Formal Aspects of Computing (1997) 3: 1–000

A Comparison of Simulation Techniques and Algebraic Techniques for Verifying Concurrent Systems[1]

Nancy Lynch and Roberto Segala

MIT- Laboratory for Computer Science, 545 Technology Sq., Cambridhge, MA 02139, USA

Keywords: I/O automata; Process algebras; Simulation method; Verification

Abstract. Simulation-based assertional techniques and process algebraic techniques are two of the major methods that have been proposed for the verification of concurrent and distributed systems. It is shown how each of these techniques can be applied to the task of verifying systems described as input/output automata; both safety and liveness properties are considered. A small but typical circuit is verified in both of these ways, first using forward simulations, an execution correspondence lemma, and a simple fairness argument, and second using deductions within the process algebra DIOA for I/O automata. An extended evaluation and comparison of the two methods is given.

1. Introduction

Simulation-based assertional techniques and process algebraic techniques are two of the major methods that have been proposed for the verification of concurrent and distributed systems. Although the two methods are used for the same task, the proofs that are carried out in the two styles seem to be quite different. Indeed, the two methods have been developed by largely disjoint research communities, using different semantic models. The literature contains many examples of proofs using the two methods: some typical examples of simulation proofs appear in [LyT87, SLL93a, SLL93b], while examples of algebraic proofs appear in [Bae90, Jos92, OrP92].

[1] Supported by NSF grant CCR-89-15206, by DARPA contracts N00014-89-J-1988 and N00014-92-J-4033, and by ONR contract N00014-91-J-1046.

Correspondence and offprint requests to: Nancy Lynch and Roberto Segala

In this paper, we unify, evaluate and compare the simulation-based and process algebraic verification techniques in terms of the Input/Output automaton (I/O automaton) model of Lynch and Tuttle [LyT87]. This framework has been used extensively for the verification of complex algorithms and pieces of distributed systems [WLL88, LaS92, LyP92, SLL93b], and has already been given a process algebraic characterization [Vaa91, Seg92, DnS92]. We show how each of these techniques can be applied to the common task of verifying both safety and liveness properties of systems described as I/O automata. We then use each technique to verify a small but typical delay insensitive circuit taken from [Jos92]: a Muller C element [MuB59] implemented in terms of a majority element and a wire. Both the implementation and the specification are described as I/O automata, and the verification consists of showing that the *fair preorder* relation (i.e., fair trace inclusion) holds between the implementation and the specification automata.

The two proofs proceed very differently. First, the simulation proof uses a forward simulation [LyV91] from the implementation to the specification, then invokes an execution correspondence lemma [GSSL93] to obtain a correspondence between executions of the implementation and the specification. Then a simple argument about fairness is made, based on the correspondence between executions; this fairness argument uses the convenient notion of a *forcing condition* for an I/O automaton fairness class. The fairness argument could easily be formalized using a temporal logic of states and actions [Sta84, SLL93b], although we do not do this in this paper.

The algebraic proof uses deductions within the process algebra DIOA [Seg92] for I/O automata. This process algebra contains a collection of axioms (i.e., sound proof rules) asserting that the *quiescent preorder* relation holds for a pair of I/O automata. The quiescent preorder is defined in [Vaa91] and consists of trace inclusion and quiescent trace inclusion. It is an approximation, based on finite traces only, of the fair preorder. The reason for the use of the quiescent preorder rather than the fair preorder is that quiescence fits nicely into a process algebraic theory containing recursion whereas fairness does not. We state conditions (proved in [Seg93]) giving some circumstances under which the quiescent preorder is equivalent to the fair preorder. Since these circumstances hold in our example, the DIOA deductions that prove quiescent trace inclusion are also sufficient to prove the needed fair trace inclusion.

We emphasize that our two proofs are constructed to prove exactly the same theorem. To make this clear we first give a "neutral" description of the verification problem in terms of I/O automata. Then we describe and verify the same problem in terms of an assertional representation of I/O automata and in terms of DIOA expressions, using simulation and algebraic techniques, respectively. We show formally that the two proofs are both solving the problem given in the "neutral" description. This last step is essential in order to ensure sure that, although we are using different formalisms, we are actually solving the same problem.

We then give an extended comparison of the two verification methods, based on our experiences in carrying out this research and on our other experiences with related examples. Our comparisons consider the correspondence between the two methods, their ability to model fairness, the style of their representation of system components, their suitability for mechanization, and the byproducts yielded by the proofs.

The rest of the paper is organized as follows. Section 2 contains a brief description of the I/O automaton model. Section 3 contains a formal statement of the circuit problem to be solved, i.e., showing that the fair preorder relation holds between a particular implementation and a Muller C element specification. Section 4 contains the verification using the simulation method. Section 5 contains the verification using process algebra. Section 6 contains an extended comparison between the two methods; Section 7 contains some additional conclusions.

2. The Input/Output Automaton Model

We begin with a brief review of the I/O automaton model, which will be used as the basis of the rest of the work in this paper. For a complete account, we refer the reader to [LyT87].

Definition 2.1. Given an alphabet A, let A^* be the set of finite length sequences made of elements of A and let A^ω be the set of infinite length sequences made of elements of A. Finally, let $A^* \cup A^\omega$ be denoted by A^∞.

Definition 2.2. An *I/O automaton* A consists of five components:

- a set $states(A)$ of states.
- a nonempty set $start(A) \subseteq states(A)$ of start states.
- an action signature $sig(A) = (in(A), out(A), int(A))$ where $in(A)$, $out(A)$ and $int(A)$ are disjoint sets of input, output and internal actions, respectively. We denote with $ext(A)$ the set $in(A) \cup out(A)$ of *external* actions, and by $local(A)$ the set $out(A) \cup int(A)$ of *locally controlled* actions. We denote by $acts(A)$ the set $ext(A) \cup int(A)$ of actions. We call $(in(A), out(A), \emptyset)$ the external action signature of A.
- a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ with the property that for each state q and each input action a there is a step from q with action a. We say that A is input enabled.
- A partition $part(A)$ of $local(A)$.

A transition $(q, a, q') \in steps(A)$ is also denoted by $q \xrightarrow{a} q'$. We extend the notion of transition to finite sequences of symbols by saying that

$$q \xrightarrow{a_1 \cdots a_n} q' \text{ iff } \exists_{q_0,\ldots,q_n}\ q_0 = q,\ q_n = q', \text{ and } q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n.$$

Similarly, for infinite sequences, we write

$$q \xrightarrow{a_1 a_2 \cdots} \text{ iff } \exists_{(q_i)_{i \in N}}\ q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots$$

Two derived transition relations, abstracting from internal computations, are

$$q \overset{a}{\Longrightarrow} q' \text{ iff } \exists_{s_1, s_2 \in int^*(A)}\ q \xrightarrow{s_1 a s_2} q',$$

$$q \overset{a}{\Longrightarrow} q' \text{ iff } \exists_{s_1 \in int^*(A)}\ q \xrightarrow{s_1 a} q'.$$

The last two transition relations can be extended to finite and infinite sequences of actions in the same way as for $steps(A)$.

Definition 2.3. An *execution fragment* of an I/O automaton A is a (finite or infinite) sequence of alternate states and actions starting with a state and, if the execution fragment is finite, ending in a state

$\alpha = q_0 a_1 q_1 a_2 q_2 \cdots$

where each $(q_i, a_{i+1}, q_{i+1}) \in \mathit{steps}(A)$. We denote by $\mathit{frag}^*(A), \mathit{frag}^\omega(A)$ and $\mathit{frag}(A)$ the sets of finite, infinite and all execution fragments of A, respectively. An *execution* is an execution fragment whose first state is a start state. We denote by $\mathit{exec}^*(A)$, $\mathit{exec}^\omega(A)$ and $\mathit{exec}(A)$ the sets of finite, infinite and all execution of A, respectively.

The *trace* of an execution fragment α of an I/O automaton A, denoted by $\mathit{trace}_A(\alpha)$, or just $\mathit{trace}(\alpha)$ when A is clear, is the list obtained by projecting α onto the set of external actions of A, i.e., $\mathit{trace}(\alpha) = \alpha \lceil \mathit{ext}(A)$.[2] We say that β is a *trace* of an I/O automaton A if there exists an execution α of A with $\mathit{trace}(\alpha) = \beta$. We denote by $\mathit{traces}^*(A), \mathit{traces}^\omega(A)$ and $\mathit{traces}(A)$ the sets of finite, infinite and all traces of A, respectively.

A key feature of the I/O automaton model is that the behavior of I/O automata is observed through their fair executions, i.e., those executions in which each "subcomponent" which is continuously willing to perform some of its locally controlled actions will eventually do so.

Definition 2.4. A *fair execution fragment* of an I/O automaton A is an execution fragment $\alpha \in \mathit{frag}(A)$ such that for all $X \in \mathit{part}(A)$

- If α is finite then no action of X is enabled from the final state of α.
- If α is infinite then either actions from X appear infinitely often in α or states from which no action of X is enabled appear infinitely often in α.

A *fair execution* is a fair execution fragment whose first state is a start state. A *fair trace* is the trace of a fair execution. We denote the set of fair traces of an I/O automaton A by $\mathit{ftraces}(A)$.

Now we can define the usual preorder relation for I/O automata.

Definition 2.5. Given two I/O automata A and B with the same external action signature, the *fair preorder* is defined as

$A \sqsubseteq_F B$ iff $\mathit{ftraces}(A) \subseteq \mathit{ftraces}(B)$.

The fair preorder is the relation that is used to model implementation in the I/O automaton model. Since input enabling ensures that any implementation must accept any external stimulus at any time, this preorder ensures that the implementation must contain a "rich" set of traces – enough to describe responses to any possible input pattern. Fairness ensures that the correctness of a solution is judged only on the basis of those behaviors in which the system is actually given the chance to make progress. Note that this preorder ensures that the implementation must provide output whenever the specification must do so.

Three main operators are defined on I/O automata: hiding, renaming and parallel composition.

Definition 2.6. Given an I/O automaton $A = (Q, Q_0, S, t, P)$ and a set of actions I such that $I \cap \mathit{in}(A) = \emptyset$, we define $\mathit{Hide}_I(A)$ to be the I/O automaton (Q, Q_0, S', t, P) where S' differs from S in that

[2] Our definition of *trace* coincides with the usual definition of *behavior* for I/O automata. We have changed the terminology in the interests of consistency with the usual notation of process algebra.

- $out(Hide_I(A)) = out(A) \backslash I$, and
- $int(Hide_I(A)) = int(A) \cup (acts(A) \cap I)$.

The hiding operator transforms external actions into internal ones, i.e., it hides some locally controlled actions from the external environment. The only difference between the original and the resulting I/O automaton is in the signature. The executions stay the same, but the traces change.

Definition 2.7. An injective mapping f is applicable to an I/O automaton A if $acts(A) \subseteq dom(f)$. Given an I/O automaton $A = (Q, Q_0, S, t, P)$ and a mapping f applicable to it, we define $f(A)$ to be (Q, Q_0, S', t', P') where S', t' and P' are defined as follows

- $in(S) = f(in(A))$, $out(S) = f(out(A))$, $int(S) = f(int(A))$,
- $t = \{(q, f(a), q') : (q, a, q') \in steps(A)\}$, and
- $P = \{(f(a), f(a')) : (a, a') \in part(A)\}$.

Thus, the renaming operator simply renames actions of its operand. For the parallel composition we need a notion of compatibility for action signatures.

Definition 2.8.

1. A set of action signatures $\{S_i : i \in I\}$ are *strongly compatible* iff for all $i, j \in I$, $i \neq j$,
   (a) $out(S_i) \cap out(S_j) = \emptyset$, and
   (b) $int(S_i) \cap acts(S_j) = \emptyset$.
2. A set of I/O automata $\{A_i : i \in I\}$ are *strongly compatible* iff their action signatures are strongly compatible.

Definition 2.9. The parallel composition $\prod_{i \in I} A_i$ of strongly compatible I/O automata $\{A_i : i \in I\}$ is defined to be the I/O automaton A with

1. $states(A) = \prod_{i \in I} states(A_i)$,
2. $start(A) = \prod_{i \in I} start(A_i)$,
3. $sig(A) = \prod_{i \in I} sig(a_i)$,
   where the composition $S = \prod_{i \in I} S_i$ of strongly compatible action signatures $\{S_i : i \in I\}$ is defined by
   (a) $in(S) = \bigcup_{i \in I} in(S_i) - \bigcup_{i \in I} out(S_i)$,
   (b) $out(S) = \bigcup_{i \in I} out(S_i)$,
   (c) $int(S) = \bigcup_{i \in I} int(S_i)$,
4. $part(A) = \bigcup_{i \in I} part(A_i)$,
5. $steps(A) = \{ \ ((q_i)_{i \in I}, a, (q'_i)_{i \in I}) : \forall i \in I a \notin acts(A_i)$ implies $q_i = q'_i$, and $a \in acts(A_i)$ implies $(q_i, a, q'_i) \in steps(A_i)\}$

3. The Problem

In this section, we define the problem that we are going to solve using both the simulation and algebraic methods. This problem is that of verifying the correctness of a particular circuit implementation. We begin with an informal description, then present the formal version in several pieces.

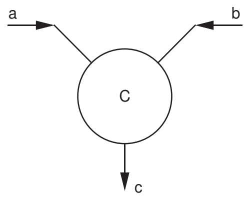

Fig. 1. The Muller C element

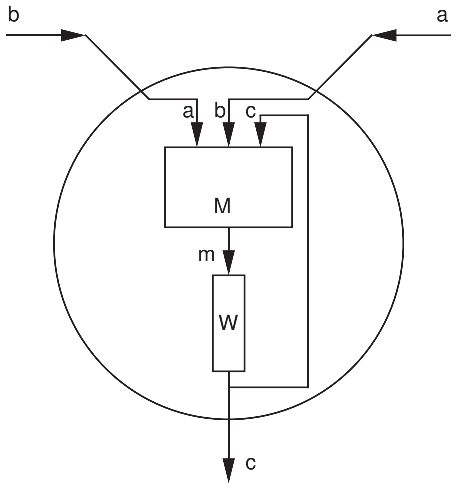

Fig. 2. A majority element and a wire implementing a Muller C element

3.1. Informal Description

The example consists of a simple delay insensitive circuit, taken from [Jos92], called the *Muller C element* [MuB59]. Its interface is shown in Figure 1. A Muller C element has two input ports a, b and one output port c. Once it is in its initial state with all input and output voltage levels low, a Muller C element waits for both its inputs to reach the high voltage level for then raising its output voltage level. It then waits for both its inputs to reach the low voltage level for then reaching again its initial state. In our specification no changes on the input ports are allowed whenever the voltage level of an output port has to change. Real implementations may exhibit unexpected behaviors (such as the glitch phenomenon) in such cases. For the above reason we do not specify the behavior of any element whenever an output voltage level has to change and an input occurs.

A Muller C element can be implemented by a *majority element* and a *wire* as shown in Figure 2. A majority element is a device with three input ports and one output port. The voltage level of its output port is that of the majority of its input ports. For the majority element we allow the change of level of an input port even if the output port has to change level. The required condition is that the new input does not affect the ports that have to change voltage level.

A wire is simply a device with one input and one output. It waits for a change of level on its input port for then changing the voltage level of its output port.

Our problem is to verify that a Muller C element can really be implemented by a majority element and a wire.

3.2. Formal Description

3.2.1. Actions as Voltage Level Transitions

In our formalization we use actions to model changes of voltage level (either from low to high or from high to low) at a port. The observation of an action does not give any information whether the voltage transition is from high to low or vice versa. Our use of actions is a consequence of the fact that the elements of the problem we are analyzing can be described simply in terms of voltage level transitions.

3.2.2. Specifications of the Elements

The specification $\mathcal{S}$ of an element is a tuple (Q, Q_0, S, T, P) consisting of a set of states Q, a set of start states Q_0, an interface S consisting of three disjoint sets of input, output and internal actions respectively, a transition table T, and a partition of the locally controlled actions P. The transition table gives, for each state and action, the future state, or *not specified* (NS), or *not enabled* (NE). The entry *not specified* is reserved for input actions and stands for "the environment is not supposed to provide input at this point"; the entry *not enabled* is reserved for local actions and stands for "this action cannot occur at this point".

The specification style outlined above does not define I/O automata directly; however, it allows specifications that are very close to the informal specifications of Section 3. Later in this section we define formally how to interpret the specifications below as I/O automata. The Muller C element, the wire and the majority element specifications are denoted by C_N, W_N and M_N, respectively. Here, N stands for "neutral" in the sense that these specifications are not biased toward either of the representation methods or verification techniques we introduce later. We start with the formal specification of a Muller C element.

Specification 3.1. A Muller C element C_N is defined as follows.

$S = (\{a, b\}, \{c\}, \emptyset)$
$Q = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$
$Q_0 = \{\emptyset\}$
$P = \{\{c\}\}$

The transition relation is defined by the following table:

	a	b	c
$\emptyset$	$\{a\}$	$\{b\}$	NE
$\{a\}$	$\emptyset$	$\{a, b\}$	NE
$\{b\}$	$\{a, b\}$	$\emptyset$	NE
$\{a, b\}$	NS	NS	$\emptyset$

It is easy to check that the above specification corresponds to the informal one given in Section 3. Starting from a state $\emptyset$ where the voltage level of each port is the same (say low), the occurrence of an input action would cause the system to move to a new state in which the new voltage level of the given input port is considered. When the voltage level of both the input ports is different from the voltage level of the output port (state $\{a, b\}$) the output action c is enabled and no input is allowed to occur.

Specification 3.2. A wire W_N is defined as follows.

$S = (\{m\}, \{c\}, \emptyset)$
$Q = \{\lambda, m\}$
$Q_0 = \{\lambda\}$
$P = \{\{c\}\}$

The transition relation is defined by the following table:

	m	c
λ	m	NE
m	NS	λ

Specification 3.3. A majority element M_N is defined as follows.

$S = (\{a, b, c\}, \{m\}, \emptyset)$
$Q = 2^{\{a,b,c\}}$
$Q_0 = \{\emptyset\}$
$P = \{\{m\}\}$

The transition relation is defined by the following table:

	a	b	c	m
$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	NE
$\{a\}$	$\emptyset$	$\{a, b\}$	$\{a, c\}$	NE
$\{b\}$	$\{a, b\}$	$\emptyset$	$\{b, c\}$	NE
$\{c\}$	$\{a, c\}$	$\{b, c\}$	$\emptyset$	NE
$\{a, b\}$	NS	NS	$\{a, b, c\}$	$\{c\}$
$\{a, c\}$	NS	$\{a, b, c\}$	NS	$\{b\}$
$\{b, c\}$	$\{a, b, c\}$	NS	NS	$\{a\}$
$\{a, b, c\}$	$\{b, c\}$	$\{a, c\}$	$\{a, b\}$	$\emptyset$

3.2.3. From Specifications to I/O Automata

The formal specifications of Section 3.2.2 are not I/O automata since their transition relations are not input enabled. In particular it is necessary to define carefully the meaning of the two special symbols NE and NS. The meaning of NE is trivial: if $T(q, a) = \text{NE}$ for a state q and an output action a, then no transition with action a occurs from state q. If $T(q, a) = \text{NS}$ for a state q and an input action a, then, since an I/O automaton is input enabled, a transition from q with action a must be defined. Intuitively we do not wish to constrain the behavior of any implementation in the presence of an unspecified input. In other words we want any implementation to be correct independently of the behaviors it exhibits in the presence of some input that is not specified in the specification. Since the implementation relation of I/O automata is the fair preorder, the above intuition is captured by introducing a new special state Ω, and, whenever $T(q, a) = \text{NS}$, by introducing a transition $q \xrightarrow{a} \Omega$. The transition relation on

Ω has to be defined in such a way that, given any sequence of actions β, it is possible to find a fair execution fragment α whose first state is Ω and such that $trace(\alpha) = \beta$.

Definition 3.4. Given a specification $\mathcal{S} = (Q, Q_0, (in, out, int), T, P)$ the I/O automaton $A = \mathcal{A}(\mathcal{S})$ is defined as

- $states(A) = Q \cup \{\Omega\}$.
- $start(A) = Q_0$.
- $sig(A) = (in, out, int \cup \{\tau_p \mid p \in P\})$.
- $(q, a, q') \in steps(A)$ iff
  - $T(q, a) = q'$ or
  - $T(q, a) = \mathrm{NS}$ and $q' = \Omega$ or
  - $q = q' = \Omega$.
- $part(A) = \{p \cup \{\tau_p\} \mid p \in P\}$.

The following proposition states that everything is possible whenever Ω is reached, i.e., any choice of implementation is correct whenever the specification reaches state Ω.

Proposition 3.5. Given a specification $\mathcal{S}$ and given any (possibly infinite) sequence β of external actions of $\mathcal{S}$, there exists a fair execution fragment α of $\mathcal{A}(\mathcal{S})$ whose first state is Ω and such that $trace(\alpha) = \beta$.

Proof. The execution fragment α interleaves the actions of β with one internal action from each class of $part(\mathcal{A}(\mathcal{S}))$. If β is finite then α fairly loops forever on the internal actions from each class of $part(\mathcal{A}(\mathcal{S}))$ after β is completed. By construction we know that each class has at least one internal action. Moreover Ω has a self loop with each action. $\square$

Now we can state the problem formally: verify that

$$Hide_{\{m\}}(\mathcal{A}(M_N) \parallel \mathcal{A}(W_N)) \sqsubseteq_F \mathcal{A}(C_N).$$

4. A Verification using Simulation

In this section we carry out the verification of Section 3.2 using simulation-based assertional techniques. We begin by presenting the relevant theory, then give variants of the specifications of Section 3.2 that are better suited for carrying out a simulation proof, and finally carry out the steps of the proof.

4.1. The Theory

In order to prove that an I/O automaton A implements another I/O automaton B, it is necessary to prove that each fair trace of A is also a fair trace of B. Our strategy for doing this is to obtain a strong correspondence between each execution of A and *some* execution of B; one way of obtaining such a correspondence is by using a forward simulation. Then, the proof of fair trace inclusion can be carried out in terms of the correspondence between executions.

In the fairness proof, it is notationally advantageous to use a generalization of I/O automata known as *forcing I/O automata*; this generalization does not increase the expressive power of the model, but does allow more concise representations.

Below, we define forward simulations, state the Execution Correspondence Lemma, and give the needed definitions and results for forcing I/O automata.

4.1.1. Forward Simulations and the Execution Correspondence Lemma

The notion of forward simulation that we use is taken from the comprehensive paper by Lynch and Vaandrager [LyV91].

Definition 4.1. A *forward simulation* from an I/O automaton A to an I/O automaton B is a relation f over $states(A)$ and $states(B)$ that satisfies:

1. If $q \in start(A)$ then $f[q] \cap start(B) \neq \emptyset$.
2. If $q \xrightarrow{a} q'$ and $p \in f[q]$, then there exists a state $p' \in f[q']$ such that $p \stackrel{a\lceil ext(B)}{\Longrightarrow} p'$.

The usual conclusion that is drawn from the existence of a forward simulation is trace inclusion:

Lemma 4.2. Given two I/O automata A, B, if there is a forward simulation from A to B, then $traces(A) \subseteq traces(B)$.

However, since we would like to base our proof of fair trace inclusion on our proof of trace inclusion, it is useful to have a stronger consequence of the existence of a forward simulation. This lemma is proved in [GSSL93].[3]

Lemma 4.3. Let f be a forward simulation from an I/O automaton A to an I/O automaton B. Then, for each execution $\alpha = q_0 a_1 q_1 a_2 q_2 \cdots$ of A there is an execution $\alpha' = q'_0 b_1 q'_1 b_2 q'_2 \cdots$ of B and a total monotone nondecreasing mapping $c : \{0, \ldots, |\alpha|\} \rightarrow \{0, \ldots, |\alpha'|\}$ such that

1. $c(0) = 0$,
2. $q'_{c(i)} \in f(q_i)$ for all $0 \leq i \leq |\alpha|$,
3. $b_{c(i)+1} \cdots b_{c(i+1)} \lceil ext(B) = a_{i+1} \lceil ext(A)$ for all $0 \leq i \leq |\alpha|$, and
4. for all q'_j there exists an i such that $c(i) \geq j$.

If the forward simulation is well chosen, then Proposition 4.3 can be used as the basis for a proof of fair trace inclusion as follows. For each fair execution α of A, first produce a corresponding execution α' of B. Then show that the fairness of α implies the fairness of *any* corresponding execution of B. This is the general strategy that is followed in our proof.

4.1.2. Forcing I/O Automata

In carrying out the proof of fairness, it turns out to be notationally convenient to use a slight generalization of I/O automata that we call *forcing I/O automata* [SLL93b]. The generalization consists of associating a set of states, called a *forcing set*, with each class of $part(A)$. Forcing I/O automata are no more expressive

[3] In [GSSL93], it is also shown that a similar lemma holds for other types of simulation relations such as backward simulations.

than ordinary I/O automata in terms of the sets of fair traces they can represent; they are useful, however, because sometimes they admit more concise representations.

Definition 4.4. A *forcing I/O automaton* A is an I/O automaton with the following additional structure:

- a function $force(A)$ associating a set of states with each partition of $part(A)$ such that, for each partition $p \in part(A)$ and each state $q \in force(A)(p)$, there exists an action from p which is enabled from q. The set $force(A)(p)$ is called the *forcing set* of p. It is a subset of the states enabling some action from p. The set of states enabling some action from p is denoted by $enabling(p)$.

The notion of fair execution for forcing I/O automata differs from that of ordinary I/O automata in that fairness is now expressed only with respect to states in the forcing set of each class p of local actions.

Definition 4.5. A *fair execution fragment* of a forcing I/O automaton A is an execution fragment $\alpha \in frag(A)$ such that for all $X \in part(A)$

- If α is finite then the final state of α is not in the forcing set of X.
- If α is infinite then either actions from X appear infinitely often in α or states not in the forcing set of X appear infinitely often in α.

A *fair execution* is a fair execution fragment whose first state is a start state.

The following proposition says that forcing I/O automata do not add any new expressive power to the I/O automaton model; moreover, it gives a particular transformation from forcing I/O automata to I/O automata.

Proposition 4.6. Given a forcing I/O automaton A, consider an I/O automaton $\mathcal{F}(A)$ where

- $states(\mathcal{F}(A)) = states(A)$
- $start(\mathcal{F}(A)) = start(A)$
- $sig(\mathcal{F}(A)) = (in(A), out(A), int(A) \cup \{\tau_p \mid p \in part(A)\})$
- $steps(\mathcal{F}(A)) = steps(A) \cup \{(q, \tau_p, q) \mid p \in part(A),\ q \in (enabling(p) \setminus force(p))\}$
- $part(\mathcal{F}(A)) = \{p \cup \{\tau_p\} \mid p \in part(A)\}$

Then $ftraces(A) = ftraces(\mathcal{F}(A))$.

Proof. Let β be a fair trace of A and let α be a fair execution of A such that $trace(\alpha) = \beta$. Build an execution α' of $\mathcal{F}(A)$ from α in the following way: at each state of α add a self loop with all the τ_p actions that are enabled; if α is finite, loop forever on the final state of α by performing all the enabled τ_p actions in a Round-Robin way. Note that $trace(\alpha') = \beta$, so it is enough to show that α' is a fair execution of $\mathcal{F}(A)$. Suppose that α' is not fair for $\mathcal{F}(A)$. If α' is finite then there exists a class p of A with an enabled action from the last state of α' and such that τ_p is not enabled from the last state of α'. Also, α is finite and its last state enables an action from p. By definition of $\mathcal{F}$ the last state of α is in the forcing set of p, therefore α is not a fair execution of A; a contradiction. Suppose that α' is infinite and that there is a class p of A and a suffix α'' of α' such that actions from $p \cup \{\tau_p\}$ are continuously enabled but never performed in α''. By definition of α', τ_p is never enabled in α'', hence actions from p are always enabled and never performed in α''. By definition of $\mathcal{F}$, all the states of

α'' are in the forcing set of p, hence there exists a suffix of α where actions from p are always enabled and never performed and whose states are all in the forcing set of p, i.e., α is not fair; again a contradiction.

Conversely, let β be a fair trace of $\mathcal{F}(A)$ and let α be a fair execution of $\mathcal{F}(A)$ such that $trace(\alpha) = \beta$. Build an execution α' of A by removing from α all the transitions with actions of the form τ_p. Note that $trace(\alpha') = \beta$, so it is enough to show that α' is a fair execution of A. If α is finite, then the last state of α does not enable any action from any class p, hence also the last state of α' does not enable any action from any class p, and α' is fair. If α is infinite and α' is finite, then, by definition of α' and $\mathcal{F}$, the last state of α' is not in the forcing set of any class p, hence α' is fair. If α is infinite and α' is infinite, then, for each class p, there are three possible cases. If states not enabling actions from $p \cup \{\tau_p\}$ appear infinitely often in α, then states not enabling actions from p appear infinitely often in α'; if actions from p appear infinitely often in α, then actions from p appear infinitely often in α'; if actions from $p \cup \{\tau_p\}$ appear infinitely often in α but actions from p appear finitely many times in α, then, by definition of $\mathcal{F}$, states not in the forcing set of p appear infinitely often in α'. In all of the above cases the conditions for α' to be fair are satisfied, therefore α' is a fair execution of A. $\square$

The standard operators of I/O automata can be easily extended to forcing I/O automata. The only nontrivial extension is that of the parallel operator, where the forcing set of each class has to be modified to take into account the states of the other forcing I/O automata. Consider for example a forcing I/O automaton A composed in parallel with a forcing I/O automaton B and let q be in the forcing set of some class p of A. Whenever A reaches state q in the composition $A \parallel B$, we want the global state of $A \parallel B$ to be in the forcing set of p. Therefore all states of $\{q\} \times states(B)$ have to be in the new forcing set of p.

Definition 4.7. The parallel composition $\prod_{i \in I} A_i$ of strongly compatible forcing I/O automata $\{A_i : i \in I\}$, denoted by A, is the composition of their ordinary part augmented with new forcing sets as follows: for each class $p \in part(p_j)$, $force(A)(p) = force(A_j)(p) \times \prod_{i \in I \setminus j} A_i$.

Proposition 4.8. Given two forcing I/O automata A, B,

1. $\mathcal{F}(Hide_I(A))$ and $Hide_I(\mathcal{F}(A))$ are the same I/O automaton;
2. $\mathcal{F}(A \parallel B)$ and $\mathcal{F}(A) \parallel \mathcal{F}(B)$ are the same I/O automaton.

Proof. The first statement is trivial since the hiding operator changes only the signature of an I/O automaton and the result of $\mathcal{F}$ does not depend on which actions of an I/O automaton are internal and which ones are external. For the second statement we verify that the two involved I/O automata are the same one by verifying each component separately.

$$\begin{aligned} states(\mathcal{F}(A \parallel B)) &= states(A \parallel B) \\ &= states(A) \times states(B) \\ &= states(\mathcal{F}(A)) \times states(\mathcal{F}(B)) \\ &= states(\mathcal{F}(A) \parallel \mathcal{F}(B)) \end{aligned}$$

$$\begin{aligned} start(\mathcal{F}(A \parallel B)) &= start(A \parallel B) \\ &= start(A) \times start(B) \\ &= start(\mathcal{F}(A)) \times start(\mathcal{F}(B)) \\ &= start(\mathcal{F}(A) \parallel \mathcal{F}(B)) \end{aligned}$$

$$\begin{array}{rcl} out(\mathcal{F}(A \parallel B)) & = & out(A \parallel B) = out(A) \cup out(B) \\ & = & out(\mathcal{F}(A)) \cup out(\mathcal{F}(B)) \\ & = & out(\mathcal{F}(A) \parallel \mathcal{F}(B)) \end{array}$$

$$\begin{array}{rcl} in(\mathcal{F}(A \parallel B)) & = & in(A \parallel B) \\ & = & (in(A) \cup in(B)) \backslash out(A \parallel B) \\ & = & (in(\mathcal{F}(A)) \cup in(\mathcal{F}(B))) \backslash out(\mathcal{F}(A) \parallel \mathcal{F}(B)) \\ & = & in(\mathcal{F}(A) \parallel \mathcal{F}(B)) \end{array}$$

$$\begin{array}{rcl} int(\mathcal{F}(A \parallel B)) & = & int(A \parallel B) \cup \{\tau_p \mid p \in part(A \parallel B)\} \\ & = & int(A) \cup int(B) \cup \{\tau_p \mid p \in part(A) \cup part(B)\} \\ & = & (int(A) \cup \{\tau_p \mid p \in part(A)\}) \cup \\ & & (int(B) \cup \{\tau_p \mid p \in part(B)\}) \\ & = & int(\mathcal{F}(A)) \cup int(\mathcal{F}(B)) \\ & = & int(\mathcal{F}(A) \parallel \mathcal{F}(B)) \end{array}$$

$$\begin{array}{rcl} part(\mathcal{F}(A \parallel B)) & = & \{p \cup \{\tau_p\} \mid p \in part(A \parallel B)\} \\ & = & \{p \cup \{\tau_p\} \mid p \in part(A) \cup part(B)\} \\ & = & \{p \cup \{\tau_p\} \mid p \in part(A)\} \cup \{p \cup \{\tau_p\} \mid p \in part(B)\} \\ & = & part(\mathcal{F}(A)) \cup part(\mathcal{F}(B)) \\ & = & part(\mathcal{F}(A) \parallel \mathcal{F}(B)) \end{array}$$

The argument for *steps* is more complicated. Let $((q_A, q_B), a, (q'_A, q'_B)) \in steps(\mathcal{F}(A \parallel B))$. If a is not an action of the form τ_p, then $((q_A, q_B), a, (q'_A, q'_B)) \in steps(A \parallel B)$. From the definition of the parallel composition operator and from the fact that $steps(C) \subseteq steps(\mathcal{F}(C))$ for each forcing I/O automaton C, it is immediate do derive that $((q_A, q_B), a, (q'_A, q'_B)) \in steps(\mathcal{F}(A) \parallel \mathcal{F}(B))$. If a is of the form τ_p, then suppose without loss of generality that $p \in part(A)$. Then $(q_A, q_B) \in enabling(p) \backslash force(p)$ in $A \parallel B$, and, by definition of parallel composition for forcing I/O automata, $q_a \in enabling(p) \backslash force(p)$ in A. By definition of $\mathcal{F}$, $(q_A, a, q'_A) \in steps(\mathcal{F}(A))$. By definition of parallel composition, $((q_A, q_B), a, (q'_A, q'_B)) \in steps(\mathcal{F}(A) \parallel \mathcal{F}(B))$.

Conversely, let $((q_A, q_B), a, (q'_A, q'_B)) \in steps(\mathcal{F}(A) \parallel \mathcal{F}(B))$. If a is not an action of the form τ_p, then a direct analysis of the definition of the parallel composition operator shows that $((q_A, q_B), a, (q'_A, q'_B)) \in steps(\mathcal{F}(A \parallel B))$. If a is of the form τ_p, then suppose without loss of generality that $p \in part(A)$. By definition of $\mathcal{F}$, $q_a \in enabling(p) \backslash force(p)$ in A. By definition of parallel composition for forcing I/O automata, $(q_A, q_B) \in enabling(p) \backslash force(p)$ in $A \parallel B$. By definition of $\mathcal{F}$, $((q_A, q_B), a, (q'_A, q'_B)) \in steps(\mathcal{F}(A \parallel B))$. $\square$

4.2. Specification of the Components

In Section 3.2 we described the system components using a "neutral" formalism that is not biased toward either verification method. Each of the two methods, however, has its own characteristic language for describing system components. In this section, we represent each element of Section 3.2 using a variant of the precondition-effect language of [LyT87] that is suitable for describing forcing I/O automata. We also relate the new specifications to the neutral ones.

In our precondition-effect language a forcing I/O automaton is described by means of its action signature, its states, its initial states, its transition relation, and its classes with forcing sets. The transition relation is specified by means of the preconditions for the execution of each action and the effect each action produces on the state. The precondition of an action gives the set of states from which it is enabled or from which it is expected; the effect gives the next state. If an input action occurs when its precondition is not satisfied, then the system moves to a special state Ω. The state Ω implicitly has a transition to itself for each action and it does not appear in the forcing set of any class of local actions.

Note that this representation can be more concise than the neutral representation, because states need not all be listed explicitly. Rather, they are described in terms of values of a collection of state variables.

In order to simplify the notation we introduce an operator $\oplus$ on sets corresponding to the symmetric difference operator. Note that the transition relations of the forcing I/O automata we introduce below differ from those of Section 3.2 only in the definition of state Ω. As a consequence, the specifications of this section and those of Section 3.2 denote I/O automata with the same set of fair traces. In fact, the connection between the I/O automata is much closer than this; we give a formal statement of the connection after the specifications.

Specification 4.9. A Muller C element C_F is defined as follows.

$S = (\{a, b\}, \{c\}, \emptyset)$
$Q = \{\emptyset, \{a\}, \{b\}, \{a, b\}, \Omega\}$
$Q_0 = \{\emptyset\}$
$P = \{\{c\}\}$ where $\{c\}$ has forcing set $\{\{a, b\}\}$

Transitions:

Action a
  Precondition: $q \neq \{a, b\}$
  Effect: $q' := q \oplus \{a\}$

Action b
  Precondition: $q \neq \{a, b\}$
  Effect: $q' := q \oplus \{b\}$

Action c
  Precondition: $q = \{a, b\}$
  Effect: $q' := \emptyset$

Specification 4.10. A majority element M_F is defined as follows.

$S = (\{a, b, c\}, \{m\}, \emptyset)$
$Q = 2^{\{a,b,c\}} \cup \{\Omega\}$
$Q_0 = \{\emptyset\}$
$P = \{\{m\}\}$ where $\{m\}$ has forcing set $\{\{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$

Transitions:

Action a
  Precondition: $q \notin \{\{a, b\}, \{a, c\}\}$
  Effect: $q' := q \oplus \{a\}$

Action b
  Precondition: $q \notin \{\{a,b\},\{b,c\}\}$
  Effect: $q' := q \oplus \{b\}$

Action c
  Precondition: $q \notin \{\{a,c\},\{b,c\}\}$
  Effect: $q' := q \oplus \{c\}$

Action m
  Precondition: $|q| \geq 2$
  Effect: $q' := \{a,b,c\} \backslash q$

Specification 4.11. A wire W_F is defined as follows.

$S = (\{m\},\{c\},\emptyset)$
$Q = \{\lambda, m, \Omega\}$
$Q_0 = \{\lambda\}$
$P = \{\{c\}\}$ where $\{c\}$ has forcing set $\{\{m\}\}$

Transitions:

Action m
  Precondition: $q = \lambda$
  Effect: $q' := m$

Action c
  Precondition: $q = m$
  Effect: $q' := \lambda$

Proposition 4.12.

1. $\mathcal{A}(C_N)$ and $\mathcal{F}(C_F)$ denote the same I/O automaton.
2. $\mathcal{A}(M_N)$ and $\mathcal{F}(M_F)$ denote the same I/O automaton.
3. $\mathcal{A}(W_N)$ and $\mathcal{F}(W_F)$ denote the same I/O automaton.

Proof. The proof is a simple analysis of the definitions. We argue a bit more about the first statement and leave the other two to the reader. The states of $\mathcal{A}(C_N)$ and $\mathcal{F}(C_F)$ are the same since the states of C_F are those of C_N plus Ω and $\mathcal{A}$ adds a new state Ω to the states of C_N. Similarly, the start states are the same in $\mathcal{A}(C_N)$ and $\mathcal{F}(C_F)$. The partitions of the locally-controlled actions are the same since both C_N and C_F have a unique class and both $\mathcal{A}$ and $\mathcal{F}$ add a new internal action τ_p to each class p. Similarly, the action signatures of the two I/O automata are the same. The transition relations of the two I/O automata are the same since the preconditions of the actions of C_F identify those cells of the transition table of C_N that do not contain NS or NE, the effects of each action coincide in C_N and C_F, all the states of C_F but Ω are in the forcing set of the unique partition of $part(C_F)$, and $\mathcal{A}$ deals with unspecified inputs by moving to Ω. $\square$

4.3. The Verification

We finally prove that a Muller C element is implemented by a majority element and a wire. We first prove a proposition expressing this claim for forcing I/O automata. At the end of this subsection, we show how to derive the precise claim of Section 3.2.

Proposition 4.13. $Hide_{\{m\}}(M_F \parallel W_F) \sqsubseteq_F C_F$, i.e., a Muller C element can be implemented by a majority element and a wire.

Proof. We define a mapping from the implementation to the specification and show that it is a forward simulation. Then we use the Execution Correspondence Lemma to obtain corresponding executions and use this correspondence to prove fair trace inclusion.

More precisely, the mapping f to use is the following:

$(\emptyset, \lambda) \mapsto \{\emptyset\}$
$(\{a\}, \lambda) \mapsto \{\{a\}, \Omega\}$
$(\{b\}, \lambda) \mapsto \{\{b\}, \Omega\}$
$(\{a, b\}, \lambda) \mapsto \{\{a, b\}, \Omega\}$
$(\{c\}, m) \mapsto \{\{a, b\}, \Omega\}$
all other pairs $\mapsto \{\omega\}$

We first prove that the above relation is a forward simulation. The condition on the initial states is immediate to verify since the initial state $(\emptyset, \lambda)$ is mapped to the initial state $\emptyset$. For the transition relation we proceed by cases analysis on action names.

Action a: We distinguish the following cases:

- If a occurs from (x, λ) where $x \in \{\emptyset, \{a\}, \{b\}\}$ then $(x, \lambda) \xrightarrow{a} (x \oplus \{a\}, \lambda)$ and $x \xrightarrow{a} x \oplus \{a\}$.
- If a occurs from $(\{a, b\}, \lambda)$ then $(\{a, b\}, \lambda) \xrightarrow{a} (\Omega, \lambda)$. Moreover $\{a, b\} \xrightarrow{a} \Omega$ and $\Omega \xrightarrow{a} \Omega$.
- If a occurs from $(\{c\}, m)$ then $(\{c\}, m) \xrightarrow{a} (\Omega, m)$. Moreover $\{a, b\} \xrightarrow{a} \Omega$ and $\Omega \xrightarrow{a} \Omega$.
- If a occurs from any state (x, λ) where $x \notin \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ then $(x, \lambda) \xrightarrow{a} (x', \lambda)$ and $x' \notin \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. Moreover $\Omega \xrightarrow{a} \Omega$.
- If a occurs from any state (x, m) where $x \neq \{c\}$ then $(x, m) \xrightarrow{a} (x', m)$ and $\Omega \xrightarrow{a} \Omega$. Note that, since for $x = \{a, c\}$ we have $x' = \{c\}$, we need Ω in the mapping for $(\{c\}, m)$.

Action b: This case is the same as the case for action a.
Action c: This action is enabled only from states of the form (x, m) and yields a new state (x', λ). If $x = \{c\}$ then $x' = \emptyset$ and $\{a, b\} \xrightarrow{c} \emptyset$. In all other cases x' can be anything but $\emptyset$. This is the case for which we need to map $(\{a\}, \lambda)$, $(\{b\}, \lambda)$ and $(\{a, b\}, \lambda)$ to Ω.
Action m: This action is enabled from each state (x, λ) and (x, m) with $|x| \leq 2$. If the starting state is (x, m) then the final state is (x', Ω). Moreover both starting and final states are mapped to Ω. If the starting state is (x, λ) with $x \neq \{a, b\}$ then the final state is (x', m) and both starting and final states are mapped to Ω. If the starting state is $(\{a, b\}, \lambda)$ then $(\{a, b\}, \lambda) \xrightarrow{m} (\{c\}, m)$ and both starting and final states are mapped to $\{a, b\}$ and Ω.

The existence of the above forward simulation allows us to conclude that each trace of $Hide_{\{m\}}(M_F \parallel W_F)$ is a trace of C_F. We now use the same simulation to argue that each fair trace of $Hide_{\{m\}}(M_F \parallel W_F)$ is a fair trace of C_F. Consider a generic fair execution α of $Hide_{\{m\}}(M_F \parallel W_F)$. By the Execution Correspondence Lemma, there is an execution α' of C_F corresponding to α through the mapping f. It is sufficient to argue that α' is fair to conclude.

Suppose that α' is not a fair execution of C_F. The only way the fairness for C_F can be violated is for the states in α' to be $\{a, b\}$ for some point on without c ever occurring. (In fact $\{a, b\}$ is the only state in the forcing set for $\{c\}$.) Then in α, the correspondence says that the states are all either (ab, ϕ) or (c, m) from that point on. If there is any occurrence of a (c, m) state, then the fairness condition for W_F says that eventually c occurs in α, so also in α', a contradiction. So the state must be (ab, ϕ) forever. But then the fairness condition for M_F says that eventually m occurs, changing the state to (c, m), again a contradiction. $\square$

Note that the fairness part of the proof above is done somewhat less formally than the simulation part; the fairness part can be formalized using a temporal logic of states and actions [Sta84, SLL93b].

Now we can give the main result:

Theorem 4.14. $Hide_{\{m\}}(\mathcal{A}(M_N) \parallel \mathcal{A}(W_N)) \sqsubseteq_F \mathcal{A}(C_N)$.

Proof. From Propositions 4.13 and 4.6 we derive $\mathcal{F}(Hide_{\{m\}}(M_F \parallel W_F)) \sqsubseteq_F \mathcal{F}(C_F)$. From Proposition 4.8 we derive $Hide_{\{m\}}(\mathcal{F}(M_F) \parallel \mathcal{F}(W_F)) \sqsubseteq_F \mathcal{F}(C_F)$. From Proposition 4.12 we obtain $Hide_{\{m\}}(\mathcal{A}(M_N) \parallel \mathcal{A}(W_N)) \sqsubseteq_F \mathcal{A}(C_N)$. $\square$

5. A Verification using Process Algebras

In this section we carry out the verification of Section 3.2 using process algebra. Again, we begin by presenting the relevant theory, then give new specifications, and finally carry out the steps of the proof.

5.1. The Theory

As before, our job is to prove a fair trace inclusion relationship between two I/O automata. In general, process algebra is not well suited for proving results about fairness, because fairness does not fit nicely into the theory of a process algebra containing recursion. However, process algebra can be used to reason about an approximation to fairness known as *quiescence*, and under certain circumstances, this may be enough.

Below, we define quiescence and relate it to fairness. We then define DIOA ("Demonic I/O Automata"), a process algebra for proving quiescent trace inclusion relationships between I/O automata. [4]

[4] The adjective "demonic" is used suggestively in [Seg92] to emphasize the fact that demonic I/O automata behave catastrophically in the presence of unexpected inputs. It is in contrast with the approach of [Vaa91] which is called "angelic" in [Seg92].

5.1.1. From the Quiescent Preorder to the Fair Preorder

Definition 5.1. A *quiescent execution* of an I/O automaton A is a finite fair execution of A. A *quiescent trace* is the trace of a quiescent execution. We denote the set of quiescent traces of an I/O automaton A by $qtraces(A)$.

Definition 5.2. Given two I/O automata A and B with the same external action signature, the *quiescent preorder* is defined as

$$A \sqsubseteq_Q B \text{ iff } traces^*(A) \subseteq traces^*(B) \text{ and } qtraces(A) \subseteq qtraces(B).$$

The quiescent preorder was first introduced in [Vaa91] and is an attempt at approximating the fair preorder by looking only at the finite executions of an I/O automaton. As pointed out through some examples in [Seg92], the quiescent preorder is not an intuitively reasonable notion of implementation in general; however, [Seg93] gives some sufficient conditions for the quiescent preorder to coincide with the fair preorder. Below we present some of the results of [Seg93]. We start with some definitions.

Definition 5.3. An I/O automaton A is *quiescent detectable* if each finite fair trace of A is also a quiescent trace of A.

Quiescence detectability requires each divergence to be detected through a quiescent trace. The fair preorder, in fact, does not distinguish between divergence and quiescence, while the quiescent preorder does.

Definition 5.4. An I/O automaton A is *quiescent continuous* if the limit of any chain of quiescent traces of A is a fair trace of A.

The quiescent preorder deals only with finite executions, while the fair preorder also considers infinite ones. A condition for the two preorders to coincide is that the information about infinite executions be captured by the information on the finite ones. To guarantee the above fact we also need finite internal nondeterminism.

Definition 5.5. An I/O automaton A has *finite internal nondeterminism* (FIN) if $\forall_{h \in acts^*(A)} \{q \mid \exists_{q_0 \in start(A)} q_0 \stackrel{h}{\Longrightarrow} q\}$ is finite.

The above definition of FIN is weaker than the definition given in [LyV91]. The definition of [LyV91] requires, for every trace h, the set of reachable states with h to be finite. In our definition we only require a smaller set to be finite, i.e., the set of states reachable through h with its last external transition.

Definition 5.6. An I/O automaton A is *input quiescent detectable* if each infinite fair trace of A with finitely many output actions has infinitely many prefixes that are quiescent for A.

An infinite fair trace made of input actions only can be obtained from an execution containing infinitely many internal transitions. The quiescent preorder, on the other hand, can detect only quiescent states.

Theorem 5.7. Given two I/O automata A_1, A_2 with the same external action signature such that $part(A_1) = \{local(A_1)\}$ and $part(A_2) = \{local(A_2)\}$, if A_1 is quiescent detectable and input quiescent detectable, and A_2 is fair continuous and has FIN, then

$A_1 \sqsubseteq_Q A_2$ implies $A_1 \sqsubseteq_F A_2$.

If A_2 is quiescent detectable then

$A_1 \sqsubseteq_F A_2$ implies $A_1 \sqsubseteq_Q T_2$.

Quiescent detectability and FIN are generally met by practical systems. Note, in fact, that systems without any infinite internal computation are quiescent detectable. Also quiescent continuity is generally true. In [Seg93] it is shown that, if an I/O automaton has FIN and is *input deterministic* (for each state q and each input action a there exists a unique state q' such that $q \stackrel{a}{\Longrightarrow} q'$), then it is quiescent continuous. It is not clear yet to us how general input quiescent detectability is.

Theorem 5.7 shows how the quiescent preorder can capture the fair preorder of some I/O automata with a single class of locally controlled actions. This is not the case for general I/O automata. However, there are cases in which the quiescent preorder is sufficient for concluding fair trace inclusion in the presence of multiple classes. When an I/O automaton has more than one class of locally controlled actions, the quiescent preorder is not of great help in deriving the fair preorder. The following proposition is of help whenever the specification automaton has a single class and the implementation automaton has multiple classes.

Proposition 5.8. Let A be an I/O automaton. If for each transition $q \stackrel{a}{\longrightarrow} q'$ of $steps(A)$ where a is an input action and each class x of $part(A)$, an action of x is enabled from q' if an action of x is enabled from q (i.e., input actions do not disable any class of $part(A)$), then $ftraces(A) \subseteq ftraces(A')$ where A' differs from A only in that $part(A') = \{local(A)\}$.

If an I/O automaton A with multiple classes implements an I/O automaton B with a single class, and if the involved automata satisfy the conditions of Theorem 5.7, then the proposition above gives a sufficient condition for deriving the full fair preorder from the quiescent preorder. In fact, from $A' \sqsubseteq_Q B$, where A' is the I/O automaton A with a single class, we derive $A' \sqsubseteq_F B$, and, from Proposition 5.8, we derive $A \sqsubseteq_F B$. Examples of systems satisfying the condition of Proposition 5.8 are the monotone I/O automata of [Sta90], which can model a large class of dataflow networks, and the semi-modular, speed-independent circuits of [MuB59]. Our problem is based on delay insensitive circuits.

5.1.2. The Calculus of Demonic I/O Automata

The calculus of Demonic I/O Automata (DIOA) is a process algebra for I/O automata [Seg92]. Each I/O automaton is an expression which is obtained by applying operators to basic automata. Each expression is *sorted* and each sort represents an external action signature. Each DIOA expression has a unique internal action τ. Multiple internal actions, in fact, are used within I/O automata for expressing fairness with respect to different internal tasks; however, DIOA does not deal with fairness. In this paper we present a slightly modified version of DIOA in which we consider multiple internal actions. Each sort represents a full action signature with multiple internal actions. Our modification does not change the algebraic properties of DIOA (the axioms do not change), but it makes it easier to relate DIOA proofs to simulation proofs. We assume that the sort of each DIOA expression contains at least one internal action and we use τ

Table 1. The signature of DIOA

Name	Op.	Domain	Range	Restrictions
quiescent	nil_S	λ	S	
omega	Ω_S	λ	S	
prefixing	$a._S$	S	S	$a \in ext(S)$
ichoice	$\oplus_S$	S, S	S	
echoice	${}_I+^S_J$	S, S	S	$I, J \subseteq in(S)$
parallel	${}_{S_1}\|_{S_2}$	S_1, S_2	S_3	$int(S_1) \cap acts(S_2) = acts(S_1) \cap int(S_2) = \emptyset$ $out(S_1) \cap out(S_2) = \emptyset$ $out(S_3) = out(S_1) \cup out(S_2)$ $in(S_3) = (in(S_1) \cup in(S_2)) \backslash out(S_3)$ $int(S_3) = int(S_1) \cup int(S_2)$
hiding	τ^S_I	S	S'	$I \subseteq out(S), S' = (in(S), out(S) \backslash I, int(S) \cup I)$
renaming	ρ_S	S	S'	for each injective $\rho : acts(S) \longrightarrow acts(S')$ $S' = (\rho(in(S)), \rho(out(S)), \rho(int(S)))$
process	X_S	λ	S	$X_S \in \mathcal{X}_S$

to denote a generic internal action. This assumption is necessary to model some of the operators.

Table 1 contains all the operators of DIOA; note that the sort of an expression is computable. Table 2 contains the operational semantics of DIOA in terms of transition systems. The operators of DIOA recall the standard operators of CCS [Mil89]; however they are different in the sense that they also guarantee input enabling by moving an automaton to the state Ω whenever some unexpected input is provided. The expression nil models a quiescent automaton that moves to Ω for any input. The prefixing operator allows the specification of an automaton which first performs a specific action a. The internal choice operator models nondeterministic choice independently of the external environment. Particularly unfamiliar to the process algebraic community is the external choice operator, which is parameterized by two sets of input actions. The two parameters describe which arguments of the operator deal with different input actions. Consider the expression $exp = a \,.\, e \;{}_{\{a\}}+_{\{b\}}\; b \,.\, f$. The subexpression $a \,.\, e$ describes the behavior of exp in the presence of input action a while the subexpression $b \,.\, f$ describes the behavior of exp in the presence of input action b. The parameters are necessary since $a \,.\, e$ also reacts to input b although that reaction is not desired. However, the meaning of an expression like $a \,.\, e + b \,.\, f$ is intuitively clear. Although this intuition is not expressible for general DIOA expressions, Table 3 defines a function $si(e)$ (Specified Inputs) which is capturing our intuitive idea for DIOA expressions of the kind $a_1 \,.\, e_1 + \cdots + a_n \,.\, e_n$. Function si allows us to define an unparameterized choice operator by writing $e + f$ for $e \;{}_{si(e)}+_{si(f)}\; f$, where function si is defined in Table 3. The interested reader is referred to [Seg92] for a more detailed description of si and its generalization to all DIOA expressions. Recursion is obtained by means of process variables and a declaration mapping E, which associates a guarded expression of sort S with each process variable of sort S. An expression is guarded if each process variable occurs within the scope of some prefixing operator.

Table 2. The transition rules for DIOA. τ is any internal action.

nil	$nil_S \xrightarrow{a} \Omega_S$	$\forall a \in in(S)$			
ome$_1$	$\Omega_S \xrightarrow{a} \Omega_S$	$a \in ext(S)$	**ome**$_2$	$\Omega_S \xrightarrow{\tau} nil_S$	
pre$_1$	$a \, ._S \, e \xrightarrow{a} e$		**pre**$_2$	$a \, ._S \, e \xrightarrow{b} \Omega_S$	$\forall b \in in(S) \backslash \{a\}$
ich$_1$	$e_1 \oplus_S e_2 \xrightarrow{\tau} e_1$		**ich**$_2$	$e_1 \oplus_S e_2 \xrightarrow{\tau} e_2$	
ich$_3$	$\dfrac{e_1 \xrightarrow{a} e_1'}{e_1 \oplus_S e_2 \xrightarrow{a} e_1'}$	$\forall a \in in(S)$	**ich**$_4$	$\dfrac{e_2 \xrightarrow{a} e_2'}{e_1 \oplus_S e_2 \xrightarrow{a} e_2'}$	$\forall a \in in(S)$
ech$_1$	$\dfrac{e_1 \xrightarrow{a} e_1'}{e_1 \; {}_I{+}_J^S \; e_2 \xrightarrow{a} e_1'}$	$\forall a \in I \cup out(S)$			
ech$_2$	$\dfrac{e_2 \xrightarrow{a} e_2'}{e_1 \; {}_I{+}_J^S \; e_2 \xrightarrow{a} e_2'}$	$\forall a \in J \cup out(S)$			
ech$_3$	$e_1 \; {}_I{+}_J^S \; e_2 \xrightarrow{a} \Omega_S$	$\forall a \in in(S) \backslash (I \cup J)$			
ech$_4$	$\dfrac{e_1 \xrightarrow{\tau} e_1'}{e_1 \; {}_I{+}_J^S \; e_2 \xrightarrow{\tau} e_1' \; {}_I{+}_J^S \; e_2}$		**ech**$_5$	$\dfrac{e_2 \xrightarrow{\tau} e_2'}{e_1 \; {}_I{+}_J^S \; e_2 \xrightarrow{\tau} e_1' \; {}_I{+}_J^S \; e_2'}$	
tau$_1$	$\dfrac{e \xrightarrow{a} e'}{\tau_I^S(e) \xrightarrow{a} \tau_I^S(e')}$		**rho**	$\dfrac{e \xrightarrow{a} e'}{\rho_S(e) \xrightarrow{\rho(a)} \rho_S(e')}$	
par$_1$	$\dfrac{e_1 \xrightarrow{a} e_1' \qquad e_2 \xrightarrow{a} e_2'}{e_1 \; {}_{S_1}\|_{S_2} \; e_2 \xrightarrow{a} e_1' \; {}_{S_1}\|_{S_2} \; e_2'}$				
par$_2$	$\dfrac{e_1 \xrightarrow{a} e_1'}{e_1 \; {}_{S_1}\|_{S_2} \; e_2 \xrightarrow{a} e_1' \; {}_{S_1}\|_{S_2} \; e_2}$	$a \in acts(S_1) \backslash ext(S_2)$			
par$_3$	$\dfrac{e_2 \xrightarrow{a} e_2'}{e_1 \; {}_{S_1}\|_{S_2} \; e_2 \xrightarrow{a} e_1 \; {}_{S_1}\|_{S_2} \; e_2'}$	$a \in acts(S_2) \backslash ext(S_1)$			
rec	$\dfrac{e \xrightarrow{a} e'}{X \xrightarrow{a} e'}$	if $X \stackrel{\text{def}}{=} e$			

Table 3. Definition of si and so for DIOA.

$si(nil) = \emptyset$	$so(nil) = \emptyset$
$si(\Omega) = \emptyset$	$so(\Omega) = out(\Omega)$
$si(a \, . \, e) = \{a\} \cap in(e)$	$so(a \, . \, e) = \{a\} \cap out(e)$
$si(e_1 \oplus e_2) = si(e_1) \cap si(e_2)$	$so(e_1 \oplus e_2) = so(e_1) \cup so(e_2)$
$si(e_1 \; {}_I{+}_J \; e_2) = (I \cap si(e_1)) \cup (J \cap si(e_2))$	$so(e_1 \; {}_I{+}_J \; e_2) = so(e_1) \cup so(e_2)$
$si(X) = si(E(X))$	$so(X) = so(E(X))$

Given a DIOA expression, there is a natural way of associating an I/O automaton with it. We arbitrarily choose not to partition its locally controlled actions. In this way Theorem 5.7 applies directly.

Definition 5.9. Given a DIOA expression e of sort S, the associated I/O automaton $\mathcal{D}(e)$ is defined as

- $states(\mathcal{D}(e)) = \{e' \mid \exists t \in acts(S)^*, e \xrightarrow{t} e'\}$
- $start(\mathcal{D}(e)) = \{e\}$
- $sig(\mathcal{D}(e)) = (in(S), out(S), int(S))$
- $steps(\mathcal{D}(e)) = \{(e', a, e'') \mid e' \in states(\mathcal{D}(e)), e' \xrightarrow{a} e''\}$
- $part(\mathcal{D}(e)) = \{local(S)\}$

For notational convenience we refer to $in(S)$, $out(S)$, $int(S)$, $acts(S)$ and $local(S)$ by $in(e)$, $out(e)$, $int(e)$, $acts(e)$ and $local(e)$, respectively.

Proposition 5.10. Given two DIOA expressions e, f,

1. $\mathcal{D}(\tau_I(e))$ and $Hide_I(\mathcal{D}(e))$ are isomorphic I/O automata under the isomorphism $h : states(\mathcal{D}(\tau_I(e))) \to states(Hide_I(\mathcal{D}(e)))$ such that $h(\tau_I(e')) = e'$;
2. $\mathcal{D}(e \parallel f)$ and $\mathcal{D}(e) \parallel \mathcal{D}(f)$ are almost isomorphic I/O automata uder the isomorphism $h : states(\mathcal{D}(e \parallel f)) \to states(\mathcal{D}(e) \parallel \mathcal{D}(f))$ such that $h(e' \parallel f') = (e', f')$. The only difference is in that $part(\mathcal{D}(e \parallel f)) = \{local(e) \cup local(f)\}$ and $part(\mathcal{D}(e) \parallel \mathcal{D}(f)) = \{local(e), local(f)\}$.

Proof. We give the proof for the hiding operator. The proof for the parallel composition operator is similar and is left to the reader.

$$\begin{aligned}
states(Hide_I(\mathcal{D}(e))) &= states(\mathcal{D}(e)) \\
&= \{e' \mid \exists t \in acts(e)^*, e \xrightarrow{t} e'\} \\
&= \{h(\tau_I(e')) \mid \exists t \in acts(\tau_I(e)), \tau_I(e) \xrightarrow{t} \tau_I(e') \\
&= \{h(\tau_I(e')) \mid \tau_I(e') \in states(\mathcal{D}(\tau_I(e)))\} \\
&= h(states(\mathcal{D}(\tau_I(e))))
\end{aligned}$$

$$\begin{aligned}
start(Hide_I(\mathcal{D}(e))) &= start(\mathcal{D}(e)) \\
&= \{e\} \\
&= h(\{\tau_I(e)\}) \\
&= h(start(\mathcal{D}(\tau_I(e))))
\end{aligned}$$

$$\begin{aligned}
sig(Hide_I(\mathcal{D}(e))) &= (in(\mathcal{D}(e)), out(\mathcal{D}(e))\backslash I, int(\mathcal{D}(e)) \cup I) \\
&= (in(e), out(e)\backslash I, int(e) \cup I) \\
&= (in(\tau_I(e)), out(\tau_I(e)), int(\tau_I(e))) \\
&= (in(\mathcal{D}(\tau_I(e))), out(\mathcal{D}(\tau_I(e))), int(\mathcal{D}(\tau_I(e))))
\end{aligned}$$

$$\begin{aligned}
steps(Hide_I(\mathcal{D}(e))) &= steps(\mathcal{D}(e)) \\
&= \{(e', a, e'') \mid e' \in states(\mathcal{D}(e)), e' \xrightarrow{a} e''\} \\
&= \{(h(\tau_I(e')), a, h(\tau_I(e''))) \mid \\
&\qquad \tau_I(e') \in states(\tau_I(e)), \tau_I(e') \xrightarrow{a} \tau_I(e'')\} \\
&= \{(h(\tau_I(e')), a, h(\tau_I(e''))) \mid \\
&\qquad (\tau_I(e'), a, \tau_I(e'')) \in steps(\mathcal{D}(\tau_I(e)))\}
\end{aligned}$$

Table 4. Some axioms for the quiescent preorder of DIOA.

$\mathbf{Ec}_7$	$e \;{}_I{+}_J\; f \sqsubseteq_Q e$ if $Quiet(f)$, $si(e) \subseteq I$ and $si(e) \cap J = \emptyset$
$\mathbf{I}_3$	$\tau_I(a \,.\, e) \equiv_Q a \,.\, \tau_I(e)$ if $a \notin I$
$\mathbf{I}_4$	$\tau_I(e \;{}_H{+}_K\; f) \equiv_Q \tau_I(e) \;{}_H{+}_K\; \tau_I(f)$ if $so(e) \cap I = so(f) \cap I = \emptyset$
$\mathbf{I}_{11}$	$\tau_I(i \,.\, e) \equiv_Q \tau_I(e)$ if $si(e) = \emptyset$

$$\begin{array}{rcl} part(Hide_I(\mathcal{D}(e))) & = & part(\mathcal{D}(e)) \\ & = & \{local(e)\} \\ & = & \{local(\tau_I(e))\} \\ & = & part(\mathcal{D}(\tau_I(e))) \end{array}$$

□

The implementation relation for DIOA is the quiescent preorder, which is a weak congruence for all the operators but the unparameterized $+$. A weak congruence is a relation that is preserved under legal contexts, i.e., $x \,\mathcal{R}\, y$ implies $C[x] \,\mathcal{R}\, C[y]$ if $C[\cdot]$ is a legal context for both x and y. Table 4 contains some axioms for the quiescent preorder over DIOA. The axioms we present are just some of those of [Seg92]; however, they are sufficient for our examples. They are sound in the sense that they state true properties of the I/O automata associated with the expressions. Axiom $\mathbf{Ec}_7$ uses a function $Quiet(f)$ which is true only if f is a quiescent expression, i.e., $\mathcal{D}(f)$ enables only input actions in its start state. The function $Quiet(f)$ depends only on the syntax of f and the declaration mapping E. If E is computable or f is guarded, then $Quiet(f)$ is computable. $\mathbf{Ec}_7$ models the idea that, whenever a specification e does not say anything about some input actions, any choice of implementation f in the presence of those actions is correct. Axiom $\mathbf{I}_3$ allows us to move external actions out of the hiding operator. Axiom $\mathbf{I}_4$ uses a function so in its side condition. Function so (Specified Outputs) is defined in Table 3 and gives those output actions of its argument that can be performed up to internal transitions. The side condition for Axiom $\mathbf{I}_4$ is necessary since an external choice context is not resolved with internal actions (see transition rules $\mathbf{ech}_{4,5}$). Axiom $\mathbf{I}_{11}$ allows us to eliminate initial internal computation from I/O automata whenever no input is expected ($si(e) = \emptyset$). Two other important axioms deal with the parallel operator and with recursion. The expansion axiom permits to unfold a parallel expression into a nondeterministic sequential one; the recursive substitutivity rule states conditions for which a set of equations have unique fixpoint, and gives a method for proving that a process is implementing the fixpoint of a set of equations. In Section 5 the recursive substitutivity rule plays a fundamental role.

Proposition 5.11. The following axiom is sound for the quiescent preorder.

$\mathbf{E}_2$ Let $e \equiv e_1 \parallel e_2 \parallel \cdots \parallel e_n$ where each e_i is of the form $\sum_j a_{ij} \,.\, e_{ij}$. For each action $a \in ext(e)$ let

$$E^i_a = \begin{cases} \{e_{ij} \mid a_{ij} = a\} & \text{if } a \in acts(e_i) \\ \{e_i\} & \text{otherwise} \end{cases}$$

Let $out(a)$ be the index j such that a is an output action of j (0 otherwise) and let

$$E_a = \begin{cases} \emptyset & \text{if } out(a) \neq 0 \text{ and } E^{out(a)}_a = \emptyset \\ \{f_1 \parallel \cdots \parallel f_n : f_i \in E^i_a \vee (E^i_a = \emptyset \wedge f_i \equiv \Omega)\} & \text{otherwise} \end{cases}$$

Then $e \equiv_Q \sum_{a \in ext(e)} (\sum_{f \in E_a} a.f)$.

Theorem 5.12. Let $\tilde{X} \stackrel{\mathrm{def}}{=} \tilde{E}(\tilde{X})$ be a set of equations $\{E_i \stackrel{\mathrm{def}}{=} \sum_j (a_j \,.\, X_{i_j})\}$, and let $\tilde{P}$ be a set of DIOA expressions. If $\tilde{P} \sqsubseteq_Q \tilde{E}[\tilde{P}/\tilde{X}]$ then $\tilde{P} \sqsubseteq_Q \tilde{X}$.

5.2. Specification of the Components

In this section we specify the components of Section 3.2 using DIOA expressions. In this way we can use the DIOA axioms for the actual verification. The new specifications will explicitly consider only specified input actions at each state. The demonic approach guarantees the existence of a transition to Ω for each non-specified input action. The I/O automata of this section differ from those of Section 3.2 in the definition of Ω. Since DIOA deals with finite and quiescent traces only, we need any fair trace of $\mathcal{D}(\Omega)$ to be a quiescent trace of $\mathcal{D}(\Omega)$, i.e., we need $\mathcal{D}(\Omega)$ to be quiescent detectable. Quiescent detectability is obtained through the transition $\Omega \stackrel{\tau}{\longrightarrow} nil$. Note that each sequence of external actions is a fair trace of $\mathcal{D}(\Omega)$; moreover the I/O automata we specify in this section and those of Section 3.2 differ only in the transitions for state Ω. As a consequence the specifications of this section and those of Section 3.2 denote the same objects in the sense that the corresponding I/O automata exhibit the same fair traces. A formal equivalence statement will be given after the specifications.

Specification 5.13. A Muller C element is specified as follows:

$$
\begin{array}{lcl}
C & \stackrel{\mathrm{def}}{=} & a \,.\, C_a + b \,.\, C_b \\
C_a & \stackrel{\mathrm{def}}{=} & a \,.\, C + b \,.\, C_{ab} \\
C_b & \stackrel{\mathrm{def}}{=} & a \,.\, C_{ab} + b \,.\, C \\
C_{ab} & \stackrel{\mathrm{def}}{=} & c \,.\, C
\end{array}
$$

where a, b are input actions and c is an output action.

The DIOA specification of a Muller C element is represented by the process variable C. In order to be consistent with the specifications of the previous sections the process variable name should be C_D, however, we decided to drop the parameter D to avoid confusion with the parameters of the other process variables. The subscripts in the process variables represent the input ports that have changed voltage level. When both the inputs have changed (state C_{ab}) the output voltage level is changed. Note that in state C_{ab} no inputs are accepted. The underspecification of the Muller C element in such cases is implicit in the structure of DIOA. Note that $\mathcal{D}(C)$ has FIN and is input deterministic.

Specification 5.14. A majority element is specified by the following equations

$$
\begin{array}{lcl}
M & \stackrel{\mathrm{def}}{=} & a \,.\, M_a + b \,.\, M_b + c \,.\, M_c \\
M_a & \stackrel{\mathrm{def}}{=} & a \,.\, M + b \,.\, M_{ab} + c \,.\, M_{ac} \\
M_{ab} & \stackrel{\mathrm{def}}{=} & m \,.\, M_c + c \,.\, M_{abc} \\
M_{abc} & \stackrel{\mathrm{def}}{=} & m \,.\, M + a \,.\, M_{bc} + b \,.\, M_{ac} + c \,.\, M_{ab}
\end{array}
$$

where a, b, c are input actions and m is an output action. The equations for

M_b, M_c, M_{ac} and M_{bc} are similar to the equations above and can be easily derived.

The process variable M represents the majority element where the voltage levels of its input ports are the same as the voltage level of its output port. The process variables containing subscripts represent the majority element where only the voltage levels of the input ports not appearing as subscripts are the same as the voltage level of the output port. Note that the equation for M_{ab} specifies that no inputs causing a variation in the output voltage level can occur when the output voltage level already has to change. If such inputs occur then the system implicitly moves to Ω.

Specification 5.15. A wire is specified by the following equation:

$$W \stackrel{\mathrm{def}}{=} m \,.\, c \,.\, W$$

where m is an input action and c is an output action.

Proposition 5.16. $\mathcal{A}(C_N) \equiv_F \mathcal{D}(C)$. $\mathcal{A}(M_N) \equiv_F \mathcal{D}(M)$. $\mathcal{A}(W_N) \equiv_F \mathcal{D}(W)$.

Proof. We prove a stronger equivalence statement, namely that the involved I/O automata are isomorphic if we do not consider states Ω and *nil*. By observing that Proposition 3.5 holds also for I/O automata associated with DIOA expressions (move to *nil* whenever it is possible) we complete the proof. We give the isomorphism for the Muller C element; the isomorphisms for the other elements are given in a similar way and are left to the reader. The isomorphism that we use for the Muller C element is $h : states(\mathcal{A}(C_N)) \to \mathcal{D}(C)$ such that $h(\emptyset) = C$, $h(\{a\}) = C_a$, $h(\{b\}) = C_b$, and $h(\{a, b\}) = C_{ab}$. It is easy to check that h preserves the transitions of $\mathcal{A}(C_N)$ and $\mathcal{D}(C)$ if we do not consider transitions leaving from Ω and transitions from/to *nil*. $\square$

5.3. The Verification

We now formally prove that a Muller C element can be implemented using a majority element and a wire. The implementation relation that we use is the quiescent preorder; however it is easy to verify that all the specified elements satisfy the hypothesis of Theorem 5.7 and Proposition 5.8, therefore we can conclude fair trace inclusion from quiescent trace inclusion. We first prove the statement concerning the quiescent preorder, the DIOA verification; then we show how the formal statement of Section 3.2 is derived.

Proposition 5.17. $\tau_{\{m\}}(M \parallel W) \sqsubseteq_Q C$, i.e., a Muller C element C can be implemented using a majority element and a wire.

Proof. We show that $\tau_{\{m\}}(M \parallel W) \sqsubseteq_Q C$. For doing that we consider a family of processes I, I_a, I_b, I_{ab} where $I \stackrel{\mathrm{def}}{=} \tau_{\{m\}}(M \parallel W)$ and show that they satisfy the equations of C with $\sqsubseteq_Q$. It is then enough to use the recursive substitutivity axiom to conclude.

By applying the expansion axiom and the hiding axioms we obtain

$$
\begin{array}{lll}
I & \equiv_Q & \tau_{\{m\}}(M \| W) \\
 & \equiv_Q & \tau_{\{m\}}((a \,.\, M_a + b \,.\, M_b + c \,.\, M_c) \| (m \,.\, c \,.\, W)) \\
 & \equiv_Q & \tau_{\{m\}}(a \,.\, (M_a \| (m \,.\, c \,.\, W)) + b \,.\, (M_b \| (m \,.\, c \,.\, W))) \\
 & \equiv_Q & \tau_{\{m\}}(a \,.\, (M_a \| W) + b \,.\, (M_b \| W)) \\
 & \equiv_Q & \tau_{\{m\}}(a \,.\, (M_a \| W)) + \tau_{\{m\}}(b \,.\, (M_b \| W)) \\
 & \equiv_Q & a \,.\, \tau_{\{m\}}(M_a \| W) + b \,.\, \tau_{\{m\}}(M_b \| W) \\
 & \equiv_Q & a \,.\, I_a + b \,.\, I_b
\end{array}
$$

where the second step is obtained by expanding $M \| W$, the third step follows from Axiom $\mathbf{E}_2$, the fourth step follows by substituting W for $E(W)$, the fifth step follows from Axiom $\mathbf{I}_4$, the sixth step follows from Axiom $\mathbf{I}_3$, and the seventh step follows by defining

$$I_a \stackrel{\text{def}}{=} \tau_{\{m\}}(M_a \| W) \text{ and}$$

$$I_b \stackrel{\text{def}}{=} \tau_{\{m\}}(M_b \| W).$$

With the same method we have

$$
\begin{array}{lll}
I_a & \equiv_Q & \tau_{\{m\}}(M_a \| W) \\
 & \equiv_Q & a \,.\, \tau_{\{m\}}(M \| W) + b \,.\, \tau_{\{m\}}(M_{ab} \| W) \\
 & \equiv_Q & a \,.\, I + b \,.\, I_{ab}
\end{array}
$$

and

$$
\begin{array}{lll}
I_b & \equiv_Q & \tau_{\{m\}}(M_b \| W) \\
 & \equiv_Q & a \,.\, \tau_{\{m\}}(M_{ab} \| W) + b \,.\, \tau_{\{m\}}(M \| W) \\
 & \equiv_Q & a \,.\, I_{ab} + b \,.\, I
\end{array}
$$

where we define

$$I_{ab} \stackrel{\text{def}}{=} \tau_{\{m\}}(M_{ab} \| W)$$

We now proceed with the analysis of I_{ab}. Step by step comments are below.

$$
\begin{array}{lll}
I_{ab} & \equiv_Q & \tau_{\{m\}}(M_{ab} \| W) \\
 & \equiv_Q & \tau_{\{m\}}(a \,.\, (\Omega \| W) + b \,.\, (\Omega \| W) + m \,.\, (M_c \| c \,.\, W)) \\
 & \sqsubseteq_Q & \tau_{\{m\}}(m \,.\, (M_c \| c \,.\, W)) \\
 & \equiv_Q & \tau_{\{m\}}(m \,.\, (a \,.\, (M_{ac} \| c \,.\, W) + b \,.\, (M_{bc} \| c \,.\, W) + c \,.\, (M \| W))) \\
 & \sqsubseteq_Q & \tau_{\{m\}}(m \,.\, c \,.\, (M \| W)) \\
 & \equiv_Q & c \,.\, \tau_{\{m\}}(M \| W) \\
 & \equiv_Q & c \,.\, I
\end{array}
$$

The first step follows the lines of the previous derivations by expanding process variables, applying the expansion theorem, and reconverting untouched expanded expressions to their corresponding process variable; the second step is an application of Axiom $\mathbf{Ec}_7$ where inputs a and b are eliminated. According to the specification of $C_{a,b}$, in fact, no input should occur before output c occurs. The expression on the second line specifies an implementation choice in the presence of inputs a and b while the expression on the third line does not specify any implementation choice. The third step is similar to the first one while the fourth step consists of successive applications of the hiding axioms. Action m is eliminated through Axiom $\mathbf{I}_{11}$ and action c is brought outside the scope of the hiding operator through Axiom $\mathbf{I}_3$. The last step is a direct consequence of the definition of I.

We can now apply the recursive substitutivity axiom and conclude $\tau_{\{m\}}(M \|$

$W) \sqsubseteq_Q C$. The fair trace inclusion follows from Theorem 5.7 and Proposition 5.8. All the involved I/O automata, in fact, are quiescent detectable, quiescent continuous, input quiescent detectable and have FIN. Moreover no input action disables any output action. □

Theorem 5.18. $Hide_{\{m\}}(\mathcal{A}(M_N) \parallel \mathcal{A}(W_N)) \sqsubseteq_F \mathcal{A}(C_N)$.

Proof. From Proposition 5.17, the soundness of the DIOA proof system, and Theorem 5.7, we derive $\mathcal{D}(\tau_{\{m\}}(M \| W)) \sqsubseteq_F \mathcal{D}(C)$. From Proposition 5.10 we derive $Hide_{\{m\}}(\mathcal{D}(M \| W)) \sqsubseteq_F \mathcal{D}(C)$. From Proposition 5.10 and Proposition 5.8 we have $\mathcal{D}(M) \| \mathcal{D}(W) \sqsubseteq_F \mathcal{D}(M \| W)$, therefore we derive $Hide_{\{m\}}(\mathcal{D}(M) \| \mathcal{D}(W)) \sqsubseteq_F \mathcal{D}(C)$. Finally, from Proposition 5.16 we derive $Hide_{\{m\}}(\mathcal{A}(M_N) \| \mathcal{A}(W_N)) \sqsubseteq_F \mathcal{A}(C_N)$. □

6. Comparison of the Algebraic and the Simulation Techniques

In this section, we compare the simulation and algebraic proof techniques for their usefulness in carrying out verifications of the sort outlined in this paper. The first thing to note is that both of the outlined proofs were fairly easy to carry out, once the machinery described in the "theory" sections had been developed. Naturally, people more familiar with one style of proof or the other will find it somewhat easier to use, but we did not find any appreciable difference for this example. The interesting question is whether both methods will scale equally well to a wide range of more complex examples. Here we think there are important differences and similarities, which we have tried to identify below.

6.1. Correspondence between the Proof Methods

There is a strong similarity between our reasoning in the simulation proof and in the algebraic proof. It seems that the recursive substitutivity rule is used in this example somewhat as an algebraic version of the notion of forward simulation. That is, we consider the process variables of the set of equations comprising the specification as representing states of the specification. Then we consider the processes that we substitute for the process variables as representing states of the implementation that are related to the process variables for which they are substituted.

This leads to the question of whether the simulation and algebraic methods we have used might be equivalent in general; however, it turns out that they are incomparable.

Let a, b, c be output actions and consider the processes

$$X \stackrel{\text{def}}{=} a \,.\, b \,.\, X + a \,.\, c \,.\, X$$

$$Y \stackrel{\text{def}}{=} a \,.\, (b \,.\, Y + c \,.\, Y).$$

It is easy to prove that $Y \sqsubseteq_Q a \,.\, b \,.\, Y + a \,.\, c \,.\, Y$ by using the axioms of [Seg92] and the recursive substitutivity rule; however there is no forward simulation from the transition system associated with Y and that associated with X. State Y, in fact, would be mapped to X. State $b \,.\, Y + c \,.\, Y$, instead, should be mapped to either $b \, . \, X$ or $c \, . \, X$ or both since Y can move with a only to those states. Unfortunately each of the choices above gives problems on the next transition.

The difference between the systems X and Y arises when the decision about whether to perform b or c is made: X decides before Y. A forward simulation between two processes A and B exists only if B does not decide before A. Y can be proved to implement X by using a different simulation technique based on a notion of *backward simulation* [LyV91]. However, there are also examples that can be proved using DIOA deductions but not by backward simulations. One example is

$$X \stackrel{\text{def}}{=} a \, . \, c \, . \, X + b \, . \, Z \qquad Z \stackrel{\text{def}}{=} c \, . \, X$$

$$Y \stackrel{\text{def}}{=} a \, . \, Z' + b \, . \, Z' \qquad Z' \stackrel{\text{def}}{=} c \, . \, Y$$

where a, b and c are output actions. It is easy to show algebraically that Y and Z' satisfy the equations for X and Z; however, there is no backward simulation from Y to X.

There are also cases in which there is a forward simulation between two processes but quiescent trace inclusion cannot be proved using DIOA, because the recursive substitutivity rule cannot be applied. Consider, for example, the processes

$$X \stackrel{\text{def}}{=} a \, . \, X \qquad \text{and} \qquad X_i \stackrel{\text{def}}{=} a \, . \, X_{i+1}$$

for an infinite set of process variables $X_i \, : \, i \in N$. The mapping that maps each X_i into X is trivially a forward simulation from X_0 to X; however, since none of the given equations relates some X_i to X_j with $j \leq i$, we cannot prove that $X_0 \leq a \, . \, X_0$, so the recursive substitutivity rule does not apply. The above mapping is also a backward simulation from X_0 to X, therefore also backward simulation is incomparable with DIOA deduction.

All the examples above also work for the simple trace preorder. The reader is referred to [DnS92] for its axiomatization.

6.2. Treatment of Fairness

In the given example, a separate argument about fairness is made in the simulation proof, whereas no such argument is needed in the algebraic proof. In the given algebraic proof, fair trace inclusion is a consequence of quiescent trace inclusion, and the deductions within DIOA are strong enough to prove quiescent trace inclusion. However, the algebraic framework, as it stands, does not provide a fully general model for proving fair trace inclusion: the connection between the quiescent and fair preorders holds only under some special conditions. We argued in Section 5.1.1 that the properties of quiescent detectability, finite internal nondeterminism and quiescent continuity seem to be sufficiently general for representing physical systems; on the other hand we do not have a clear idea yet about the generality of input quiescent detectability. An example of a non-input quiescent detectable device is an infinite buffer which performs some internal update after receiving some input. An infinite fair execution leading to an infinite trace with input actions only can be obtained by interleaving each input with the internal update; however, if the buffer enables some output whenever it is not empty, no finite sequence of input actions is a quiescent trace.

For systems in which these properties fail, it is still unclear how to use the algebraic approach to reason about fair trace inclusion. It is worth remarking that all the DIOA axioms presented in [Seg92] *except for the recursive substitutivity rule* are sound for the fair preorder as well as the quiescent preorder. (The recursive substitutivity rule is sound for all I/O automata satisfying the conditions of Theorem 5.7.) So if we deal with non-recursive definitions, the axioms for DIOA provide a method for directly proving fair trace inclusion. However, this is of limited use since almost any nontrivial I/O automaton contains loops that have to be specified using recursion. Even our small example cannot be specified without using recursion.

In ACP [BaW90] there is another approach to fairness by means of a rule called Koomen's Fair Abstraction Rule (KFAR). The basic idea for KFAR is that fairness issues can be reformulated in terms of divergences. Thus, a particular infinite execution can be avoided by making all of its actions internal. However, we think this approach is unwieldy; for example the requirement that each one of two independent automata provides some output (an external action) infinitely often requires a conspicuous amount of encoding, making a specification difficult to read.

It is also unlikely that a result similar to the Execution Correspondence Lemma could be used together with an algebraic proof. Even by axiomatizing a different preorder relation such as "existence of a forward simulation", an algebraic proof would prove the existence of a simulation without exhibiting it. The fairness part of our simulation proof, on the other hand, is strongly based on the actual forward simulation from the implementation to the specification. The simple knowledge that a forward simulation exists is not sufficient. It is possible that new techniques, perhaps based on the structure of an algebraic proof, could be developed, but this remains to be done.

The generality of our approach to fairness in the simulation proof also remains to be considered; however, in this case there is already good evidence that this approach works well in practice [LaS92, SLL93b]. The approach based on the Execution Correspondence Lemma provides a convenient way to base a fairness proof on a simulation proof; it may be that there are some fairness proofs that are inherently unable to be split in this way, but we do not know of any such examples. The use of forcing conditions provides a useful generalization of the usual I/O automaton fairness notion, but it seems likely to us that further generalizations will be required in order to describe some realistic liveness requirements. What those extensions might be, and whether they will work well in conjunction with the Execution Correspondence Lemma, remain to be seen.

Note that the arguments of this subsection hold only for fairness sensitive semantics such as the semantics of I/O automata. If the semantics is based on relations like bisimulation [Mil89] or testing [DnH84, Hen88], then the problems of this subsection disappear.

6.3. Representation of Automata

The two different proof methods typically use very different ways of representing automata, each best suited for carrying out the corresponding type of proof. In order to give a fair comparison between the two methods, we began with a neutral representation, which is basically just a state-transition table that enumerates the results of all transitions performed in all states. We then gave two other representation methods, and asserted their equivalence with the neutral method.

The precondition-effect language represents an automaton in an *action-based* way. That is, the information associated with each *action* is given in one place; this information consists of the set of enabling states and the allowed transitions for that action. In terms of the neutral representation, we can think of this language as presenting the automaton by *columns*.

On the other hand, DIOA represents an automaton in a *state-based* way. That is, the information associated with each *state* is given in one expression; this information consists of a list of the enabled transitions from that state. We can think of this language as presenting the neutral automaton by *rows*.

In our small example, the state-based method gives a more elegant and concise representation of the circuits than the action-based method, but this will not be true in general. The choice of which representation is better will vary among different automata, depending upon whether the automaton table is most easily described by columns or by rows. Our experience shows that, for complex systems, the action-based description is usually the better one [SLL93b].

There is one main reason for this. The states of a complex automaton can usually be described in terms of a small number of state variables or data objects, which permits a description to be parameterized by the values of those objects. A typical complex automaton exhibits *locality of activity*: each action typically involves only a small portion of the state, i.e., its occurrence depends on the values of a small number of data objects, and its results affect only a small number of objects. This locality leads to concise descriptions for each action, but it is unclear how a state-based description might take advantage of it. Note that parallel decomposition cannot be used in general to describe this kind of locality.

Although the action-based representation method generally works better than the state-based one, there is complete freedom in the choice of the representation style for an I/O automaton whenever a simulation proof technique is used, i.e., it is always possible to use a description language like the state-based one in conjunction with assertional reasoning. On the other hand the description language for DIOA is strictly determined by the algebra itself, so there is apparently no way to use an action-based representation method in process algebras. Moreover, the pure DIOA calculus does not provide tools to deal with structured states.

A standard technique to deal with structured states within process algebras makes use of parameterized process variables [Hoa85, Mil89, Bae90]. For example, a counter can be represented by a process variable X parameterized over a natural number n in the following way:

$$
\begin{aligned}
&X_0 \stackrel{\mathrm{def}}{=} up \,.\, X_1 \\
&X_n \stackrel{\mathrm{def}}{=} down \,.\, X_{n-1} + up \,.\, X_{n+1} \quad \text{if } n > 0.
\end{aligned}
$$

Such a technique is generally used when the size of a system is large [Bae90, OrP92] since a specification would become unreadable otherwise. Our example, although small, makes use of parameters. It is also possible to add standard programming languages constructs and define a new equation of the form

$$
X_n \stackrel{\mathrm{def}}{=} up \,.\, X_{n+1} + (\text{if } n > 0 \text{ then } down \,.\, X_{n-1}).
$$

By means of the above ideas it is possible to directly encode an action-based represented automaton A into DIOA. The encoding consists of one process variable X parameterized over $states(A)$. The equation for X is then of the form

if $precondition(a_1)$ then $effect(a_1)$ else
if $precondition(a_2)$ then $effect(a_2)$ else $\cdots$

Unfortunately, the more structure we add to the algebraic notation, the more complicated it is to apply the DIOA axioms to carry out a proof. Also, the recursive substitutivity rule requires one to find a set of processes that satisfy a given set of inequations. When states are parameterized, finding those processes is often tantamount to finding a simulation relation between states of the implementation and states of the specification, which is consistent with the initial observation of Section 6.1. In this case, the task of applying the axioms becomes the equivalent of proving that a given simulation is a forward simulation. For example, consider the counter we specified before and consider an implementation as follows:

$$Y_{10} \stackrel{\text{def}}{=} up \,.\, Y_{11}$$
$$Y_n \stackrel{\text{def}}{=} down \,.\, Y_{n-1} + up \,.\, Y_{n+1} \quad \text{if } n > 10.$$

The recursive substitutivity rule requires us to show that each Y_i satisfies the equation for X_{i-10}. The association $h : Y_i \mapsto X_{i-10}$ is a sort of simulation, and the algebraic proof shows its correctness.

6.4. Mechanization

The process of carrying out either a simulation proof or an algebraic proof can be long and tedious, and therefore error-prone, when the involved automata are large. A simulation proof typically involves a case analysis based on actions; each case involves logical deduction based on descriptions of the state transitions in both the implementation and specification automata and on a description of the forward (or other kind of) simulation relation. An algebraic proof involves a series of deductions using the algebraic axioms. In both cases, it should be possible to check the correctness of the deduction steps using an automatic prover. However, we would also like some help from an automatic prover in actually carrying out these tedious steps.

An automatic prover can help in the production of a simulation proof, but we do not expect that the proof process will be completely automatic since the problem is undecidable in general. In addition to descriptions of the two automata, the writer of such a proof will have to provide a description of the simulation relation and possibly some invariances. Once this information is provided, an automatic prover can be used to help in filling in enough details to verify that the simulation is correct. As described in [SGG$^+$93], the Larch prover has been used successfully for this purpose. Also the theorem prover Isabelle was used for the same purpose in [Nip89], and the proof assistant Coq (Calculus of Inductive Constructions) was used in [HSV94]. The work on mechanical simulation-based verifications is still under development, and [HSV94, Nip89, SGG$^+$93] are just the first attempts at solving the problem.

It seems unlikely that an automatic prover will be of much help in defining the simulation relation in a simulation proof. In small cases, essentially when there are finitely many states as in our example, a model-checking approach might be helpful. The task of defining the simulation relation by hand will often not be easy; its difficulty is comparable to that of defining an invariant assertion. However, usually the designer of a system has enough intuitions about the design to be able to define a relation that is almost correct, and this can be used as a starting point for constructing the correct relation.

In the process algebraic proof given in this paper the axioms that have to be applied during each step are partially determined by the equations defining the specification automaton. Our proof steps were essentially repeated applications of the expansion axiom followed by some simplifications based on the given specification. This heuristic is generally applicable when dealing with (finite state) circuit descriptions. It is also applied in [Jos92, Seg92, OrP92] and in several of the examples of [Bae90]. In these cases, algebraic manipulators like those of [MaV91, Lin91] can be used. However, when the problem becomes large or is described by an infinite state machine, the remarks at the end of Section 6.3 show that some form of simulation has to be defined even for an algebraic proof; therefore, the difficulties involved in the mechanization of simulation and algebraic proofs are comparable. A case study in [GrP93], which is an algebraic verification of the same protocol as [HSV94] using Coq, shows, in our opinion, how deriving a process algebraic proof for a large system is tantamount to finding a simulation relation. Yet, in [HSV94] the authors relate their work to [GrP93] by commenting that "whether one prefers process algebra or the I/O automata model appears to be a matter of taste".

6.5. Additional Benefits Obtained from the Proof

Experience with large simulation-based verifications [WLL88, LyP92, SLL93b] has shown that the formal description of the simulation relation in a simulation proof constitutes an important piece of documentation of the key ideas of the implementation, in much the same way that an invariant assertion does; invariants and simulations typically express the key intuitions that make the implementation work. Similarly, due to the remarks at the end of Section 6.3, an algebraic proof can embed some form of mapping which can be used as a documentation.

Because of the Execution Correspondence Lemma, a simulation-based proof provides a correspondence between executions rather than just trace inclusion. This correspondence enables us, for example, to base proofs of fairness on proofs of ordinary trace inclusion. A process algebraic proof, on the other hand, proves only the properties for which the axioms are certified to be sound. In our example we were able to prove liveness because the quiescent preorder coincides with the fair preorder under some particular conditions; however, if those conditions are not met, or if we need to prove other properties (e.g., based on forcing sets) the algebraic proof provides no help.

In our experience simulation proofs are flexible in the sense that a given proof can usually be modified fairly easily in order to verify new properties of an implementation. A typical verification task, for example the one in [SLL93b], involves the definition of specification and implementation automata and the proof that the implementation meets the specification. During the proof some errors might be discovered and the involved automata might need to be modified. Also, after the proof is completed, the specification and/or implementation automata might be slightly modified in order to make them cleaner and more general. The simulation relation and the correctness proof might then have to be correspondingly modified. In general the structure of the simulation proof seems to provide us with a lot of guidance in carrying out such modifications, since its general structure is usually preserved. To the extent that an algebraic proof embeds a simulation proof, the same advantages for modifiability would accrue.

7. Conclusion

Using a simple example based on delay insensitive circuits, we have compared two widely used verification techniques for concurrent and distributed systems. The assertional methods based on I/O automata have been successfully used for the verification of very complex systems [LyT87, WLL88, LyP92, SLL93b] while the algebraic techniques of process algebras [Mil89] have generally been used for relatively small examples [Bae90, Jos92].

We have verified the correctness of the implementation of a Muller C element taken from [Jos92] both in the assertional framework and in the process algebraic framework. The algebraic proof is based on DIOA [Seg92], a process algebra for I/O automata.

The example we have used is one of the typical examples of the process algebraic community; therefore, it should not be surprising that the process algebraic analysis looks shorter than the simulation-based one. Starting from the presented example, however, our discussion has shown that scaling simulation and algebraic proofs to more complex systems leads to comparable verification processes, both in terms of verification steps and difficulty.

Although we have emphasized verification in this paper, it is important to remember that verification is not the only purpose, nor even the main purpose, of process algebra. Rather, process algebra is intended to provide compositional semantics for programs. Of course, one important use for such a semantics is to provide a basis for carrying out formal correctness proofs for systems. Since one of the most practical verification methods is simulation, it is important that an algebraic semantics be designed with a view toward compatibility with simulation proofs. Given a program that is supposed to implement a given specification, a process algebraic characterization of the semantic model can be used to compute compositionally the semantics of the given program; then a simulation-based technique can be used to prove the correctness of the implementation. Perhaps, we could also add an intermediate step where algebraic techniques are used to simplify a system before starting with the assertional part of the correctness proof.

Acknowledgments. We thank Rocco De Nicola, Alan Fekete, John Guttag, Jørgen Søgaard-Andersen, Frits Vaandrager and Jeannette Wing for very useful comments and criticism on previous versions of this paper. We also thank one of the anonymous referees for suggestions in the comparison section.

References

[Bae90] J.C.M. Baeten. *Applications of Process Algebra*. Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

[BaW90] J.C.M. Baeten and W.P Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[DnH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[DnS92] R. De Nicola and R. Segala. A process algebraic view of I/O automata. Technical Report SI-92/05, Dipartimento di Scienze dell'Informazione, Università degli studi di Roma La Sapienza, September 1992.

[GSSL93] R. Gawlick, R. Segala, J.F. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, MIT Laboratory for Computer Science, November 1993.

[GrP93] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. A case study in computer checked verification. Technical Report 100, Logic Group Preprint Series, Utrecht University, 1993.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Massachusetts, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[HSV94] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. Technical Report CS-R9420, Programming Research Group, University of Amsterdam, 1994.

[Jos92] M.B. Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.

[Lin91] H. Lin. PAM: A Process Algebra Manipulator. In Larsen and Skou [LaS91], pages 136–146.

[LyP92] N. Lynch and B. Patt-Shamir. Distributed Algorithms. Fall 1992 Lecture Notes for 6.852. MIT/LCS/RSS 16, MIT Laboratory for Computer Science, 1992.

[LaS91] K.G. Larsen and A. Skou, editors. *Proceedings of the third international workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[LaS92] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, September 1992.

[LyT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, Canada, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[LyV91] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations for timing-based systems. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446. Springer-Verlag, 1991.

[MuB59] D.E. Muller and W.S. Bartky. A theory of asynchronous circuits. *Annals of the Computation Laboratory of Harvard University. Volume XXIX: Proceedings of an International Symposium on the Theory of Switching, Part I*, pages 204–243, 1959.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[MaV91] S. Mauw and G.J. Veltink. A proof assistant for PSF. In Larsen and Skou [LaS91], pages 158–168.

[Nip89] T. Nipkow. Formal verification of data type refinement - theory and practice. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop "Stepwise Refinement of Distributed Systems"*, volume 430 of *Lecture Notes in Computer Science*, pages 561–591. Springer-Verlag, 1989.

[OrP92] F. Orava and J. Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4:497–593, 1992.

[Seg92] R. Segala. A process algebraic view of I/O automata. Technical Memo MIT/LCS/TR-557, MIT Laboratory for Computer Science, Cambridge, MA 02139, October 1992.

[Seg93] R. Segala. Quiescence, fairness, testing and the notion of implementation. In E. Best, editor, *Proceedings CONCUR 93,* Hildesheim, Germany, volume 715 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[SGG+93] J.F. Søgaard-Andersen, S.J. Garland, J.V. Guttag, N.A. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In C. Courcoubetis, editor, *Proceedings of the fifth international conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[SLL93a] J.F. Søgaard-Andersen, B. Lampson, and N.A. Lynch. Correctness of at-most-once message delivery protocols. In *FORTE '93 - Sixth International Conference on Formal Description Techniques*, 1993.

[SLL93b] J.F. Søgaard-Andersen, N.A. Lynch, and B.W. Lampson. Correctness of communication protocols. a case study. Technical Report MIT/LCS/TR-589, MIT Laboratory for Computer Science, November 1993.

[Sta84] E.W. Stark. *Foundations of a theory of specification for Distributed Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, August 1984. Available as Technical Report MIT/LCS/TR-342.

[Sta90] E.W. Stark. On the relations computable by a class of concurrent automata. In *Proceedings of the 1990 SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1990.

[Vaa91] F.W. Vaandrager. On the relationship between process algebra and Input/Output automata. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, 1991.

[WLL88] J.L. Welch, L. Lamport, and N. Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm. Technical Report MIT/LCS/TM-361, MIT Laboratory for Computer Science, June 1988.