Copyright 1998 IEEE. Published in the Proceedings of CSD'98, March 1998 Fukushima, Japan. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

Event-Driven Verification of Switch-Level Correctness Concerns [*]

Radu Negulescu
Department of Computer Science, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
radu@maveric.uwaterloo.ca

Abstract

We propose a technique for verification of MOS circuits, focusing on signal transitions (events) rather than signal levels. Diverse conditions, behaviors, and even delay assumptions are modeled as processes that can be coupled and compared to circuit specifications in a unified formalism. Verification is performed modularly and hierarchically by a BDD-based tool. We illustrate this technique on a self-timed RAM.

1 Introduction

Switch-level models, introduced in [Br87], offer a viable trade-off between accuracy and efficiency of the analysis of digital MOS circuits. Such models approximate an MOS transistor as an on/off switch, and simplify charge sharing and path strength calculations to the extreme. Nevertheless, such models can capture logical behavior that can be used further for gate level analysis, and can deal with specific switch-level concerns, such as floating states, collisions, and shorts.

In a *floating state*, a circuit wire is isolated from both ground and power supply. A *collision* occurs if a wire is connected to both ground and power supply through transistor channel networks of commensurate strengths. A *short* is similar to a collision, except that the connections to ground and power supply may be of different strengths. If any of these situations occur in stable states, we call them *persistent*; otherwise we call them *transient*. Floating states, collisions, and shorts do not necessarily constitute malfunctions, especially if they are transient; in fact, floating states and transient non-collision shorts are routinely put to use in certain design styles. Still, floating states and collisions may be undesirable in other designs, because the respective wire voltages decay to an unknown intermediate value; also, non-collision shorts may be undesirable because of undue power dissipation. For generality, our technique permits both to detect and to ignore such situations.

[*] This research was supported by grant no. OGP0000871 from the National Sciences and Engineering Research Council of Canada.

Switch-level analysis methods have been proposed in [Br87], [BS95], [KS95], [KSLP95], and [ZH92], among others. However, except for [BS95], these methods have focused only on steady-state or synchronous behavior. The methods in [Br87] and [KSLP95] extract the steady-state response of switch-level networks, characterizing signal relationships in stable states. The result can be used further in gate-level analysis, and [ZH92] and [KS95] also apply it to sequential, synchronous circuits that stabilize over a clock period. The framework for MOS circuit analysis in [BS95] can be used for various checks and can even incorporate timing information, but the subsequent timed analysis is only for reachable stable states and it requires the knowledge of bounds on individual component delays, thus postponing the use of delay constraints till after the transistors are sized.

The technique we propose here aims to complement the methods above and to provide additional flexibility. We do not extract the behavior of a switch-level network, but we perform computationally-intensive verifications of switch-level correctness concerns, as well as gate-level analysis. Our analysis is event-driven, in the sense that it focuses on signal transitions rather than signal levels and does not assume (nor forbid) synchrony. The event-driven approach permits to use certain processes that are not easy to cast in a Boolean framework. Notably, these processes can model several assumptions regarding the behavior of charge-retaining circuit nodes, diverse switch-level correctness concerns, and relative speeds of the components. As in [Ne97], these speed assumptions do not require knowledge of numerical bounds on component delays, thus, very importantly, the circuit can be analyzed before sizing its transistors and wires.

The same formalism is used across various levels of abstraction, and modular and hierarchical verification are fully supported. The verification and diagnosis are done by a tool based on binary decision diagrams (BDDs).

We tour the features of the technique by means of a case study, a self-timed RAM circuit from [NS95]. Previous verification of this circuit was reported in [NS95], but it did not address the switch-level correctness concerns, and it did not cover parts of the design where the delay constraints intervene.

2 The Formalism Used

Process spaces are a simple theory that unifies the treatment of several types of systems and correctness criteria, by abstracting the notion of 'execution.' Executions can be sequences of events, functions of time, etc., but, in the general theory, they are just elements of an arbitrary set $\mathcal{E}$. Process spaces were introduced in [Ne95]; also see the technical report for [Ne97].

A *process* over set $\mathcal{E}$ is a pair (X, Y) of subsets of $\mathcal{E}$ such that $X \cup Y = \mathcal{E}$. A process represents a contract between a device and its environment, from the device viewpoint. A process (X,Y) splits $\mathcal{E}$ into subsets $X \cap Y$, $\overline{X}$, and $\overline{Y}$, whose executions are called *goals*, *errors*, and *rejects*, respectively. Executions in $X \cap Y$ are legal for both the device and the environment. Executions in $\overline{X}$ represent errors of the device. Finally, executions in $\overline{Y}$ represent bad behavior on the part of the environment, and, as such, can be rejected by the device.

In this paper, we use only the following operations and conditions on processes:

- *Refinement* is a binary relation, written $p \sqsubseteq q$, meaning 'process q is a satisfactory substitute for process p.'
- *Product*, written $\times$, is a binary operation, yielding a process for a system of two devices operating 'jointly.'
- *Meet*, written $\sqcap$, is a binary operation, yielding a process that behaves either like p or like q, where the choice is made pessimistically for each execution.

Refinement is reflexive, transitive, and antisymmetric. Product is commutative, associative, and idempotent. Furthermore, for processes p, q, and r,

$$p \sqsubseteq q \;\Rightarrow\; p \times r \sqsubseteq q \times r.$$

As described in [Ne95] and [Ne97], these properties enable us to break a refinement verification problem into several layers of refinement of partial specifications, and each layer into several modules, and to verify only one module at a time instead of verifying the overall problem in one piece.

For the application we treat in this paper, we use finite sequences of events as executions, and we interpret them as 'complete operations' of a system rather than 'initial fragments' of the operation.. This instance of process spaces, with operations and properties, is closely related to the prefix-closed trace structure theory in [Di89], and also includes a deadlock-freedom concern as in [Ve94].

Manipulations of finite-word processes are implemented by a tool called FIREMAPS [Ne97] (for finitary and regular manipulation of processes and systems). The tool uses a BDD library [BRB90] which offers basic routines for manipulating large Boolean functions. The operations and conditions implemented in FIREMAPS include (but are not limited to) the process space operations and conditions mentioned above, and constructors of models for circuit components and constraints, described below.

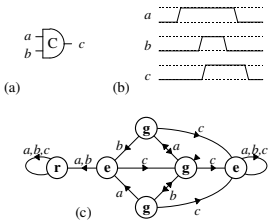

Figure 1: Example process: (a) the C-element symbol we use; (b) waveform; (c) hazard-intolerant process.

For an example of the models used in this paper, consider the C-element in Figure 1 (a). A C-element copies at the output the value of the inputs if the inputs have the same value, and keeps the output constant otherwise. Waveforms are represented by finite sequences of actions corresponding to signal transitions, such as *abcbac* for the waveform in Figure 1 (b). We sometimes indicate that a certain action represents a rising or falling transition, as in $a\uparrow\ b\uparrow\ c\uparrow\ b\downarrow\ a\downarrow\ c\downarrow$. Lower-level models are also given later in the paper.

If all signals start low, the C-element is represented by the process in Figure 1 (c), where **r**, **g**, and **e** stand for reject, goal, and error. Illegal output events lead to an error state that cannot be left, call it an *error trap*, and illegal input events lead to a reject state that cannot be left, call it a *reject trap*.

The state where *ab* leads is also marked as an error, making it illegal for the device to complete its operation by stopping there. This convention follows from the interpretation we chose for executions, and permits to capture not only illegal events, but also illegal stopping,

There are some variations of the CMOS cell models, because, in the presence of hazards, the behavior of a CMOS cell is not fully standardized. A hazard is a situation where an output transition is enabled and then disabled without being fired. For example, execution *abb* is a hazard for the C-element in Figure 1. We recast in process spaces several models described e.g. in [BS95], [Di89], and [La92]. Models that simply require the environment to avoid hazards, as in Figure 1 (c), are called *hazard-intolerant*. Models that ignore hazards,

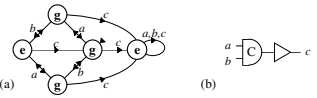

Figure 2: Alternative C-element models: (a) inertial model process; (b) example of 2-bounded model.

e.g. by making *abba* lead the C-element back to its initial state as in Figure 2 (a), are called *inertial*[1]. Other models, which can store up to k transitions after hazards, are called 'k-bounded pure delay' models; we are interested in even values for k, especially $k = 2$. An example is shown in Figure 2 (b), obtained by simply attaching an inertial buffer at the output of an inertial cell. Alternately, one might insert inversions instead of buffers, to modify the behavior at initialization.

We build process models for CMOS cells directly from Boolean descriptions of the unstable states. For instance, it suffices to specify that the C-element is unstable if and only if its signals satisfy $\neg(a \oplus b) \wedge (a \oplus c)$, considering that the output is c, where $\oplus$ is the Boolean exclusive-or.

In [Ne97], we also use processes to model relative delay assumptions of the following form[2]:

$$D\,(b_1\, b_2 \ldots\, b_n) \;>\; D\,(a_1\, a_2 \ldots\, a_m)$$

where a_1, ..., a_m, b_1, ..., b_n are events such that a_1 is the same as b_1, and the D's are the durations of the chains of events. Such a constraint, called a *chain constraint*, imposes that the b chain of events will not be completed before the a chain, unless one of the a or b actions involved occurs out of order and disrupts the chains.

FIREMAPS provides special operators for building a chain constraint process from the two chains of events involved and from an initialization sequence specifying the system states where the constraint applies.

Due to cases of deadlock and non-determinism, where the inequalities might not apply, we treat chain constraints as processes rather than linear inequalities. Reference to the entire chains, rather than just the end events, is significant. The reader is referred to the technical report of [Ne97] for a pertaining discussion. Additional case studies and implementation issues regarding chain constraints are presented in [NP98].

3 The Circuit Under Study

The circuit we analyzed is a self-timed RAM from

[1] Strictly speaking, the usual inertial models also include numerical bounds on gate delays; see e.g. [BS95].

[2] We are now using a simplified notation, but the constraints are the same as in [Ne97].

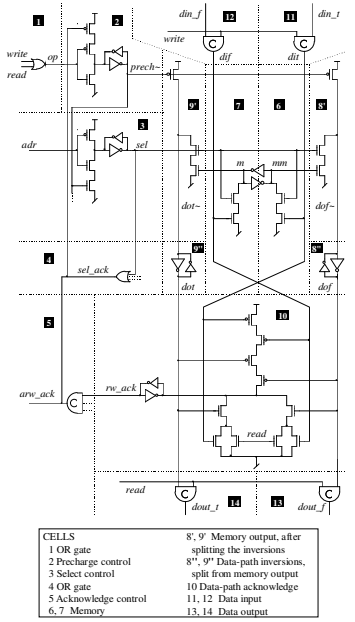

Figure 3. Circuit schematic after [NS95], with adjusted partition into cells.

[NS95], intended to operate as follows. Signals *read* and *write* are control signals; *adr* is used as a select signal; *dit* and *dif* are data inputs, for 'true' and 'false,' and they are mutually exclusive; *dot* and *dof* are data outputs, with a similar encoding; *arw_ack* is an acknowledge signal for read and write operations. Initially, all these signals are low. A read cycle starts with the environment raising *read* and *adr*; then, the circuit responds by raising *arw_ack* and either *dot* or *dof*; the environment lowers *read* and *adr*; the circuit lowers *arw_ack* and either *dot* or *dof*, completing the cycle. A write cycle starts with the environment raising *write*, *adr*, and either *dif* or *dit*; the circuit raises *arw_ack*; the environment lowers *write*, *adr*, and either *dif* or *dit*; finally, the circuit lowers *arw_ack*, completing the cycle. A formal specification for the circuit is given in Appendix 1.

The schematic of the circuit we analyze is given in Figure 3, after a figure from [NS95]. We have introduced some new signal labels. The ~ signs are part of the signal labels and stand for 'active low.'

For initial conditions, we assume signals *mm*, *dof~*, and *dot~* in Figure 3 are high, and all other labeled signals of Figure 3 are low. By taking *mm* to start high, we consider that a zero is initially stored in the memory cell. Symmetrical results would be obtained by assuming a one is stored initially.

Various problems may occur if the circuit is used immediately after power-up, before *m* and *mm* stabilize to opposite levels. However, in practice it is often assumed that a circuit is not used before it stabilizes in a desired state after power-up. One can easily check that the circuit under study will stabilize in the initial state or the symmetrical state described above if all its inputs are held low, as in the initial state of the formal specification. Thus, while noting the danger, we limit our analysis to the initial conditions where *m* is low and *mm* is high.

Apart from the overall specification and the low-level implementation, we use interface specifications for individual cells to divide the analysis problem into sub-problems. Following [NS95], let us represent cell interfaces by *production rules* [Ma90] of the form

$$\ll \textit{logical_condition_on_the_input_signals} \rightarrow \textit{output_signal_assignment} \gg .$$

For example, the OR gate with inputs *read* and *write* and output *op* can be represented by

$$\ll \textit{read} \vee \textit{write} \rightarrow \textit{op} := \textit{TRUE} \gg \quad \text{and}$$
$$\ll \neg \textit{read} \wedge \neg \textit{write} \rightarrow \textit{op} := \textit{FALSE} \gg .$$

If either *read* is high or *write* is high, the output *op* rises within a certain delay; if both *read* and write are low, the output falls after a certain delay. The delay is bounded, but the bounds are not known. A more concise representation of this OR gate is:

$$\ll \textit{TRUE} \rightarrow \textit{op} := \textit{read} \vee \textit{write} \gg .$$

Essentially, these rules amount to specifying the unstable states of the cells. As such, they are automatically translated into finite-state process models by our analysis tool, as described in Section 2. The interface specifications we used are listed in Appendix 2.

4 Switch-Level Correctness Concerns

We start our analysis by dividing an MOS transistor network into *channel-connected subnetworks* [Br87], defined by the absence of gate-drain and gate-source connections. The reason for this split is that gate capacitances usually introduce relatively large delays that need to be modeled separately. We refer to such large-load circuit nodes as *charge-retaining* nodes.

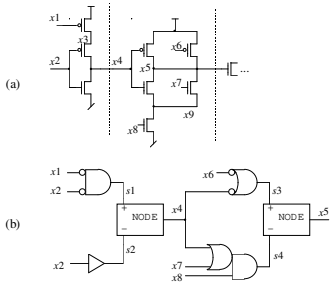

Figure 4: Modeling a switch-level network: (a) MOS transistor network split into channel-connected subnetworks; (b) replacement network.

Normally each channel-connected subnetwork contains an N-transistor network and a P-transistor network, called the *pull-down* and the *pull-up* networks. As was done in previous switch-level analysis techniques, we use Boolean variables to indicate the presence of conducting paths. We associate two such variables, call them *path signals*, to each charge-retaining node: one for a path to ground and one for a path to the power supply. For now, we assume the paths have equal strengths; we deal with differing strengths later.

For example, consider the network in Figure 4 (a). We introduce the following path signals:

*s*1: there is a conducting path from *x*4 to power supply;
*s*2: there is a conducting path from *x*4 to ground;
*s*3: there is a conducting path from *x*5 to power supply;
*s*4: there is a conducting path from *x*5 to ground.

The pull-down and pull-up networks, controlling the path signals, are modeled by processes which turn out to be similar to Boolean gates. This is not surprising, since such processes emulate the logical relationships between voltage signals and path signals. In Figure 4 (b), we actually use Boolean gate symbols for such processes.

In the following, we discuss the particular process models we used for the components of a *replacement network* like that in Figure 4 (b). These processes are our main mechanism for dealing with various switch-level correctness concerns and peculiarities of behavior. We also discuss certain modeling options that we later discard, but this does not mean that the same 'faux pas' have to be taken every the technique is applied; the purpose of the discarded options is to help explain and justify the options we consider valid.

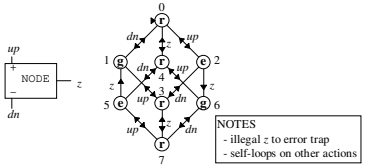

Figure 5. Node icon and possible node process.

The NODE in Figure 4 (b) represents charge retention in the presence of conducting paths. Such a component controls a voltage signal according to two path signals, representing conducting paths to ground and the power supply. In Figure 5, these signals are *z*, *up*, and *dn* respectively. In state 0, the initial state, there is no conducting path and the voltage is low. (In applications, we change the initial state as needed.) To help interpret the model, notice that: *z* is always high in states 4, 5, 6, 7, regardless of how the states were reached, and it is always low in 0, 1, 2, 3; *up* is high in 2, 3, 6, 7 and low elsewhere; *dn* is high in 1, 3, 5, 7 and low elsewhere.

To avoid clutter, we often omit some events from the state machines, indicating the omissions in 'notes' boxes. For more details regarding these notes, see Appendix 2.

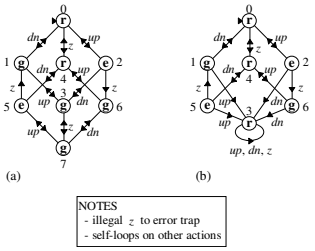

Figure 6. Additional alternatives for node processes: do we forbid collisions?

For the model in Figure 5, we forbid persistent collisions by taking states 3 and 7 in Figure 5 to be rejects. It is the responsibility of the environment to avoid stopping in such a state, where both path signals are high at the same time. Alternately, we could consider collisions to be legal, as in Figure 6 (a). In Figure 5, we allow transient collisions; for instance, execution *up z dn*

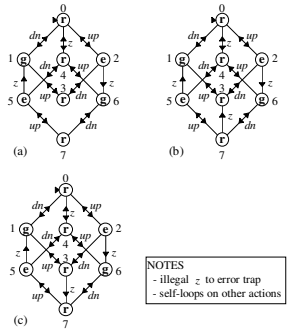

Figure 7. Alternate node processes: do we forbid dangling during collisions?

up z is a goal, despite passing through state 7. Another alternative would be to make transient collisions illegal by taking all executions that pass through a collision state to be rejects, as in Figure 6 (b); there, states 3 and 7 have been merged.

For the model in Figure 5, we assumed that, when both paths are conducting, we don't know whether the voltage is driven high, low, or close enough to a threshold level for the neighboring gates to sense small fluctuations as signal transitions. Accordingly, in Figure 5 we allow the node voltage to dangle (switch voltage arbitrarily) during a collision, following the cycle between states 3 and 7. Alternately, we could assume the node voltage constant during collisions, to model, for instance, that a feedback inverter pair introduces a hysteresis as in many of the cells in Figure 3. For that, we could take the *z* edges that leave states 3 or 7 to lead to an error-trap state instead, as in Figure 7 (a). Another alternative would be that the voltage might fall but never rise during a collision, for a case where thresholds of subsequent gates are higher than the intermediate level where the node voltage stabilizes during a collision. For that, we could allow a *z* edge from state 7 to state 3, but take the *z* edge that leaves state 3 to lead to an error-trap, as in Figure 7 (b). Symmetrically, we could consider instead that the voltage may rise but never fall during a collision, as in Figure 7 (c). Yet another alternative would be to consider that the voltage either never rises or never falls during a collision, but it may always undergo the opposite transition during a collision. For this, we could take the meet of the models in Figure 7 (b) and Figure 7 (c).

Symmetrical alternatives can be obtained for floating states, and the modifications can be combined to obtain a family of node models for diverse assumptions and correctness concerns. In this case study we settled for the model in Figure 5, which forbids persistent collisions and floating states, permits transient collisions and floating states, and accounts for dangling of the node voltage both while floating and during collisions.

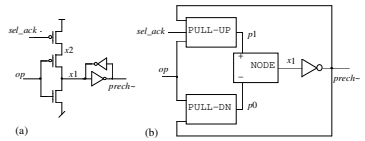

Figure 8. Modeling weak paths in the precharge control cell: (a) circuit and signals; (b) replacement network.

So far we have only considered paths of equal strengths. For differing path strengths, approximated by integers as in [Br87], the function realized by the Boolean model of a pull-down network is as follows:

$$
\begin{aligned}
&[\exists \textit{ a conducting path to ground of strength } n] \\
&\vee\ ([\exists \textit{ conducting path to ground of strength } n-1] \\
&\quad \wedge \neg [\exists \textit{ conducting path stronger than } n-1 \\
&\qquad \textit{leading to the power supply}]) \\
&\vee\ ([\exists \textit{ conducting path to ground of strength } n-2] \\
&\quad \wedge \neg [\exists \textit{ conducting path stronger than } n-2 \\
&\qquad \textit{leading to the power supply}]) \\
&\vee\ \ldots
\end{aligned}
$$

The model for a pull-up network is symmetrical.

For example, the precharge control cell from [NS95] shown in Figure 8 (a) with some new signal labels, is modeled by the network in Figure 8 (b), which includes two channel-connected subnetworks: one for the strong inverter and one for the rest of the cell. For the latter subnetwork, the pull-up and pull-down networks are modeled by complex Boolean gates, represented by the boxes in Figure 8 (b) that control the path signals $p1$ and $p0$. The function realized by PULL-UP is

$$p1 = (\neg\, sel_ack \wedge \neg\, op) \vee (\neg\, prech\sim \wedge \neg\, op)\ ,$$

where the first disjunct indicates the presence of a strong conducting path to the power supply, while the second indicates there is a weak conducting path to the power supply *and* no strong conducting path to ground. The function realized by PULL-DN is, similarly,

$$p0 \ = \ (op) \vee (prech\sim \wedge \neg\, (\neg\, sel_ack \wedge \neg\, op))\ .$$

For the strong inverter, we use an inertial model, as justified later. Now, we check refinement between the formal interface for the precharge control cell and the replacement network in Figure 8 (b). We obtain the following counter-example execution:

(1) $op\uparrow\ p0\uparrow\ x1\downarrow\ prech\sim\uparrow\ x1\uparrow\ prech\sim\downarrow\ op\downarrow\ p0\downarrow$

which wrongly indicates that *prech~* may dangle! This dangling is because a path to ground seems to start conducting at $p0\uparrow$ as the path to the power supply continues to conduct, as in the initial state, despite $op\uparrow$.

In reality, however, we would expect that *prech~* does not have time to charge and discharge before the path to the power supply is terminated, and that the transistors 'open' and 'close' much quicker than the voltage signals rise or fall. If so, execution (1) would be avoided, because, by the time $prech\sim\uparrow$ occurs, the path to the power supply will have been terminated by a $p1\downarrow$ event. To introduce this assumption, we modify the processes for the 'gates' that control path signals in the replacement network. We start from inertial processes, since paths are assumed to be formed and interrupted 'very quickly' after an input signal undergoes a transition. Then, we modify the process by taking the 'slower' transitions of the node voltage to lead to an error trap state from any unstable state of the replacement gate. We refer to the gate processes modified this way as *quickened models*.

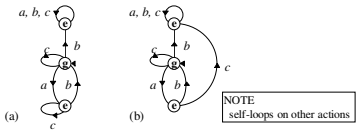

Figure 9. Obtaining a quickened model from an inertial model: (a) inertial model; (b) quickened model.

For example, the process in Figure 9 (a) represents an inertial buffer with input a and output b. Action c is external to the buffer and thus c events do not affect the state of the buffer. Assuming c transitions are slow, we take in Figure 9 (b) the c events from the unstable state to lead to an error-trap state, making it the responsibility of the device to avoid such events. How can the buffer physically avoid these external events? By being 'quick' to leave its unstable state.

Note that none of these assumptions is intrinsic to our technique, and other assumptions might be used to fit a particular application. For example, the problem in execution (1) would also be avoided by an assumption that the node does not switch voltage while floating or during collisions, instead of using quickened models. This alternate assumption can be justified, say, if the hysteresis of the output inverter pairs keeps the node voltages constant during collisions, and if the weak inverters ensure that the respective nodes are never floating. However, we use quickened models because they provide local compatibility with [Br87] and other switch-level analysis methods: if path signals switch so quickly, it suffices to consider their steady-state behavior, which can be extracted, say, by [Br87] from a transistor netlist. Note, nevertheless, that the transient behavior of the nodes is not hindered by this assumption, which refers to the N and P subnetworks but not to entire cells.

Finally, we find that refinement holds between the hazard-intolerant interface of the precharge control cell and the new replacement network. For the reasons discussed above, we settled for quickened inertial models for the gates and a node of the type in Figure 5. The hazard-intolerant model is our standard model for the interface, and assumes the cell is operated without hazards. (We verify later that the cell is indeed operated without hazards.) It follows, under these assumptions, that there are no persistent floating states and no persistent collisions for this cell.

The inertial model for the strong inverter is justified by checking refinement of this model by a replacement network for the inverter, consisting of quickened inertial models and a node of the type in Figure 5.

After a refinement check fails for the select control cell under the currently used models established above, we obtain the following counter-example execution:

(2) $adr\uparrow$

Execution (2) indicates a false deadlock: the original interface from [NS95] expects a $sel\uparrow$ to occur in response to $adr\uparrow$, whereas the actual CMOS gate does not raise its output signal, since *prech~* is still low. This points to a flaw in that interface, which should be instead:

$$\ll adr \wedge prech\sim \ \rightarrow\ sel := TRUE \gg$$
$$\ll \neg\, adr \ \rightarrow\ sel := FALSE \gg\ .$$

For this new cell interface, refinement holds.

The *prech~* signal was not shown in the schematic in [NS95]; however, from the precharge control cell interface in [NS95], one deduces that *prech~* is the output of the precharge control cell, active low. Regardless of how *prech~* is interpreted, the original cell interfaces do not correspond to the transistor schematic.

The memory cell circuit splits into two channel-connected subnetworks. The part that drives m is shown in Figure 10; the weak transistors are drawn smaller. We find that refinement holds between the memory cell interface that drives m and the network in Figure 10, and between the memory cell interface that drives *mm* and the remaining part of the memory cell (for models as above).

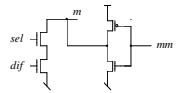

Figure 10. Part of the memory cell.

The memory output cells are symmetrical and we only discuss here the cell that drives *dot*. We obtain the following execution as a counter-example to refinement:

(3) $sel\uparrow\ m\uparrow\ p8\uparrow$

where $p8$ is the path signal of the pull-down subnetwork, and $p8\uparrow$ indicates that a path is formed to ground. Since initially *prech~* is low, the pull-up network is conducting too, and execution (3) leaves the cell in a persistent collision. The environment and the rest of the circuit should avoid this danger by interrupting the path to the power supply before the pull-down network starts conducting, and we check later whether or not they always avoid the danger.

For the data-path acknowledge cell, verification was straightforward and refinement does indeed hold between the cell interface and the replacement network.

5 Relative Delay Assumptions

As the circuit is not completely speed-independent, we use chain constraints to model its relative delay assumptions. Finding the chain constraints typically requires several verifications; to speed up this procedure, we use gate-level cell interfaces instead of the replacement networks of the switch-level cells. In the final verification, after finding candidate chain constraints, we use the replacement networks of the memory output cells, and the interface specifications of all other cells, since the replacement networks of all other cells did refine their interface specifications.

Using hazard-intolerant models for all cells and the corrected interface for the select control cell, we obtain the following counter-example execution:

(4) $write\uparrow\ din_t\uparrow\ dit\uparrow\ adr\uparrow\ op\uparrow\ prech\sim\uparrow\ sel\uparrow\ mm\downarrow\ m\uparrow\ dot\uparrow\ sel_ack\uparrow\ rw_ack\uparrow\ arw_ack\uparrow$

The $mm\downarrow$ event causes a hazard for the memory output cell that drives *dof*, since $dof\uparrow$ was enabled after $sel\uparrow$, but $dof\uparrow$ did not occur and was disabled by $mm\downarrow$. The presence of this hazard is mentioned in [NS95], where it is indicated that this hazard is tolerated for the sake of efficiency. Thus, in the following, we use inertial models for the parts of the memory output cells that drive *dof~* and *dot~*, followed by hazard-intolerant models for the inverter cells that drive *dof* and *dot*. This way, we model

the storage of a logical value on the capacitances of *dot~* and *dof~*, since the internal data bus normally has a large capacitance. This amounts to replacing the hazard-intolerant model of the memory output by a 2-bounded model. The new interfaces are as follows.

Memory output cells:

$\ll sel \wedge m \rightarrow dot\sim := FALSE \gg$
$\ll \neg prech\sim \rightarrow dot\sim := TRUE \gg$

$\ll sel \wedge mm \rightarrow dof\sim := FALSE \gg$
$\ll \neg prech\sim \rightarrow dof\sim := TRUE \gg$

Data-path inversions:

$\ll TRUE \rightarrow dot := \neg\, dot\sim \gg$
$\ll TRUE \rightarrow dof := \neg\, dof\sim \gg$

With the new interfaces, refinement still doesn't hold. We obtained the following counter-example execution:

(5) *write*↑ *op*↑ *prech~*↑ *adr*↑ *sel*↑ *din_t*↑ *dit*↑ *dof~*↓ *mm*↓ *m*↑ *dot~*↓ *dot*↑ *sel_ack*↑ *rw_ack*↑ *arw_ack*↑ *adr*↓ *write*↓ *op*↓ *sel*↓ *sel_ack*↓ *prech~*↓ *dof~*↑ *dot~*↑ *dot*↓

which shows a hazard at the data-path inversion cell that drives *dof*. The hazard is caused by *dof~*↑, which disables a *dof*↓ transition that was enabled by *dof~*↓ but never occurred.

The hazard of (5) was mentioned in [NS95], which indicated that it will not actually occur because 'the inverter is the fastest gate in a CMOS circuit.' Accordingly, we can assume that the data-path inverters will switch before *arw_ack*↑ is issued. This assumption is too general; we actually impose some weaker chain constraints that are easier to model and that follow from the assumption above:

(6) D (*dof~*↓ *rw_ack*↑ *arw_ack*↑) > D (*dof~*↓ *dof*↑)
(7) D (*dot~*↓ *rw_ack*↑ *arw_ack*↑) > D (*dot~*↓ *dot*↑)

Under these chain constraints and models for the memory output cell, refinement holds at the gate level.

To obtain speed-independence, [NS95] proposes an alternate design for the memory cell and the memory output cells, reproduced here in Figure 11 and represented by the interfaces:

$\ll TRUE \rightarrow mm := \neg\, m \wedge \neg\, (dif \wedge write) \gg$
$\ll TRUE \rightarrow m := \neg\, mm \wedge \neg\, (dit \wedge write) \gg$
$\ll sel \wedge dif \wedge mm \rightarrow dof\sim := FALSE \gg$
$\ll \neg prech\sim \rightarrow dof\sim := TRUE \gg$
$\ll sel \wedge dit \wedge m \rightarrow dot\sim := FALSE \gg$
$\ll \neg prech\sim \rightarrow dot\sim := TRUE \gg$

In Figure 11, transistors *N*1, *N*2, *N*3 and the two inverters form a single channel-connected network.

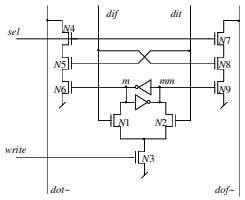

Figure 11. Speed-independent RAM design from [NS95], plus transistor labels for reference.

However, signals *dit* and *dif* are mutually exclusive; thus, no conducting path can be formed through *both* transistors *N*1 and *N*2. Following this observation, we split the memory cell into two cells, to simplify the analysis. One of the new cells is as shown in Figure 10, but with new signals; the other new cell is symmetrical.

At this point we check refinement using the alternate cell interfaces and no constraints. We obtain the following counter-example execution:

(8) *read*↑ *adr*↑ *op*↑ *prech~*↑ *sel*↑ *sel_ack*↑

which indicates that the circuit may deadlock in the middle of a read cycle, while the specification expects data output and acknowledge events. The problem is that one of the memory output cells should have discharged *dof~* after *prech~*↑ and *sel*↑, but its pull-down network was not conducting. The schematic of the alternate, speed-independent circuit from [NS95] is thus wrong.

The text in [NS95], however, does hint that the newly added transistors should be enabled during a read operation, i.e., *N*5 and *N*8 are enabled whenever *read* is high. On the other hand, in the given schematic, this hint implies that a conducting path will be formed through *N*1 and *N*2 whenever *read* is high, thus shorting the inverter outputs and endangering the stored data. A way to fix this problem is to control *N*1, *N*2, *N*5, and *N*8 by *dif*, *dit*, *dit* ∨ *read*, and *dif* ∨ *read*, respectively, but this requires to disconnect the gates of *N*1 and *N*8, and of *N*2 and *N*5. We verified that refinement holds for the modified schematic.

Since the memory output cells do not refine their interfaces, we re-do the verification using their replacement networks instead of their formalized interfaces. Refinement does hold between the overall circuit specification and the product of processes for the following items: hazard-intolerant gate-level interface specifications for all cells except the memory output cells, the chain constraint processes for assumptions (6) and (7), the quickened models of the pull-down and pull-up networks of the memory output cells, the inertial models of the strong inverters of the memory output cells, and the nodes of the memory output cells (of the type in Figure 5). (The reasons for using these particular models were discussed above.) Since we had checked the cell interfaces against the switch-level networks, it follows that the overall circuit specification is refined by the product of the switch-level replacement networks.

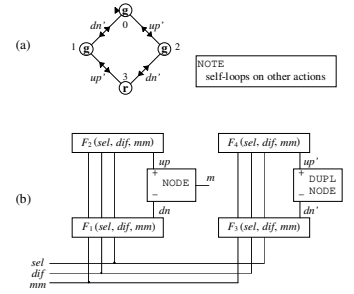

Figure 12. Model for detecting persistent shorts in the channel-connected subnetwork in Figure 10: (a) duplicate node; (b) short-detecting replacement network.

In particular, it follows that none of the node models stabilizes in a reject state. Thus, there are no persistent collisions and no persistent floating states under the assumptions discussed above.

6 Persistent Shorts

So far we have ignored persistent shorts except for persistent collisions. To forbid persistent shorts altogether, we introduce imaginary cells that duplicate the replacement cells. The genuine replacement cells are also kept in the network. The duplicate cells use simplified nodes and modified path strength schemes. The process for a duplicate node is shown in Figure 12 (a): it does not control a voltage signal, but it forbids persistent collisions by declaring collisions to be rejects. A duplicate cell permits to detect shorts as collisions at the duplicate node by declaring all paths to be of equal strength; for the pull-down network of such a cell, the path signal is thus:

[∃ *a conducting path to ground of strength n*]
∨ [∃ *a conducting path to ground of strength n* − 1]
∨ [∃ *a conducting path to ground of strength n* − 2]
∨ ...

For the memory cell in Figure 10, the short-detecting replacement network in shown in Figure 12 (b). The Boolean functions the path signals are, respectively,

$F_1(sel, m, mm) = (sel \wedge dif) \vee mm$
$F_2(sel, m, mm) = \neg((sel \wedge dif) \vee mm)$
$\quad \wedge \neg mm = \neg((sel \wedge dif) \vee mm)$
$F_3(sel, m, mm) = (sel \wedge dif) \vee mm$
$F_4(sel, m, mm) = \neg mm$

Unfortunately, this short-detecting replacement network does not refine its cell interface: if for instance *sel* and *dif* were kept high and *mm* kept down, a persistent short would occur. But may such situations happen in the context of the circuit and specified environment? We re-run the verification with respect to the overall circuit specification, after including the duplicates of the memory output cells, and we find that refinement holds. There is no need to include duplicates for the other cells, because their weak paths are only through the weak feedback inverters, and, due to the strong inverters, they will agree with the strong paths in any stable state. Thus, persistent shorts at these other cells can only be through strong paths, and would have been detected as persistent collisions, which, we have already verified, are absent. It follows there are no persistent shorts either, under the assumptions discussed.

7 Concluding Remarks

Switch-level analysis proved to be an interesting exercise for event-driven modeling. Our main mechanism for handling diverse switch-level correctness concerns and behavior assumptions is by the family of models for charge-retaining nodes in Section 4; these assumptions and conditions can be strengthened or weakened in several ways. To justify our choices, we also discussed certain modeling options that are not usable. A lucrative direction for further work may be to provide a guide to which models apply for other MOS design styles.

One notable assumption was that replacement gates are much quicker than the transitions of an external signal. The quickened models in Section 4 incorporate this assumption, making it the responsibility of the replacement gates to 'avoid,' through timing, these external transitions while the gates are unstable.

These modeling problems were not trivial. Notice for instance that the quickened models introduce control over actions that are external to their devices, and chain constraint processes bring additional control over actions that are controlled by gate processes. In the analysis, we often compute product and decide refinement using such models, and we use the algebraic properties for hierarchical and modular verification, which, in process

spaces, do apply for these unusual connectivity cases too. Due to abstract executions and absence of connectivity restrictions, we expect process spaces will provide sufficient flexibility for many such niche applications.

In our case study the cells were rather small, and we found it easier to feed in the Boolean functions of channel-connected N and P subnetworks than to describe each transistor individually. If needed, one can use the specialized method of [Br87] to extract the behavior of the N and P subnetworks from transistor netlists, then input the results into our technique. Alternately, our technique can be extended to work directly from the transistor netlists, by using a separate process for each transistor. However, we expect our technique to find sufficient applications to circuits that have small but numerous cells, as in the case study, and we leave for future work the connections to [Br87] and the other extensions mentioned above.

The circuit in [NS95] was verified previously, but only for a stability protocol at the gate level, and excluding the speed-dependent part; [NS95] reports almost 1/2 hour on a standard workstation for these verifications. Here, a session spanning the verifications reported in this paper took just over 3 minutes (182 s) on a Sparc 5/110, reaching almost 10^{11} states during the largest task (the test for persistent shorts in Section 6). A separate gate-level verification session, covering hazard-freedom, deadlock-freedom, etc., took only 32 s here.

We found some minor flaws in the original cell interfaces (execution (2)) and in the schematic of an alternate circuit from [NS95] (execution (8)). According to [St97], the cell interface flaw from [NS95] was not present in the files that were originally verified, while the alternate memory design should be used with different control signals, hinted in [NS95]; for more details, see Section 5 above. Anyway, the case study shows that our technique could detect these problems. As for the main circuit from [NS95] (not their cell interfaces or alternate circuit), under the initial stable state assumption in Section 3, the switch-level behavior assumptions in Section 4, and the delay constraints in Section 5, it passed our verifications.

Acknowledgments

The author is grateful to J. A. Brzozowski for his comments and advice, including the mention of previous results, to C.-J. H. Seger for bringing previous work to our attention, to L. S. Nielsen and J. Staunstrup for their feedback, and to C. E. Molnar for the opportunity to apply and push forward the analysis technique of [Ne97] during a summer internship at Sun Microsystems Laboratories. The simplification of the chain constraint notation was suggested by Ad Peeters.

Appendix 1 Circuit Specification

Following informal descriptions from [NS95], we built a state machine that represents the intended behavior of the circuit as observed at the interface with the outside world. This specification is given in Figure 13.

The notes at the bottom of Figure 13 mention certain transitions that we have omitted to draw. A state machine for a process space specification should be a complete automaton, i.e., for every state and every action there should be an edge leaving that state and labeled with that action. To avoid clutter, we have not drawn in Figure 13 illegal transitions, that either the environment or the circuit should avoid. We have also omitted to draw external or internal actions, which are not 'seen' at the circuit-environment interface and only generate self-loops at every state of the process.

Appendix 2 Formal Cell Interfaces

Read/write OR gate:

$$\ll TRUE \rightarrow op := read \vee write \gg$$

Precharge control cell:

$$\ll op \rightarrow prech\sim := TRUE \gg$$
$$\ll \neg\, op \wedge \neg\, sel_ack \rightarrow prech\sim := FALSE \gg$$

Select control cell (as given in [NS95]; we change it in the analysis):

$$\ll adr \wedge \neg\, prech\sim \rightarrow sel := TRUE \gg$$
$$\ll \neg\, adr \rightarrow sel := FALSE \gg$$

Select OR gate (just a wire if there is only one cell):

$$\ll TRUE \rightarrow sel_ack := sel \gg$$

Acknowledge control cell (where <-> denotes logical equivalence, a.k.a. XNOR):

$$\ll rw_ack = sel_ack \rightarrow arw_ack := rw_ack \gg$$

Memory cell (as given in [NS95]; we change it below):

$$\ll sel \wedge (dit \vee dif) \rightarrow m, mm := dit, dif \gg$$

The original memory cell interface assumes that, in a write operation, *m* and *mm* are assigned simultaneously. In the circuit, however, these assignments are causally related and must occur one after the other. To follow more closely the implementation, we split the cell interface into two parts, one that drives *m* and one that drives *mm*, as follows.

$$\ll TRUE \rightarrow m := \neg\, (mm \vee (dif \wedge sel)) \gg$$
$$\ll TRUE \rightarrow mm := \neg\, (m \vee (dit \wedge sel)) \gg$$

Memory output cells (changed in the analysis):

$$\ll sel \wedge m \rightarrow dot := TRUE \gg$$
$$\ll sel \wedge mm \rightarrow dof := TRUE \gg$$

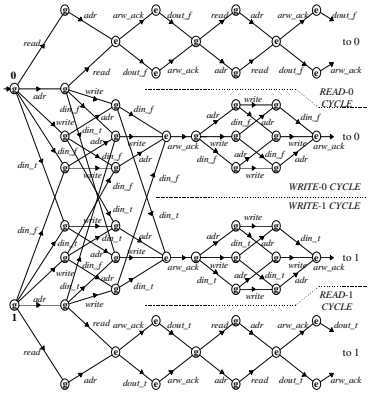

Figure 13. Circuit specification.

$$\ll \neg\, prech\sim \rightarrow dot := FALSE \gg$$
$$\ll \neg\, prech\sim \rightarrow dof := FALSE \gg$$

Data-path acknowledge cell:

$$\ll ((dit \vee read) \wedge dot) \vee ((dif \vee read) \wedge dof) \rightarrow rw_ack := TRUE \gg$$
$$\ll \neg\, (dit \vee dif \vee dot \vee dof) \rightarrow rw_ack := FALSE \gg$$

Data-path input cells:

$$\ll write = din_f \rightarrow dif := din_f \gg$$
$$\ll write = din_t \rightarrow dit := din_t \gg$$

Data-path output cells:

$$\ll read = dof \rightarrow dout_f := dof \gg$$
$$\ll read = dot \rightarrow dout_t := dot \gg$$

References

[BRB90] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, June 1990.

[Br87] R.E. Bryant. Boolean Analysis of MOS Circuits. In *IEEE Transactions on Computer-Aided Design*, 4: 634--649, July 1987.

[BS95] J.A. Brzozowski, C.J-H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[Di89] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1989.

[KS95] T. Kam, P.A. Subrahmanyam. Comparing Layouts with HDL Models: A Formal Verification Technique. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4: 503-509, April 1995.

[KSLP95] A. Kuehlmann, A. Srinivasan, D.P. LaPotin. Verity—A formal verification program for custom CMOS circuits. In *IBM Journal of Research and Development*, 1/2: 149-165, March 1995.

[La92] L. Lavagno. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits*. PhD thesis, University of California at Berkeley, 1992.

[Ma90] A. J. Martin. Programming in VLSI: from communicating processes to delay-insensitive circuits. In C.A.R. Hoare, ed., *Developments in Concurrency and Communication*, UT Year of Programming Series, pp. 1-64, Addison-Wesley, 1990.

[Ne95] R. Negulescu. Process spaces. Technical Report CS-95-48, Department of Computer Science, University of Waterloo, December 1995. `http://maveric.uwaterloo.ca/~radu/ps.html`

[Ne97] R. Negulescu. A technique for finding and verifying speed-dependences in digital circuits. In *Proceedings of the 1997 ACM / IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems* (TAU'97), December 1997 (to appear). A technical report is available from: `http://maveric.uwaterloo.ca/~radu/ps.html`

[NP98] R. Negulescu, A.M.G. Peeters. Verification of speed-dependences in single-rail handshake circuits. In *Proceedings of the Fourth International Symposium for Advanced Research in Asynchronous Circuits and Systems*, San Diego, California, USA, March 1998.

[NS95] L.S. Nielsen, J. Staunstrup. Design and verification of a self-timed RAM. In *Proceedings of the IFIP International Conference on Very Large Scale Integration 1995*, Chiba, Japan, 1995.

[St97] J. Staunstrup. Personal communication. 1997.

[Ve94] T. Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.

[ZH92] C. Zhou and C.A.R. Hoare. A Model for Synchronous Switching Circuits and its Theory of Correctness. In *Formal Methods in System Design*, 1:7-28, 1992.