation in a useful circuit.

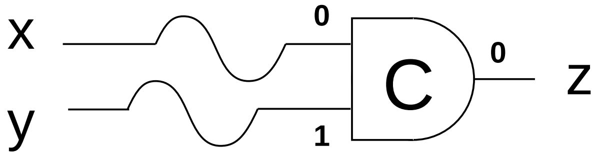

Figure 8: Example of a problem caused by a race condition

In fact, it is the case that race conditions are possible at the level of the transistors which make up the C-element. This means the gate itself could misbehave without requiring any particular and pathological sequence of inputs to occur. However due to the way in which these gates are typically laid out in silicon, the problem will never arise.

The modelling study was carried out in order to confirm the absence of problems such as problematic race conditions. In order to motivate the specific modelling decisions which appear in Section 3, some circuit properties are defined below:

- a circuit is *delay insensitive* if it is possible to insert an arbitrary delay into a gate or a wire without affecting the circuit's qualitative behaviour.
- a circuit is *quasi-delay insensitive* if a circuit is *delay insensitive* with an allowance made for *isochronic forks*. An isochronic fork is a fork in a wire such that the time it takes a signal to reach both ends of the fork, and enter the internals of the connected gates, is identical. If an arbitrary delay can be inserted into any gate or wire while allowing isochronic forks to remain so, and the qualitative behaviour of the circuit is unaffected, then the circuit is quasi-delay insensitive.
- a circuit is *speed independent* if we can insert an arbitrary delay into any gate without affecting the qualitative behaviour of the circuit. The delays in wires are assumed to be negligible (and thus certainly less than the gate delays). At a fabrication level of 0.3 microns, this assumption may cease to be reasonable—wire delays between components may be greater than the actual gate delays [13].

Asynchronous circuits which are produced to be delay insensitive have attracted attention in the literature due to this strong property of independence of timing. In fact it has been shown that the class of circuits which are purely delay insensitive is very small, and arguably not practicable. In [14], Martin shows that the only operator allowed which has multiple inputs is the C-element. In practice, speed independence may prove to be a more achievable goal.

3.2 How Latch Controllers Function

Latch controller circuits have a simple task, and that is to hold the data which is currently stored in the micropipeline. A conceptual representation of a micropipeline element is given in Figure 9.

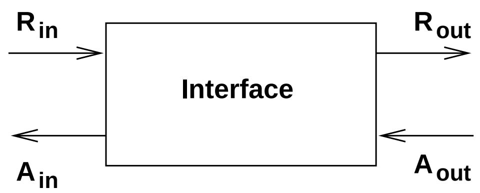

Figure 9: The interface to a micropipeline latch controller

When operating in a linear micropipeline, typically there will be controllers to both the left and the right. The R_{out} and A_{out} outputs will be connected to the R_{in} and A_{in} inputs of the controller on the right, respectively. Similarly, the R_{in} and A_{in} inputs will be connected to the R_{out} and A_{out} outputs of the controller on the left, respectively.

Data is communicated between controllers by the bundled data interface, whether using a two-phase or four-phase protocol. Therefore on receipt of a request event on R_{in}, the controller can safely "latch" the data on its data wires, and transmit both an acknowledge, A_{in}, back to the sender, and an R_{out} onwards to the receiver in the micropipeline. On receipt of an event on A_{out}, the controller knows the data it propagated has been processed by the receiver, and so the controller is now in a position to accept new data from the sender.

3.3 Modelling the Latch Controller Circuits

It was our belief that in the first instance, modelling the simplified two-phase latch controller circuit would be a routine task. In fact it proved to be more subtle, and three approaches were tried before the method was finalised. Very close to our modelling is the work of Liu [15], described in Section 5. We became aware of her research during the early stages of the modelling work, and were heartened to find that Liu used a very similar and natural approach. However, there are some subtle but important differences—these are discussed in Section 3.3.2.

Several issues were crucial in our choice of modelling style:

Suitability of CCS The process algebra seemed to be a clear choice, and the subject area looked well suited to its particular strengths. A circuit can be viewed as a collection of standard elements connected together, with signals propagating between them. The compositionality of CCS meant that this structure could be clearly represented in the model. This is in contrast to basic Petri net approaches which do not support a compositional theory.

Delay Insensitivity The reason the CCS models were constructed was in order to confirm or refute the presence of this property, or a variant. Therefore, obviously the models had to be based on the circuit such that they would reflect, in some way, the presence or absence of delay insensitivity in the circuit.

Clarity This was an important consideration. CCS allows the expression of simple and concise processes—however, these were complicated and cluttered by the need to maintain information related to delay insensitivity (for example, whether two events had arrived "too close together"—see Section 3.3.2). Different modelling styles led to different views of the true behaviour of the asynchronous building blocks, but the modelling method chosen arguably allowed elements such as XOR and the C-element to be modelled more naturally.

The different attempts to model circuits are now described. The novelty comes in the attempt to incorporate support for identifying delay insensitive behaviour.

3.3.1 Modelling Events Efficiently

We began by considering the construction of simple logic gates in CCS. Our first model attempted to capture the detail and function of a typical combinational logic gate, since our working material was ostensibly circuits and signals. This led to the following XOR gate:

$$\begin{array}{rcl} XOR_{00} & \stackrel{\text{def}}{=} & \mathtt{in_1hi}.XOR_{10} \\ & + & \mathtt{in_1lo}.XOR_{00} \\ & + & \mathtt{in_2hi}.XOR_{01} \\ & + & \mathtt{in_2lo}.XOR_{00} \\ & + & \overline{\mathtt{outlo}}.XOR_{00} \text{ etc} \end{array}$$

This style of modelling entails a certain choice in how the model will behave to reflect the circuit. The CCS implementation of the XOR gate above has four states, corresponding to the signal levels on its inputs. Note that in this style, the output is completely characterised by the inputs. The gate will accept high or low signals at either of its inputs, at any time, and on doing so will move to an appropriate next state. Note that this includes receiving a "signal" at an input which is the same as the current value of that input. Furthermore, at any time it will offer an output, which of course corresponds to the XOR of the current values of its two inputs. The initial state of the gate can be chosen by nominating a particular choice of XOR_{ij}.

This style has several shortcomings:

- In electronic terms, it is meaningless to receive a signal which corresponds to a voltage level already present.
- The receipt of a state altering input, and the transition to a new state offering the correct output, is atomic. It will prove more accurate (and indeed necessary for witnessing a lack of delay insensitivity) to model a possible delay between them.
- The gate is modelled at an unnecessary level of detail, and it is more perspicuous and indeed more illuminating to choose an event-based approach.
- From the point of view of analysis, while the modelled circuit may be stable (i.e. all voltage levels are steady), it is still possible for the model to make silent transitions.

A simple observation leads to a far more concise notation for these gates in general. Instead of explicitly representing the transmission and propagation of logic signals, only the propagation of events is modelled. This led to an XOR gate based on the following design:

$$XOR \stackrel{\text{def}}{=} \texttt{in}_1.\overline{\texttt{out}}.XOR + \texttt{in}_2.\overline{\texttt{out}}.XOR$$

This implementation has two states only. In one state, it allows an event at either input, and on receipt of that event it moves to a new state which corresponds to the gate adapting to the new state of its inputs, and attempting to propagate the change by offering an event at its output. Notice that an action must occur on the `out` port after an action on either of the $\texttt{in}_\texttt{i}$ ports. Therefore, the gate correctly performs the OR function on events.

Notably, this reduces the model of an inverter to a simple one-place buffer abstracting from the fact that the input and output of the gate, placed within a circuit, will be logically opposite.

$$Inv \stackrel{\text{def}}{=} \texttt{in}.\overline{\texttt{out}}.Inv$$

This modelling style solves some of the problems mentioned above. The operation of the model is now different—actions only occur when there is information to be communicated (propagated). Therefore, this model does not possess the capability to always offer output to be read. This modelling decision means that if a circuit is in a stable condition, it will be characterised by deadlock in the model. This is a desirable property—the model will not perform actions unless there are changes occuring in the state of the circuit it models.

The new model also has fewer states than the one stated originally. This is partly a consequence of the decision not to model voltages, and instead to model events. However this approach confers no loss of generality. The new XOR model does not distinguish between any initial circuit states; it is not possible to tell the absolute level of the signals at the gate's inputs since these are abstracted. It is true that for certain circuit elements uncharacteristic of asynchronous design, it is necessary to capture the particular signal *level* at gate inputs. An example is the combinational AND gate. The output of an AND gate is logic high if and only if both inputs are logic high. Therefore, if this gate is to be modelled as receiving and communicating events, it is necessary to know the initial state of the inputs. If both inputs are low, then it will take two events, one on each input, to generate what will correspond to an event on the output. If one input is already logic high, then we need only witness an event on the other input to see an event on the output. However, these observations do not imply that it is necessary to model logic levels once more, only that care must be taken to distinguish some internal states of the model.

Perhaps the most useful consequence of the new modelling style is that it captures the property that an electronic gate does not react instantaneously to a change in inputs. Traditional clocked circuits must set a clock frequency low enough such that signals have time to settle into a new state when perturbed. The purpose of this modelling study was to investigate the delay insensitivity of a new latch controller circuit. Therefore, the circuit models must capture the presence or absence of this property in the circuit itself. Since the definition of delay insensitivity considers arbitrary delays in both wires and gates, it suggests that our models must capture the fact that such delays can occur. They are possible, and indeed explicit in our new choice of model, but were absent from the old.

3.3.2 Building Asynchronous Circuits—Compositional Modelling

Given a reasonable design for the asynchronous building blocks, the next task was to join these together to construct a representation of the electronic circuit. The compositionality of CCS provided an algebraic mechanism for constructing larger circuits from smaller ones. However, there were modelling issues to be tackled, such as how to allow two gates to communicate events. Our first choice was perhaps the "obvious" choice, and is illustrated in the following example.

Consider the following CCS definitions, including an XOR gate, and a C-element:

$$\begin{aligned}
XOR &\stackrel{\text{def}}{=} \mathtt{in}_1.\overline{\mathtt{out}}.XOR + \mathtt{in}_2.\overline{\mathtt{out}}.XOR \\
C &\stackrel{\text{def}}{=} \mathtt{in}_1.\mathtt{in}_2.\overline{\mathtt{out}}.C + \mathtt{in}_2.\mathtt{in}_1.\overline{\mathtt{out}}.C \\
C\text{-}XOR &\stackrel{\text{def}}{=} (C[\mathtt{a}/\mathtt{out}]|XOR[\mathtt{a}/\mathtt{in}_1])\backslash\{\mathtt{a}\}
\end{aligned}$$

Recall the C-element performs the AND operations on events. This realisation is functionally correct; an event is only offered at the gate output once an event has been received at each gate input. The third definition is an attempt to combine the XOR and the C-element into a larger part of a circuit. It realises a link between the output of the C-element, and the first input of the XOR. This models a wire connecting the output gate of the C-element, with the first input gate of the XOR in the circuit. This is achieved by using the CCS parallel, relabelling and restriction operators. First the correct ports on the processes are relabelled so they can perform complementary actions. Next the two processes are placed in parallel, free to act independently of each other. Finally, the composition is restricted on the action used in relabelling. This means that if the C-element is to perform an output action, the XOR will witness an input action on the port that was called $\mathtt{in}_1$—both will happen instantaneously, as the result of a CCS "handshake". This models the transmission of an event down the wire between two gates.

We subsequently discovered that our event-based gate models, and our exploitation of compositionality, were discovered by Liu, described in her thesis on modelling the AMULET1 asynchronous microprocessor in CCS [15]. However further analysis revealed a shortcoming with the approach when modelling for delay insensitivity. This can be illustrated by considering the following diagram.

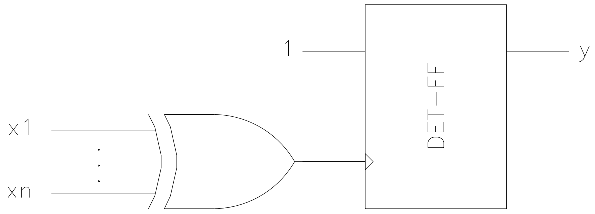

Figure 10: Circuit illustrating a modelling flaw

It represents a simple section of a circuit where the output of an n-input XOR gate is connected to the "clock" input of a *double edge-triggered flip-flop*. The flip-flop has a simple operation—when its clock input changes, the output follows the input. At the moment, the input is at logic high. The problem arises if we consider two signals arriving at two different inputs to the XOR gate. Since there is no global clock, there is no control regulating the time interval between these two signals. When the first arrives, in reality the XOR gate will become *excited*. This means it will be in a state where it will cause a change in the voltage on its output at some (very short) time in the future, without any further stimulus. This phenomenon was described in Section 3.3.1. When the gate produces the change in its output, it is no longer excited. If the second input arrives after this happens, the CCS model will reflect the operation correctly. However, in the asynchronous world, it is perfectly possible for this signal to arrive while the XOR gate is excited. The output of the gate before any events must match the output of the gate after two events, and it may be the case that the original event thus has no effect. We say that the XOR gate has been *quenched* from an excited state. The CCS model described above cannot capture this phenomenon.

It is simple to reason why the current CCS model is deficient. After the arrival of the first input event, the XOR model insists that the next action it is able to perform is an output action. Thus the second input action is "blocked" until the output occurs. If the output occurs, the event has effectively been communicated to the flip-flop. Therefore, it is not possible to witness two inputs to the CCS XOR gate without the output happening after the first. This blocking feature is the reason the current model is inadequate for our purposes.

For this reason, these models, used in their current form, are unsuitable for our application. Modelling to capture the presence or absence of delay insensitivity required that we allow arbitrary events to happen at arbitrary times. Therefore the next task was to modify our models to overcome the problem of blocking.

3.3.3 Overcoming Blocking

We tried two approaches to solving the problem of blocking, both of which are discussed here. The first attempt provided "wires" between gates, an explicit medium for the propagation of events. The second, and ultimately more suitable approach, built a quenching capability into the CCS models of gates themselves.

The motivation behind the first approach was to nullify the blocking problem by providing an explicit mechanism to act as a buffer between gates. We modelled a wire with the following simple CCS definition:

$$Wire \stackrel{\text{def}}{=} \mathtt{in}.\overline{\mathtt{out}}.Wire$$

A wire process was to be used in the construction of larger circuits by acting as a buffer in all connections between gates. For example, our model for the $C\text{-}XOR$ model described above would now become:

$$C\text{-}XOR \stackrel{\text{def}}{=} (C[\mathtt{a}/\mathtt{out}]|Wire[\mathtt{a}/\mathtt{in}, \mathtt{b}/\mathtt{out}]|XOR[\mathtt{b}/\mathtt{in}_1])\backslash\{\mathtt{a}, \mathtt{b}\}$$

Instead of connecting the output of the C-element directly to the input of the XOR, the wire is placed in between, effectively acting as a one-place buffer. The principle of this design was that gates would not be forced to block any inputs since a wire would always be available to read any output event produced.

However this design does not solve the problem—it merely postpones it. Since any suitable model of a wire would have a finite capacity to act as a buffer, it would still be possible to queue finitely many inputs such that the final one would be blocked. A harsh solution to this predicament is to insist that as soon as a position is reached in which any input is blocked, the circuit is in error, and the model should reflect this. This led to an alternative definition of a wire:

$$\begin{array}{rcl} Wire & \stackrel{\text{def}}{=} & \mathtt{in}.Wire' \\ Wire' & \stackrel{\text{def}}{=} & \overline{\mathtt{out}}.Wire + \mathtt{in}.Error \\ Error & \stackrel{\text{def}}{=} & \mathtt{error}.Error \end{array}$$

This gave a simple pre-verification technique. If the circuit was ever capable of generating race conditions leading to a confusion of two signals, then it would be possible to witness an `error` action in the model. Therefore, if any states in the model of the circuit satisfied the mu-calculus formula

$$\mu Z.\, \langle\!\langle \mathtt{error} \rangle\!\rangle \mathtt{tt} \vee \langle\!\langle - \rangle\!\rangle Z$$

which expresses "eventually, an error may occur", then immediately it would be clear that a problem existed.

The second approach takes a different view, and directly models the possibility that a gate may be quenched from an excited state. The final model of the XOR gate is given below:

$$\boxed{\begin{array}{rcl} XOR & \stackrel{\text{def}}{=} & \mathtt{in}_1.XOR_e + \mathtt{in}_2.XOR_e \\ XOR_e & \stackrel{\text{def}}{=} & \mathtt{in}_1 XOR + \mathtt{in}_2.XOR + \overline{\mathtt{out}}.XOR \end{array}}$$

If the XOR receives any input while in an excited state, it is quenched, and returns to its initial state. This is a "hands off" approach to the problem—we take the viewpoint that if a circuit is designed such that quenching can occur, then it is not considered to be in error. If the quenching behaviour does not affect the qualitative behaviour of the circuit, then it is perfectly acceptable.

After subsequent research, we found that this differs from an attitude prevalent in the asynchronous modelling community. In a similar way to the $Error$ process mentioned above, other modelling strategies consider the quenching of a signal to be an error immediately, and model it as such, for example [16, 17]. These differences are discussed later in Section 6.

In the second modelling style, all quenching detection is moved to the gates themselves, and the wire process is now redundant. A possible downside to this is that the function of the gates is made murky by the necessity to allow quenching to occur at all stages. However for the price of unwieldy building blocks, in general we gain a decrease in the size of the state space, making them more tractable for automatic analysis. This can be informally illustrated with the $C\text{-}XOR$ example: with a wire acting as a buffer between the C-element and the XOR gate, the model has 22 states. However when the same section of circuit is modelled without wires, the model has 8 states. The gates with quenching avoid the fundamental problem of blocking inputs and outputs, as do the models with wires which insist on providing "error" states. It seems that the use of wires entails a large increase in the size of the state space of circuit models. Instead, error states could have been built into the gate models themselves, and this is arguably a valid way to continue. However, the semantics of CCS insist that any such model built compositionally would have an error state for each gate modelled. Therefore we can argue that fundamentally, the quenching method will give rise to smaller models.

3.3.4 Further Modelling

Many circuits, including the latch controllers to be modelled, have forks which lead to single events being propagated to more than one destination. It was necessary to model this "fan-out", and this is achieved with the following definition of a Fork process:

$$
\boxed{
\begin{array}{rcl}
Fork & \stackrel{\mathrm{def}}{=} & \mathtt{in}.Fork_e \\
Fork_e & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{out}_1}.Fork_{2e} + \overline{\mathtt{out}_2}.Fork_{1e} + \mathtt{in}.Fork \\
Fork_{1e} & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{out}_1}.Fork + \mathtt{in}.Fork_{2e} \\
Fork_{2e} & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{out}_2}.Fork + \mathtt{in}.Fork_{1e}
\end{array}
}
$$

This fork is non-deterministic, in that once it has received an input event, it may propagate this to its outputs in either order. However once the event has propagated to one output, the next output is determined. Furthermore, since the fork must not block inputs, it is always possible that another input event to the fork will be generated. The effect of this is intuitively determined by how far the fork has proceeded in propagating the first event. It may be the case that the fork has successfully propagated the event down one output, or none at all. If none, then the original event is quenched, and no output events follow. If the original event has been propagated to, say, the first output, then the next input event may prevent the second output being generated, yet still eventually cause a further event on the first output.

The fork process was studied by Tofts *et al.* [18] where they show in special cases, it is possible to replace a non-deterministic definition with a deterministic definition, such that the order of the outputs is pre-determined. They define a class of asynchronous circuits where this optimisation is valid, and does not affect the correctness of the model. However, our use of the fork does not fall into their class, and therefore we retained the more expansive definition.

One further simple element was common to all latch controller models. Each of the circuits is designed to ensure that a micropipeline element can correctly store, and forward, data. Therefore we provided an abstract representation of the storage element, called $Datalatch$. Its definition is given below:

$$\boxed{Datalatch \stackrel{\text{def}}{=} \texttt{in}.\texttt{capture}.\overline{\texttt{out}}.\texttt{in}.\overline{\texttt{out}}.Datalatch}$$

The data latch plays no active part in the model of the circuit. It is simply there to provide an externally observable action, `capture` to indicate that the micropipeline element has correctly captured the sender's data. The model ignores quenching considerations, and justifiably so. It is important to note that it provides a simple interface—one input, and one output. Any element which generates an `in` action will have further interaction with the data latch blocked until the data latch has produced an $\overline{\texttt{out}}$ action. Therefore we can view the data latch as simply a passive indicator.

The circuit diagrams show the data latch in a slightly different position from where they appeared in the construction of the CCS models. This reflects a circuit level optimisation which leads to a faster implementation in practice.

3.4 Analysis with the Edinburgh Concurrency Workbench

When tackling "real-life" problems, the increase in detail, causing large state spaces, rapidly becomes unmanageable without computer assistance. This problem led to the development of an automated tool, the Edinburgh Concurrency Workbench (CWB [19, 20]), which provides facilities for the verification of finite-state (and in some cases infinite-state) processes. A major strength of the Workbench is its versatility; it can bring many verification methods to bear on formally defined systems. For example, it is capable of:

- determining the equivalence of two processes, using any one of a number of relations (e.g. weak bisimulation equivalence [9]).
- automatically deriving logical formulae (expressed in Hennessy-Milner logic) that distinguish two non-equivalent processes.
- manipulating a process in a number of ways e.g. analysing its state space and interactively simulating its behaviour.
- model checking using the modal mu-calculus.

3.4.1 Circuits with the CWB

Having decided on a modelling style, constructing circuit models to represent the various latch controllers studied was straightforward. An excerpt from some early CWB code to represent the two-phase AMULET1 latch controller is given in Figure 11.

The correspondence between the code and CCS notation should be obvious. In this way, we could arrive at a final representation of the circuit, which was then placed in a suitable "context". Since latch controller circuits act as elements in micropipelines, their interfaces (i.e. their externally visible actions) were connected to processes that would mimic the environment of such a circuit. Input request and acknowledge lines were connected to a process that was continually able to act as though

```
set A = { a } ;

agent C' = C1[rin/in2,b/in1,a/out] ;
agent XOR' = XOR[aout/in2,a/in1] ;
agent C-XOR = ( C' | XOR' ) \ A ;

agent Datalatch' = Datalatch[a/in] ;
agent C-XOR' = C-XOR[a/out] ;
agent C-Data = ( C-XOR' | Datalatch' ) \ A ;
```

Figure 11: Excerpt from a CWB representation of the AMULET1 latch controller

it was supplying data, and output request and acknowledge lines were connected to a process that could act as though it was continually able to receive data. This is illustrated in Figure 12.

```
set   Env = { rinp, routp, ainp, aoutp };

agent Env-L   =  rin.'rinp.Env-Lw;
agent Env-Lw  =  ainp.('ain.Env-L + ainp.Env-Lw);
agent Env-R   =  routp.('rout.aout.'aoutp.Env-R
               + routp.Env-R);

agent Implementation = ( Env-L | Circuit | Env-R ) \ Env;
```

Figure 12: Modelled interface to latch controller circuits

3.4.2 Verification with the CWB

From here, the verification technique was simple, and standard in the concurrency community. If all is well with the operation of the individual latch controller circuits (and assuming of course that we have a faithful model), then the behaviour of the models to an external observer should follow a simple pattern. That pattern can be expressed by the following CCS specification for a two-phase latch controller, $Spec$:

$$\boxed{\begin{array}{rcl} LS & \stackrel{\text{def}}{=} & \mathtt{r_{in}}.\mathtt{sync}.\mathtt{capture}.\overline{\mathtt{sync}}.\overline{\mathtt{a_{in}}}.LS \\ RS & \stackrel{\text{def}}{=} & \overline{\mathtt{sync}}.\mathtt{sync}.\overline{\mathtt{r_{out}}}.\overline{\mathtt{a_{out}}}.RS \\ Spec & \stackrel{\text{def}}{=} & (LS|RS)\setminus\{\mathtt{sync}.\} \end{array}}$$

Since we had confidence in the specification, the task was only to show that the implementations we studied were equivalent to the specification. The notion of equivalence we chose was Milner's *observation equivalence* [9]. Informally, if two processes cannot be distinguished by an observer with which they may interact, then they are considered observationally equivalent. Implicit in this notion is that silent (or τ) actions cannot be witnessed. Fortunately, the CWB provides a facility to automatically determine if two processes are equivalent in this way. This was used to attempt to show that to an observer, several latch controller circuits, including the circuit by Taylor, were equivalent to the specification of a correct two-phase latch controller. These results are discussed in Section 4.

It is important to note the strength of the result we obtain by determining equivalence in this way. First of all, all CCS models abstract from an explicit notion of time. By modelling only causality, we implicitly capture the notion that signals may propagate *through* gates, and *between* gates with arbitrary delays. By our choice of modelling design, we furthermore removed all unfaithful event "blocking" characteristics from the latch controller models, whereby gates could "hold up" incoming signals. The family of delay insensitivity properties require that we do not place timing constraints on any part of the circuit, and through the two measures above, this we believe we have achieved. There are certain caveats, as there will be with any modelling approach. As pointed out in Section 3.1, in reality we cannot hope that the basic building blocks themselves can truly be delay insensitive. Therefore, more accurately we can claim that assuming the circuit, when being implemented in silicon, is laid out such that gates will suffer no timing problems internally, then the circuit is delay insensitive. This is still a useful result, since in reality, these pathological problems do not occur. We therefore believe that a positive result in such an equivalence test gives us strong grounds for claiming that if the modelled circuit was implemented, it would be delay insensitive.

4 Results

Three latch controller circuits were modelled and tested for delay insensitivity. The first was the latch controller used by the AMULET group at the University of Manchester in the design of their AMULET1 processor [7]. It operates using the two-phase bundled data interface. The second was the simplified latch controller studied by Taylor in [5]; this also uses the two-phase bundled data interface. The final circuit modelled was a semi-decoupled latch controller used in the design of the AMULET2 processor [8]. This operates using the four-phase bundled data interface. In this section, the results from each study are explained. The first of these studies, the AMULET1 micropipeline latch controller, is discussed next.

4.1 Issues and Results for the AMULET1 Latch Controller

The first circuit modelled was used in the design of the AMULET1 processor [7]. The circuit is illustrated in Figure 13.

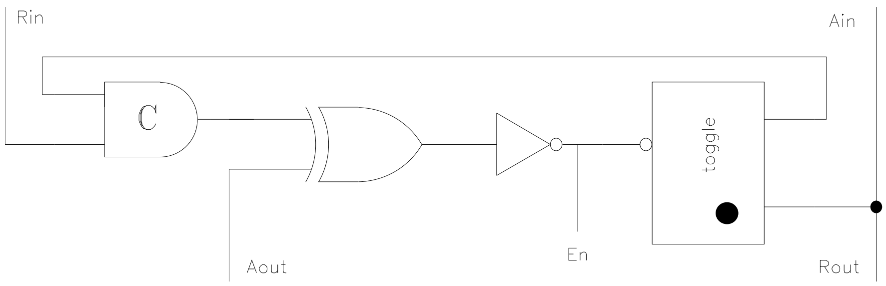

Figure 13: AMULET1 micropipeline control circuit

Their design incorporates conventional level-sensitive latches, and the asynchronous *Toggle* element. However since the latch is level-sensitive, the two-phase protocol is not directly suitable for its control, and internal circuitry to convert to and from a level-sensitive (four-phase like) representation is needed.

The XOR element provides the conversion from event signals to a level controlled signal and the Toggle element the reverse conversion. For each event the Toggle receives on its input, it generates an output event, alternating between the two outputs. The dot indicates which output is chosen first after initialisation. The Toggle element is present to "sense" when the data latch (connected to the enable En signal) has completed its operation.

Modelling the Toggle element was straightforward, employing the quenching style. Its definition is given below:

$$\begin{array}{|rcl|}\hline Toggle_1 & \stackrel{\text{def}}{=} & \texttt{in}.Toggle_{2e} \\ Toggle_{2e} & \stackrel{\text{def}}{=} & \overline{\texttt{out}_2}.Toggle_2 + \texttt{in}.Toggle_1 \\ Toggle_2 & \stackrel{\text{def}}{=} & \texttt{in}.Toggle_{1e} \\ Toggle_{1e} & \stackrel{\text{def}}{=} & \overline{\texttt{out}_1}.Toggle_1 + \texttt{in}.Toggle_2 \\ \hline\end{array}$$

With this new definition, it was possible to construct a model of the circuit shown in Figure 13. This model is given in full in Appendix B. The CWB was used to determine whether the model was equivalent to the two-phase specification given in Section 3.4.2. The following dialogue is a transcript of the output:

```
Command: eq(SpecTwoPhase,Implementation);
true
Command:
```

This analysis backs up an accepted fact that this circuit is delay insensitive (subject to the conditions described in Section 3.1). This gave us confidence that our modelling technique was effective.

Day and Woods analyse this circuit in [21]. It performs unimpressively, and one of the causes is the Toggle element. This is expensive to implement in silicon, requiring many transistors, and is relatively slow. Previous results from studies of the AMULET1 processor [4] indicate that the asynchronous processor does not perform significantly better than a synchronous counterpart. Day and Woods conclude that the performance of the micropipeline controllers must be improved. They propose that switching to a four-phase bundled data interface protocol would remove the requirement for conversion circuitry in the controller, and thus improve its performance.

4.2 Issues and Results for the Simplified Latch Controller

With confidence in our modelling technique, we progressed onto modelling Taylor's two-phase micropipeline control circuit, described in [5]. The behaviour, while correct under restricted SPICE simulations, was, under all timing circumstances, unknown to us. The circuit consisted of only a few elements, and the hope was that it was delay insensitive. It is illustrated in Figure 14.

Elements l_1 and l_2 are conventional latches, that is they are level-sensitive, rather than transition sensitive. When the latch enable input is logic high, it allows its output to follow the value on its second input (it "latches" the data). At all other times, the value on the output stays constant. Our model of a latch is given below. It illustrates how to model a conventional level-sensitive logic gate in terms of events.

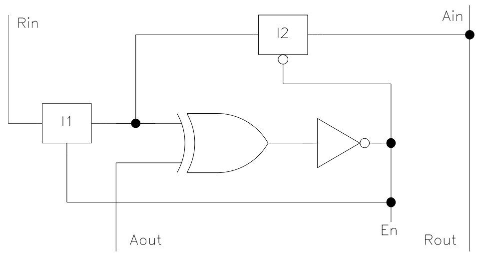

Figure 14: Taylor's two-phase micropipeline control circuit

$$
\begin{array}{rcl}
\mathit{Latch\text{-}En} & \stackrel{\mathrm{def}}{=} & \mathtt{in}.\mathit{Latch\text{-}En}_e + \mathtt{enable}.\mathit{Latch\text{-}Dis} \\
\mathit{Latch\text{-}En}_e & \stackrel{\mathrm{def}}{=} & \mathtt{in}.\mathit{Latch\text{-}En} + \mathtt{enable}.\mathit{Latch\text{-}Dis}_e + \overline{\mathtt{out}}.\mathit{Latch\text{-}En} \\
\mathit{Latch\text{-}Dis} & \stackrel{\mathrm{def}}{=} & \mathtt{in}.\mathit{Latch\text{-}Dis}_e + \mathtt{enable}.\mathit{Latch\text{-}En} \\
\mathit{Latch\text{-}Dis}_e & \stackrel{\mathrm{def}}{=} & \mathtt{in}.\mathit{Latch\text{-}Dis} + \mathtt{enable}.\mathit{Latch\text{-}En}_e
\end{array}
$$

An event on a latch input means a low-high or a high-low transition, and therefore after an event, there will be a conventional logic value on the input, whether it be high or low. Therefore, on receipt of an event on its enable input, the latch toggles between two sets of states: those that will subsequently generate an output event given an input event, and those that will suppress output events. Note that the latch model essentially stores the state of the conventional logic value on its input, and propagates any output events on receiving an enable signal, if necessary.

The CCS model of Taylor's latch controller was constructed, and submitted to the CWB for analysis. Again, the complete model can be found later in this report, in Appendix C. Given the same test as the Toggle circuit in Section 4.1, the Workbench this time gave a different result:

```
Command: eq(SpecTwoPhase,Implementation);
false
Command:
```

Of course the CWB is able to explain precisely why two processes differ. This it does by supplying a modal logic formula that distinguishes the two. This was the natural next question to ask, and produced the following from the CWB:

```
Command: dfweak(SpecTwoPhase,Implementation);
[Major collection... 85% used (16272008/19121476), 5300 msec]

[Increasing heap to 47722k]
<<rin>>[[capture]]<<'ain>><<rin>><<'rout>>[[aout]][['ain]]F
Command:
```

This formula is true of the two-phase specification, but is false of the simplified latch controller. Informally, the following sequence of events is possible:

1. An R_{in} arrives from the sender. The event propagates through the transparent latch l_1, and generates a `capture` action, meaning the data-latch has captured the data provided by the sender. At this stage, we assume that the signals have propagated through the circuit, so that l_1 becomes opaque, and l_2 transparent.

2. The latch controller generates an A_{in} event to acknowledge the data from the sender; the sender then generates an R_{in} to signal it has new data ready; and the latch controller produces an R_{out} event to tell the receiver that it is ready to provide it with data.
3. The receiver acknowledges it has received the new data with an A_{out} event to the controller.

It is here that in theory, things can go wrong. Since delays are modelled to be possible everywhere, we may suppose that once the signal propagates through the XOR gate, it reaches latch l_1 before latch l_2. In this pathological case, the event blocked at the input to l_1 will pass through, and may propagate through the still transparent l_2 to produce an A_{in} event. This corresponds to the problem witnessed by the modal logic formula. In fact the formula states a stronger version of this explanation. The "boxed" actions state that we can arrive at a state of the model such that *no matter* how events propagate to produce the $\overline{\texttt{ain}}$, `rin` and $\overline{\texttt{rout}}$ actions, it is then possible to witness an `aout` leading to a spurious $\overline{\texttt{ain}}$.

It can be seen that this problem arises as a result of latches l_1 and l_2 both being transparent at the same time. This suggested a way to address the problem apparent in the model. Both latch enable inputs are driven by the same signal. If that signal could be forced to arrive at both enable inputs "at the same time", then both latches would never be transparent simultaneously. This suggested that we should model the fork that appears after the data latch in the circuit as isochronic. This is a reasonable assumption, since there is no circuitry between the fork and both latch enable inputs to delay the propagation of any signal.

Modelling the isochronic fork required care, since following the principles explained in Section 3.3.2, gate inputs and outputs must never be blocked. Ensuring isochronicity entailed a doubling in the size of the latch model. Essentially, an `enable` action causes the latch to enter a set of "mirror" states in which a further `enable` is not possible. Importantly, no other actions are blocked. In these mirror states, instead of an `enable` action, a `sync` action is possible. In the final model of the latch controller l_1 is capable of `sync`, and the actions of l_2 are renamed so that instead it may produce the complementary $\overline{\texttt{sync}}$ action.

$$
\begin{array}{rcl}
\mathit{Latch\text{-}En} & \stackrel{\mathrm{def}}{=} & \texttt{in}.\mathit{Latch\text{-}En}_e + \texttt{enable}.\mathit{Latch\text{-}En}^{wait} \\
\mathit{Latch\text{-}En}^{wait} & \stackrel{\mathrm{def}}{=} & \texttt{in}.\mathit{Latch\text{-}En}_e^{wait} + \texttt{sync}.\mathit{Latch\text{-}Dis} \\
\mathit{Latch\text{-}En}_e & \stackrel{\mathrm{def}}{=} & \texttt{in}.\mathit{Latch\text{-}En} + \texttt{enable}.\mathit{Latch\text{-}En}_e^{wait} + \overline{\texttt{out}}.\mathit{Latch\text{-}En} \\
\mathit{Latch\text{-}En}_e^{wait} & \stackrel{\mathrm{def}}{=} & \texttt{in}.\mathit{Latch\text{-}En}^{wait} + \texttt{sync}.\mathit{Latch\text{-}Dis}_e + \overline{\texttt{out}}.\mathit{Latch\text{-}En}^{wait} \\
 & & \text{etc}
\end{array}
$$

The justification for this modification is as follows. On receiving an `enable` action, the latch's behaviour as regards all other actions is *unchanged.* In order for it to alter that behaviour, and in fact be consistent with the first latch model after receiving an `enable` action, it must perform a `sync` action too. However, the final circuit model insists that in order to perform this `sync` action, the other latch must perform the corresponding $\overline{\texttt{sync}}$ action, and the two latches therefore synchronise. At no point are `in` and $\overline{\texttt{out}}$ actions blocked. Therefore, in the model, the effect of the two latch enable signals happens "at the same time".

The simplified latch controller was modelled with the new latch model in place of the old, and submitted to the CWB. This time the outcome was as hoped:

```
Command: eq(SpecTwoPhase,Implementation);
true
Command:
```

This gave us confidence that the simplified latch controller circuit, with the isochronic fork assumption, behaved correctly regardless of delays. In fact, we could consider the simplified latch controller to be quasi-delay insensitive, that is delay insensitive with an allowance for isochronic forks.

4.3 Issues and Results for the Four-phase Semi-Decoupled Latch Controller

The final circuit we modelled appears in [21], and with further comments in [8]. It differs from the previous two in that its external interface provides *four-phase* control, instead of using the simpler two-phase protocol. In [21], Day and Woods argue the case for this decision; since the two-phase controllers we consider use conventional level-sensitive data latches, the controllers must incorporate circuitry to accommodate these latches. As described earlier, this is referred to as two-to-four and four-to-two phase conversion circuitry. These names are slightly misleading—the two-phase controllers still present a two-phase interface externally. Day and Woods argue that by presenting a four-phase interface externally, they can dispense with this extra circuitry, and thus make savings in speed and power.

The circuit presented in [8] is illustrated in Figure 15.

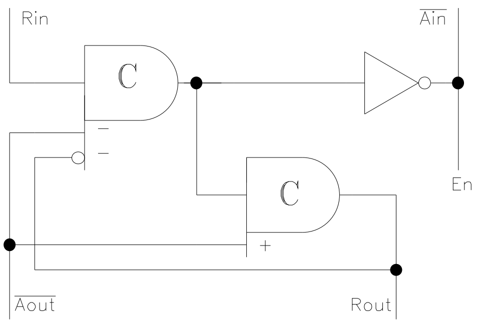

Figure 15: Four-phase micropipeline control circuit

The C-elements which feature are modified versions of those which appear elsewhere in this report. It can be seen that one C-element has three inputs. Its function is the obvious generalisation of the two-input case—the output will follow the inputs if all inputs are equal, otherwise it will retain its previous value. The other difference is that both C-elements present in Figure 15 are *asymmetric*. The '–' symbol next to the input means that the input can only play a part in the output going from logic high to logic low. If the output is currently logic low, it may change back to high without requiring an event on that input. These modifications are more an interpretation at a gate level of the circuit at a transistor level, and this inevitably leads to a more encumbered notation.

The circuit is *semi-decoupled* because due to its design, dependencies exist between the latch controller's inputs and outputs. For example, the circuit cannot generate a resetting high-to-low A_{in} transition until it receives a low-to-high event on A_{out}. A latch controller circuit which conforms to the four-phase bundled data interface protocol need not contain such dependencies. However they lead to a smaller and faster implementation in practise, and despite these restrictions, the circuit sufficiently implements the four-phase protocol. Note that in our model of this latch controller, the data latch (the En signal in Figure 15) appears directly after the three-input C-element, which is necessary for delay insensitive operation.

The reason we modelled this particular controller circuit was that in [21], Furber and Day note that it suffers from a *persistency* problem, which was first pointed out by Yakovlev with his FORCAGE tool [22]. It is important to note that this circuit was designed to be speed-independent, and not delay-insensitive. Our aim was to reproduce this error by modelling a speed-independent circuit in our setting, and thus give us more confidence in our technique.

As mentioned, the circuit presents a four-phase interface. Therefore low-high transitions are significant. In modelling terms, this means that one sequence of events will cause the data latch to capture its data, and the next sequence will exist simply for the purpose of ensuring the inputs are ready for the next set of low-high transitions. This meant that we had to produce a new CCS specification process, which acted like a perfect semi-decoupled four-phase latch controller. Since our aim was to reproduce the error discovered by Yakovlev, we based our specification on that presented by Day and Woods in [21], which they represented using *signal transition graphs* (STG). This is a graphical notation based on Petri nets; we used a small program to naively generate a CCS process whose state space was identical to that of the STG specification, and were therefore confident we had an equivalent specification.

The CCS model of the circuit was constructed, once again making use of the isochronic fork technique described in Section 4.2. We ensured that any A_{out} events arrived at the appropriate inputs to the C-elements "at the same time". This done, the circuit model was submitted to the CWB for comparison against the specification.

```
Command: eq(SpecFourPhase,Implementation);
false
Command:
```

This was encouraging, and so again we asked the CWB to produce a modal logic formula which embodies a behaviour which distinguishes the two processes.

```
Command: dfweak(SpecFourPhase,Implementation);
<<rin>><<capture>>[['ain]][['rout]]<<aout>><<rin>>[['ain]]
<<rin>>[[capture]]F
Command:
```

Informally, this expresses that in a circuit containing the implemented latch controller, the following sequence of actions is possible:

1. An R_{in} arrives from the sender. The data latch captures the data it is confident is now available, and the event propagates through the three-input C-element leading first to an A_{in} event.

2. The latch controller circuit continues to propagate the effect of the first R_{in}, which causes an R_{out} to be sent to the receiver. The receiver immediately acknowledges this by generating an A_{out} event.

3. Since the sender's first R_{in} has been acknowledged by the controller with an A_{in}, it is quite justified in generating another R_{in} (the high-to-low reset), which it does.

4. The next event witnessed is an A_{in}, the acknowledgement by the latch controller of the high-to-low reset R_{in}. This means the R_{in} must have propagated through the three-input C-element, and therefore both

    - the event propagating round the fork from the R_{out} event, and
    - the event propagating along the fork caused by the A_{out} event

    have reached the negative inputs of the three-input C-element.

5. Because the controller has acknowledged the high-to-low reset, the sender may and does now generate another R_{in} event, indicating it has new data to pass on.

It is at this point that a problem exists, in theory at least. The controller should not allow this event to propagate because the effects of the previous high-to-low reset transition may not yet have propagated fully in the circuit. However because two of the three inputs to the C-element cannot prevent propagation of a low-to-high event, the R_{in} does immediately propagate. This leads to a potential race condition on the wire leading to the two-input C-element. The specification of this circuit does not allow the final R_{in} to propagate through the three-input C-element, yet the implementation clearly does. We note, as in [8], that the race condition illustrated is pathological, but nevertheless it exists, and our confirmation of this problem gave us more confidence in our modelling technique.

5 Previous Work

Much recent work has been invested in methods and tools for the design of asynchronous circuits. An example of this is the Rainbow system [23], a tool currently being developed at the University of Manchester. Rainbow encompasses a "wide spectrum" hardware description language (HDL) for a multi-view description of micropipeline designs. For example, the user may view a model hierarchically, as a structure compositionally built from micropipeline primitives, or from an algorithmic control-flow point of view. The language is built on a formal basis, a process algebra called APA. This gives a sound foundation on which the Rainbow team are building support for analysis, simulation and verification.

Relevant work has also been carried out by Yakovlev at the University of Newcastle upon Tyne. His approach is based upon the use of Petri nets and variants thereof. Yakovlev has developed a modelling methodology [24] that allows him to verify interesting properties of circuits—one such property of particular relevance here is called *persistence*.

If a circuit is persistent it means that when any signal makes a transition, it cannot subsequently make the inverse transition until all immediate successor transitions have occurred. This makes intuitive sense as a desirable property in asynchronous circuits—failure would mean that a signal could "catch up with" another. For example, for the initial version of the semi-decoupled four-phase latch controller presented by Day and Woods [21], Yakovlev showed that there is a timing assumption (restated by Furber and Day in [8]). Although it is unlikely to manifest itself as a problem in a real circuit with reasonable layout, it nevertheless prevented the circuit from possessing the persistency property. Furber and Day correct this "deficiency".

Perhaps the work of most relevance to this report is the PhD thesis (and to some extent, the MSc thesis) of Ying Liu [15]. Her aim was to model the AMULET1 processor in CCS, and to verify aspects of its behaviour using the CWB. This was the first approach to modelling asynchronous hardware using CCS known to the authors, and served as a guide for the work described in this report. The project was a success in that Liu successfully specified and verified many useful properties of a practical complete microprocessor. In fact the verification was *post facto*, but incorporation of the work into the design phase should have definite benefits. Liu tested properties such as deadlock, livelock freedom, safety and liveness. However little mention is made of persistency properties of low level components, such as delay insensitivity and speed independence. Liu concluded that CCS and the Workbench were suitable tools for the modelling and verification work, a fact which improved our confidence in our choice of formalism.

We must also mention the work of Gopalakrishnan [17] in which he compares CCS to Dill's Trace theory for the purposes of modelling asynchronous circuits. Our work is similar in nature to his, although a crucial difference is in our attitude over the phenomenon of quenching. Gopalakrishnan, as is more commonplace, models asynchronous circuits with choke states, which are used to indicate problems with the behaviour of circuits.

6 Conclusions

We believe that the modelling study detailed in this report was successful in meeting its aims.

- We developed a compositional modelling methodology for asynchronous circuits which gave us the power to: verify the AMULET1 latch controller as delay insensitive; verify the simplified latch controller as quasi-delay insensitive i.e. delay insensitive with the assumption that one fork is isochronic; and reproduce the bug in the AMULET2 latch controller as discussed by Day and Woods [21].
- We have shown that process algebra in general provides a useful framework for the systematic development of circuit models, and a powerful mathematical theory for their verification. In particular CCS with its associated tool support proved to be a congenial choice.
- We have shown that even simple looking circuits can have subtle behaviour which may elude an informal analysis or particular simulation run.

However this study was not without its problems. One which caused problems was the sheer size of the circuit models, and the trouble the CWB had with their analysis. Models such as the AMULET1 latch controller presented little problem; it is clear from the circuit that its operation is constrained, and there is little scope for concurrency in the propagation of events between gates. When the circuit is placed within an environment that correctly communicates with the circuit's interface, it becomes highly constrained, and the state space size is small. However for circuits which do not perform correctly under all timing assumptions, there is no such tight constraint. The Workbench, in analysing such models, may attempt to generate large portions of the state space via unobservable actions before it discovers an observable difference to the specification. Our attempts to generate distinguishing modal logic formulae took from several minutes to several hours. The smallest formula which distinguishes the simplified latch controller from the specification contains seven weak logical modalities, and this gives no indication of the number of unobservable actions which may be necessary too.

A useful technique we employed was discussed by Liu [15]. This was to make use of the CWB "minimisation" operator, which given a process, computes the smallest process to which it is observationally equivalent. The theory of observational equivalence then justifies the use of the smaller process in place of the larger one[1]. This meant that as the model of the circuit was constructed compositionally, the maximal intermediate state space size could be kept lower, and the total time for analysis kept shorter.

However we do not conclude on whether explicitly modelling error conditions, or instead modelling quenching, is the most suitable modelling style for asynchronous circuits. We take the approach that as long as any quenching which may occur in a circuit does not affect the qualitative behaviour of the circuit, then it is harmless. However, we have no example of a circuit in which quenching is indicative of anything but an error. More positively, and perhaps more philosophically, it does seem that this leads to a more natural model of a logic gate. The model captures possibilities which may genuinely occur in a real-life circuit. Most other approaches deem that quenching behaviour is erroneous, and immediately signal so, perhaps by entering an error state. Empirically it seems that this leads to slightly larger models, at least with conventional process algebras, but it does inevitably lead to models of circuit elements which are "artificial". Certainly circuit elements have no concept of an error state.

The authors hope that this report may prove useful to others embarking on a study of asynchronous circuits, and that it provides an alternative approach to verification. It certainly seems that there is scope in commercial CAD packages for verification of some form.

Acknowledgements

We would like to thank Graham Birtwistle for a very helpful discussion, and for providing us with several papers. We would also like to thank Alex Yakovlev for information on his approach, and Jane Hillston and Gerard Blair for feedback and many comments. This work was supported by EPSRC awards 95305666 and 95306547.

References

[1] R. Cornerford, "How DEC developed Alpha," *IEEE Spectrum*, pp. 26–31, July 1992.

[2] S. Hauck, "Asynchronous design methodologies: an overview," in *Proceedings of the IEEE*, vol. 83, pp. 67–93, January 1995.

[3] I. E. Sutherland, "Micropipelines," in *Communications of the ACM*, vol. 32, pp. 720–738, June 1989.

[4] S. Furber, P. Day, J. Garside, N.C.Paver, S. Temple, and J. Woods, "The design and evaluation of an asynchronous microprocessor," in *Processings of ICCD'94*, October 1994.

[5] G. S. Taylor and G. M. Blair, "Reduced complexity two-phase micropipeline latch controller," in *ESSCIRC'97*, pp. 304–307, September 1997.

[6] G. Taylor, G. Clark, and G. M. Blair, "Application of CCS to verify a simplified two-phase micropipeline latch controller," in *2nd UK Asynchronous Forum*, July 1997.

[1] Note not generally in choice contexts, which we avoided

[7] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods, “AMULET1: A Micropipelined ARM,” in *CompCon’94*, March 1994.

[8] S. Furber and P. Day, “Four-phase micropipeline latch control circuits,” in *IEEE Transactions on VLSI Systems*, vol. 4, pp. 247–253, June 1996.

[9] R. Milner, *Communication and Concurrency*. International Series in Computer Science, Prentice Hall, 2nd ed., 1989.

[10] G. Bruns and S. Anderson, “The formalization and analysis of a communications protocol,” Tech. Rep. ECS-LFCS-91-115, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, 1991.

[11] M. Hennessy and R. Milner, “On observing nondeterminism and concurrency,” in *Proceedings* 7^{th} *ICALP,* Noordwijkerhout (J. W. de Bakker and J. van Leeuwen, eds.), vol. 85 of *Lecture Notes in Computer Science*, pp. 299–309, Springer-Verlag, July 1980.

[12] D. Kozen, “Results on the propositional mu-calculus,” *Theoretical Computer Science*, vol. 27, pp. 333–354, 1983.

[13] J. L. A. van de Snepscheut, *Trace Theory and VLSI Design*. PhD thesis, Technische Hogeschool, Eindhoven, 1983.

[14] A. J. Martin, “The limitations to delay-insensitivity in asynchronous circuits,” in *Beauty is our business : a birthday salute to Edsger W. Dijkstra*, ch. 34, Springer Verlag, 1990.

[15] Y. Liu, *AMULET1: Specification and Verification in CCS*. PhD thesis, Department of Computer Science, Calgary, Alberta, 1995.

[16] M. B. Josephs, “Receptive process theory,” in *Acta Informatica 29, 17-31, Springer-Verlag*, 1992.

[17] G. Gopalakrishnan, “Asynchronous circuit verification using trace theory and CCS,” June 1995.

[18] C. Tofts, Y. Liu, and G. Birtwistle, “State space reduction for asynchronous micropipelines,” in *BCS-FACS Northern Formal Methods Workshop*, 1996.

[19] R. Cleaveland, J. Parrow, and B. Steffen, “The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems,” *ACM Trans. Programming Languages and Systems*, vol. 15, pp. 36–72, Jan. 1993.

[20] F. Moller and P. Stevens, *The Edinburgh Concurrency Workbench (Version 7)*. Dept. of Computer Science, University of Edinburgh.

[21] P. Day and J. V. Woods, “Investigation into micropipeline latch styles,” in *IEEE Transactions on VLSI Systems*, vol. 3, pp. 264–272, June 1995.

[22] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky, *Concurrent Hardware—The Theory and Practice of Self-Timed Circuits*. New York: Wiley Series Parallel Computing, 1994.

[23] H. Barringer, D. Fellows, G. Gough, and A. Williams, “Abstract modelling of asynchronous micropipeline systems using Rainbow,” in *CHDL’97*, April 1997.

[24] A. V. Yakovlev, "On limitations and extensions of STG model for designing asynchronous control circuits," in *International Conference on Computer Design, VLSI in Computers and Processors*, (Los Alamitos, Ca., USA), pp. 396–400, IEEE Computer Society Press, Oct. 1992.

A CCS Models of Asynchronous Circuit Elements

```
****************** CCS models of controllers/environment *****************


* SpecTwoPhase
* A correct two phase latch controller operating in a valid
* environment.
*       agent:                  SpecTwoPhase
*       observable actions:     rin,ain,rout,aout,capture
*   ('capture' is to observe when the data latch captures new data)
*
agent   SpecTwoPhase-L  =  rin.sync.capture.'sync.'ain.SpecTwoPhase-L;
agent   SpecTwoPhase-R  =  'sync.sync.'rout.aout.SpecTwoPhase-R;
agent   SpecTwoPhase    =  ( SpecTwoPhase-L | SpecTwoPhase-R ) \ {sync};




* EnvTwoPhaseL, EnvTwoPhaseR
* A valid two-phase environment for a micropipeline to operate in.
* The inputs (that is outputs from the micropipeline) allow quenching.
*
* Example:
*   agent Implementation = ( EnvTwoPhaseL | Micropipeline | EnvTwoPhaseR )
*                          \ ENVTWOPHASESET;
*
*       agents:                         EnvTwoPhaseL, EnvTwoPhaseR
*       observable actions:             rin,ain,rout,aout
*       interface to micropipeline:     rinp,ainp,routp,aoutp
*
agent   EnvTwoPhaseL    =  rin.'rinp.EnvTwoPhaseL-w;
agent   EnvTwoPhaseL-w  =  ainp.('ain.EnvTwoPhaseL + ainp.EnvTwoPhaseL-w);
agent   EnvTwoPhaseR    =  routp.('rout.aout.'aoutp.EnvTwoPhaseR
                                 + routp.EnvTwoPhaseR);
set     ENVTWOPHASESET  =  { rinp, routp, ainp, aoutp };




* EnvFourPhaseL, EnvFourPhaseR
* A valid four-phase environment for a micropipeline to operate in.
* The inputs (that is outputs from the micropipeline) allow quenching.
*
* Example:
*   agent Implementation = ( EnvFourPhaseL | Micropipeline | EnvFourPhaseR )
*                          \ ENVFOURPHASESET;
*
*       agents:                         EnvFourPhaseL, EnvFourPhaseR
*       observable actions:             rin,ain,rout,aout
```

```
*       interface to micropipeline:     rinp,ainp,routp,aoutp
*
* Note this is correctly the same as EnvTwoPhase.
*
agent   EnvFourPhaseL    =  rin.'rinp.EnvFourPhaseL-w;
agent   EnvFourPhaseL-w  =  ainp.('ain.EnvFourPhaseL + ainp.EnvFourPhaseL-w);
agent   EnvFourPhaseR    =  routp.('rout.aout.'aoutp.EnvFourPhaseR
                                    + routp.EnvFourPhaseR);
set     ENVFOURPHASESET  =  { rinp, routp, ainp, aoutp };
```

```
******************** CCS models of circuit elements **********************
```

```
* Xor
* XOR/XNOR gate
*       agent:  Xor
*       input:  in1, in2
*       output: out
*
agent       Xor    =  in1.Xor-e + in2.Xor-e;
agent       Xor-e  =  in1.Xor   + in2.Xor    + 'out.Xor;
```

```
* Toggle
* Toggle element
*       agent:  Toggle
*       input:  in
*       output: dot, blank
*
agent    Toggle   =  in.('dot.Toggle-b + in.Toggle);
agent    Toggle-b  =  in.('blank.Toggle + in.Toggle-b);
```

```
* Latch
* Pair of level sensitive latches, as used in the two-phase
* controller without toggle. When composing LatchL1 and
* LatchL2 they must be made to synchronise on latchpairsync.
*

* the real definition
*
agent Latch-EN-wait          =  latchpairsync.Latch-DIS-sync
                                    + in.Latch-e-wait;
agent Latch-EN               =  in.Latch-e
```

```
                                    + enable.Latch-EN-wait;
agent Latch-DIS-sync-wait    =  latchpairsync.Latch-EN
                                    + in.Latch-DIS-outsync-wait;
agent Latch-DIS-sync         =  enable.Latch-DIS-sync-wait
                                    + in.Latch-DIS-outsync;
agent Latch-DIS-outsync-wait =  latchpairsync.Latch-e
                                    + in.Latch-DIS-sync-wait;
agent Latch-DIS-outsync      =  enable.Latch-DIS-outsync-wait
                                    + in.Latch-DIS-sync;
agent Latch-e-wait           =  latchpairsync.Latch-DIS-outsync
                                    + in.Latch-EN-wait
                                    + 'out.Latch-EN-wait;
agent Latch-e                =  'out.Latch-EN
                                    + in.Latch-EN
                                    + enable.Latch-e-wait ;

* Latch 'l1' (the one which rin is connected to)
*       agent:  LatchL1
*       input:  in, enable
*       output: out
*       also: latchpairsync
*
agent LatchL1 = Latch-EN;

* Minimise to save one state
min(LatchL1, LatchL1);

* Latch 'l2' (the one which drives rout and ain)
*       agent:  LatchL2
*       input:  in, enable
*       output: out
*       also:   latchpairsync
*
agent LatchL2 = Latch-DIS-sync['latchpairsync/latchpairsync];

* Minimise to save one state
min(LatchL2, LatchL2);




* DataLatch
* Simulated multibit data latch (without data), used to produce
* an observable capture action. Note quenching is not needed as
* this is just a buffer.
*       agent:  DataLatch
*       input:  enable
*       output: done
*
agent  DataLatch  =  enable.capture.'done.enable.'done.DataLatch;
```

```
* Fork
* Fork in wire, allows either output order, and quenching
*       agent:  Fork
*       input:  in
*       output: out1, out2
*
agent     Fork     =  in.Fork-e;
agent     Fork-e   =  in.Fork    + 'out1.Fork-e2 + 'out2.Fork-e1;
agent     Fork-e1  =  'out1.Fork + in.Fork-e2;
agent     Fork-e2  =  'out2.Fork + in.Fork-e1;
```

```
********************** Muller C-elements **********************************
*
* Naming examples:
*
*       Cpbn
*
* A a C-element with one input which can only make the output
* go _Positive_, one input which is a 'normal' input (can make
* the output go _Both_ positive and negative), and one input
* which can only make the output go _Negative_.
*
* Likewise the inputs are named p1, p2, b1, n1 etc.
*
* If an input is _Inverted_ (with a bubble) then the C-element
* name would be like this (but the input names are unchanged):
*
*       Cpibn
*
* 'Internal' agent names do not have inverted inputs marked (as they
* are reused), and they have the current state of inputs and maybe
* also the output in the name. For example, for Cpibn, an agent used
* in its construction might be Cpbn-010-0, indicating p1=0, b1=1,
* n1=0 and out=0.
*
```

```
* Standard 2-input C-element
*       agent:   Cbb
*       input:   b1, b2
*       output:  out
*
agent    Cbb      =  b1.Cbb-b1 + b2.Cbb-b2;
agent    Cbb-b1   =  b1.Cbb    + b2.Cbb-e;
agent    Cbb-b2   =  b1.Cbb-e  + b2.Cbb;
agent    Cbb-e    =  b1.Cbb-b2 + b2.Cbb-b1 + 'out.Cbb;
```

```
* Standard 2-input C-element with bubble on one input
*       agent:  Cbib
*       input:  b1 (inverted), b2
*       output: out
agent   Cbib      = Cbb-b1;

* Minimise to save one state
min(Cbib, Cbib);

* Other asymmetric C-elements which normally follow below omitted
```

B CCS Model of the AMULET1 Latch Controller

```
* Include standard elements
*
input "elements.cwb";



* Build up the latch controller circuit.
* Interface to "Circuit" is rin,ain,rout,aout.
*

* C-element
agent   C1 = Cbib[c1/out,t1blank/b1,rin/b2];

* xor
agent     X1 = Xor[x1/out,c1/in1,aout/in2];

* datalatch (on output of xor)
agent     D1 = DataLatch[d1/done,x1/enable];

* toggle (on output of data latch)
agent     T1 = Toggle[t1dot/dot,t1blank/blank,d1/in];

* fork1 from output of toggle to give ain and rout
agent   F1 = Fork[ain/out1,rout/out2,t1dot/in];

* whole circuit
agent     Circuit = ( C1 | X1 | D1 | T1 | F1 )
\ {c1,x1,d1,t1dot,t1blank};

* rename ports on circuit to be compatible with environment
agent     Circuit'= Circuit[rinp/rin,routp/rout,
  ainp/ain,aoutp/aout];

* attach Circuit' to environment
agent     Implementation         = ( EnvTwoPhaseL | Circuit' | EnvTwoPhaseR )
 \ ENVTWOPHASESET;

* hopefully, this will be true
eq(SpecTwoPhase,Implementation);
```

C CCS Model of the Simplified Latch Controller

```
* Include standard elements
*
input "elements.cwb";



* Build up the latch controller circuit.
* Interface to "Circuit" is rin,ain,rout,aout.
*

* latch l1
agent   L1 = LatchPairL1[l1/out,f21/enable,rin/in];

* fork from output of l1 to XOR and l2
agent     F1 = Fork[f11/out1,f12/out2,l1/in];

* xor
agent     X1 = Xor[x1/out,f11/in1,aout/in2];

* datalatch (on output of xor)
agent     D1 = DataLatch[d1/done,x1/enable];

* fork2 from output of datalatch to l1 and l2
agent   F2 = Fork[f21/out1,f22/out2,d1/in];

* latch l2
agent   L2 = LatchPairL2[l2/out,f22/enable,f12/in];

* fork 3 from output of l2 to ain and rout
agent     F3 = Fork[ain/out1,rout/out2,l2/in];

* whole circuit
agent     Circuit = ( L1 | F1 | X1 | D1 | F2 | L2 | F3 )
\ {l1,f11,f12,x1,d1,f21,
   f22,l2,latchpairsync};

* rename ports on circuit to be compatible with environment
agent     Circuit'= Circuit[rinp/rin,routp/rout,
  ainp/ain,aoutp/aout];


* attach Circuit' to environment
agent     Implementation        = ( EnvTwoPhaseL | Circuit' | EnvTwoPhaseR )
                \ ENVTWOPHASESET;

* hopefully, this will be true
eq(SpecTwoPhase,Implementation);
```