On the Relationship Between Process Algebra and Input/Output Automata*

Frits W. Vaandrager
MIT Laboratory for Computer Science
Cambridge, MA 02139, USA
frits@theory.lcs.mit.edu

Abstract

The relation between process algebra and I/O automata models is investigated in a general setting of structured operational semantics (SOS). For a series of (approximations of) key properties of I/O automata, syntactic constraints on inference rules are proposed which guarantee these properties. A first result is that, in a setting without assumptions about actions, the well-known trace and failure preorders are substitutive for any set of rules in a format due to De Simone. Next additional constraints are imposed which capture the notion of *internal actions* and guarantee substitutivity of the testing preorders of De Nicola and Hennessy, and also of a preorder related to the failure semantics with fair abstraction of unstable divergence of Bergstra, Klop and Olderog. Subsequent constraints guarantee that *input actions* are always enabled and *output actions* cannot be blocked, two key features of input/output automata. The main result is that for any *I/O calculus*, i.e. a De Simone calculus which combines the constraints for internal, input and output actions, the quiescent trace preorder and the fair trace preorder are substitutive. A simple I/O calculus is presented which is sufficiently expressive to specify all finitely branching I/O automata over a given, finite action signature.

*This work was supported by ONR contract N00014-85-K-0168.

1 Introduction

This paper studies the relationship between the algebraic theory of processes as described in [2, 14, 15, 21] and the input/output automata models of [16, 17, 19, 20, 26]. Apparently, these two formalisms for concurrent/reactive systems, process algebra and I/O models for short, have been developed on different planets. The notion of fairness plays an important role in I/O models but despite several attempts [9, 22] there has been no satisfactory treatment in process algebra thus far. Process algebra on the one hand, offers an abundance of operators, a rich algebraic structure and a well-developed fixed point theory. In the I/O models on the other hand, there are at most three operators (composition, hiding and action renaming) and verification takes place primarily on the level of the model using assertional proof techniques.

The most fundamental difference between I/O models and process algebra, and the main topic of this paper, is the distinction between input and output actions, which is crucial in I/O models but absent (on the semantical level) in process algebra. In I/O models internal and output actions are under the control of an automaton itself and input actions are under the control of the environment. Because the environment decides to provide the automaton with input, the automaton should be willing to receive every possible input in every possible state. Of course the automaton is free to decide what to do with these inputs and may discard them immediately. Reversely, since the automaton is under control of its internal and output actions, the environment is not allowed to prevent these actions from occurring. So I/O models rule out actions which are under control of *both* a process and its environment. Some proponents of I/O models argue that this is not a restriction at all because, as they say, such actions do not exist "in reality". But their argument is not convincing. Typical examples of actions which are controlled both by a process and its environment are the $c?x$ and $c!e$ actions in languages like CSP [15]. If one takes a closer look at how this kind of actions are implemented on physical machines, then one discovers that indeed this is done by means of some handshaking protocol which involves actions that are either under control of an agent or its environment. However, this does not invalidate the observation that on a higher level of abstraction actions occur which *are* controlled by both a process and its environment, and that many people find it useful to reason about concurrent systems in terms of these actions. In process algebras like CSP [15], CCS [21], ACP [2] and MEIJE [25] the concept of actions in which more than one agent participates is carried to its extreme: besides internal actions, which do not communicate, *all* actions are under control of both the agent and its environment. Apparently there is room for a formalism that combines the different notions of actions present in I/O models and CSP like models. In this paper I will not attempt to develop such a formalism, but modestly try to understand the notions of input and output actions first.

A major advantage of having just input and output actions seems to be that there is a compositional linear (trace) semantics which is compatible with the operational model, and that, unlike in algebraic process theory, there is no need for more sophisticated notions of equivalence like bisimulation or failure congruence. Another major advantage associated to the I/O postulate is that it becomes much easier to state and prove liveness properties. In calculi like CSP it is somehow problematic to say that "a process p will eventually do an action a" since the environment may prevent the occurrence of a or some other action which p needs to do before the a. Although in theory it is still possible to state liveness properties, the formulations of these properties tend to become complex and incomprehensible in practice. Because in I/O models processes are in charge of their own actions, liveness properties (and fairness properties in particular) are much easier to handle.

Disadvantages of the I/O discipline are that certain transition systems (the ones that are not input enabled) are not considered anymore, and that a large number of operators (the ones that either do not preserve input enabling or block output actions) can no longer be used. When confronted with the postulate of input enabling, adherents of algebraic process theory protest "Why am I no longer allowed to consider a 1-datum buffer which simply does not accept a new datum when it is full?" and "What is wrong with operations like action prefixing, which play such a crucial role in axiomatizing process equivalences?" and "Should I really be happy with an approach that offers no possibility to build automata from scratch and has basically no algebra?" And indeed, because prefixing operators play such an essential role in algebraic process theory, the question arises whether in their absence it is still possible to have a nice algebra of I/O automata. This paper provides a positive answer to this question.

Before presenting a particular algebra of I/O automata, a whole class of operations on I/O automata will be identified in a setting of Structured Operational Semantics (SOS) in the style of Plotkin [24]. The traditional approach to SOS is to start from a particular calculus (i.e. a language together with a set of inference rules) and next derive the properties of the operational semantics induced by this calculus [14, 21]. Recent work [6, 7, 13, 27] however shows that very strong results can be obtained if one takes the opposite approach which starts from a desired property, and then looks for the weakest syntactic constraints on calculi which guarantee this property. A specific calculus that does not satisfy the syntactic constraints still may have the property; however, counterexamples show that it is not obvious how to relax the syntactic constraints any further. In this paper, I look for the weakest syntactic constraints on calculi which guarantee that the resulting automaton is input enabled, and that moreover the semantics induced by taking fair (infinite) traces is compositional. Once these constraints have been found, it is not so difficult to come up with an interesting algebra for I/O automata.

Because the property of having input enabling and compositionality for fair traces is rather complex, I first study syntactic constraints induced by a number of simpler properties. This approach also has the advantage that it leads to several new and very general results about process algebras. After a preliminary section, Section 3 deals with substitutivity of the preorder induced by taking finite traces in a setting without special types of actions. The resulting format of rules turns out to be essentially a format introduced by De Simone in [25]. Section 3 also establishes that for any calculus in this format, the well-known failure preorder is a substitutive. These results generalize and somehow explain similar results that have been obtained previously for a large number of individual calculi in De Simone's format (see for instance [10, 4]).

Section 4 studies certain trace and failure preorders in the presence of internal actions. The main results of this section are that the testing preorders of [11, 14] and also a preorder that is related to the failure semantics with fair abstraction of unstable divergence of [3] are substitutive for any De Simone calculus that satisfies certain additional constraints. These simple constraints explain very nicely why certain preorders are compositional for the operators of CSP, but not for all of the operators of CCS and ACP.

Section 5 starts with two simple constraints on De Simone calculi which together guarantee input enabling. Next the notion of a (finite) quiescent trace is introduced and it is shown that the corresponding preorder is substitutive for any *I/O calculus*, i.e. a De Simone calculus which satisfies the constraints for internal actions and input enabling, together with a constraint which essentially says that output actions cannot be blocked. It turns out that also the preorder induced by taking (infinite) fair traces is substitutive for I/O calculi. Finally, Section 5 presents a simple I/O calculus which is sufficiently expressive to specify all finitely branching I/O automata over a given, finite action signature.

Sections 6 contains some concluding remarks.

Acknowledgements

Many thanks to Martín Abadi, Nancy Lynch, Albert Meyer, Boris Trakhtenbrot and Mark Tuttle for inspiring discussions and useful comments.

2 Preliminaries

Definition 2.1 *(Notation for sequences).* Let K be any set. The sets of finite and infinite sequences of elements of K are denoted by K^* resp. K^ω. K^∞ denotes the union of K^* and K^ω. Concatenation of a finite sequence with a finite or infinite sequence is denoted by juxtaposition; λ denotes the empty sequence and the sequence containing one element $a \in K$ is denoted a. If σ is a sequence over K and $L \subseteq K$, then $\sigma\lceil L$ denotes the sequence obtained by projecting σ to L. If S is a set of sequences, $S\lceil L$ is defined as $\{\sigma\lceil L \mid \sigma \in S\}$.

Definition 2.2 *(Signatures and terms).* To start with, there is a countably infinite set $\mathcal{V}$ of *variables* with typical elements $x, y, \ldots$. A *signature element* is a pair (f, n) consisting of a *function symbol* $f \notin \mathcal{V}$ and an *arity* $n \in \mathbb{N}$. In a signature element $(c, 0)$, the c is often referred to as a *constant symbol.* A *signature* is a set of signature elements. We write $\mathbb{T}(\Sigma)$ for the set of *(open) terms* over signature Σ with variables from $\mathcal{V}$. $T(\Sigma)$ is the set of *closed* terms over Σ, i.e. terms in $\mathbb{T}(\Sigma)$ that do not contain variables. With $var(t)$ the set of variables occurring in a term t is denoted. A *substitution* ζ is a mapping from $\mathcal{V}$ to $\mathbb{T}(\Sigma)$. Term $t[\zeta]$ is the result of the simultaneous substitution, for all x in t, of x by $\zeta(x)$. The expression $t[t_1/x_1, \ldots, t_n/x_n]$ denotes the term obtained from t by simultaneous substitution of t_1 for x_1, t_2 for x_2, etc.

Definition 2.3 *(Contexts).* Let Σ be a signature. A *context of n holes* C over Σ is simply a term in $\mathbb{T}(\Sigma)$ in which n variables occur, each variable only once. If $t_1, \ldots, t_n$ are terms over Σ, then $C[t_1, \ldots, t_n]$ denotes the term obtained by substituting t_1 for the first variable occurring in C, t_2 for the second variable, etc. Thus, if $x_1, \ldots, x_n$ are all different variables, $C[x_1, \ldots, x_n]$, denotes a context of n holes in which x_i is the i-th variable that occurs. A preorder (i.e. a transitive and reflexive relation) $\sqsubseteq$ on $T(\Sigma)$ is *substitutive* if for all contexts $C[x]$: $t \sqsubseteq t'$ implies $C[t] \sqsubseteq C[t']$.

The intersection of substitutive preorders is again substitutive. In particular, if a preorder $\sqsubseteq$ is substitutive, then its *kernel* (the equivalence relation defined by $t \equiv u$ iff $t \sqsubseteq u$ and $u \sqsubseteq t$) is substitutive, and hence, by definition, a congruence.

Definition 2.4 *(Calculi).* Let A be a given set of *labels* and let Σ be a signature. The set $Tr(\Sigma, A)$ of *transitions* consists of all expressions of the form $t \xrightarrow{a} t'$ with $t, t' \in \mathbb{T}(\Sigma)$ and $a \in A$. The set $Cf(\Sigma, A)$ of *inference rules* or *conditional formulas* over Σ and A consists of all expressions

$$\frac{\psi_1, \ldots, \psi_n}{\psi},$$

where $\psi_1, \ldots, \psi_n, \psi$ in $Tr(\Sigma, A)$. The transitions ψ_i are called the *premises* and ψ is called the *conclusion* of the rule. If no confusion can arise, a rule $\frac{}{\psi}$ is also written ψ. The notions "substitution" and "closed" extend to transitions and rules in the obvious way.
A *transition system specification* or *calculus* is a triple $P = (\Sigma, A, R)$ with Σ a signature, A a set of labels and $R \subseteq Cf(\Sigma, A)$ a set of inference rules.

Definition 2.5 *(Proof trees).* Let $P = (\Sigma, A, R)$ be a calculus. A *proof tree* from P for a transition ψ is a finite tree whose edges are ordered and whose vertices are labeled by transitions in $Tr(\Sigma, A)$, such that:

- the root is labeled with ψ,
- if ϕ is the label of a node q and $\phi_1, \ldots, \phi_n$ are the labels of the children of q, then there is a rule $\frac{\phi_1', \ldots, \phi_n'}{\phi'}$ in R and a substitution ζ such that $\phi_i = \phi_i'[\zeta]$ and $\phi = \phi'[\zeta]$.

If a proof tree for ψ exists, then ψ is *provable* from P, notation $P \vdash \psi$. The set *enabled*(t) of actions which

δ	0	inaction
$a\cdot$	1	prefixing; for each $a \in A$
$+$	2	choice, sum
$\|\|$	2	parallel, (free) merge
$\times$	2	synchronous composition, product
φ	1	renaming; for each $\varphi : A \longrightarrow A$
X	0	process names; for each $X \in \mathcal{X}$

Table 1: The signature of PC.

are *enabled* in a term t is defined by

$$enabled(t) \triangleq \{a \in A \mid \exists t' : P \vdash t \xrightarrow{a} t'\}.$$

If $t, t' \in T(\Sigma)$ and $\sigma = a_1 \cdots a_n \in A^*$, write $P \vdash t \xrightarrow{\sigma} t'$ if there are terms $t_0, \ldots, t_n$ such that $t = t_0$, for all $i < n$: $P \vdash t_i \xrightarrow{a_{i+1}} t_{i+1}$ and $t_n = t'$.

Example 2.6. As a simple example of a calculus in the style of CCS and ACP, consider the Process Calculus *PC*. PC is parametrized by a finite set A of *actions*, ranged over by $a, b, \ldots$, and a set $\mathcal{X}$ of *process names*, ranged over by $X, Y, \ldots$. The signature Σ_{PC} of PC is displayed in Table 1. Infix notation will be used for the binary function symbols, and I write $a \cdot p$ instead of $a \cdot (p)$. To avoid parentheses, it is assumed that prefixing has most binding power, followed by product, which in turn is followed by free merge, which is followed by alternative composition (which has the weakest binding power). In the case of several sum, merge or product operations I will mostly omit brackets since semantically these operations are associative. For a finite index set $I = \{i_1, \ldots, i_n\}$ and PC terms $p_{i_1}, \ldots, p_{i_n}$, $\sum_{i \in I} p_i$ abbreviates $p_{i_1} + \ldots + p_{i_n}$. By convention $\sum_{i \in \emptyset}$ stands for δ. Trailing δ's will often be dropped.

Intuitively, δ denotes *inaction*, a process that does not do anything at all. The process $a \cdot p$ first performs an a-action and then behaves like p. Process $p + q$ will behave either like p or like q. It is not specified whether the choice between p and q is made by the process itself or by the environment. The term $p \| q$ denotes the parallel composition of p and q without synchronization. The term $p \times q$ denotes the parallel composition of p and q in which *all* actions must synchronize. Process $\varphi(p)$ behaves just like process p, except that the actions of p are renamed according to φ. The recursive definitions of the process names are given by a declaration function $E : \mathcal{X} \longrightarrow T(\Sigma_{PC})$. So the behavior of a process name X is specified recursively by $E(X)$. Only *guarded* recursion is permitted: in terms $E(X)$ process names are only allowed to occur in the scope of a prefixing operator.

The inference rules which define the operational semantics of PC are presented in Table 2. In the table a and b range over A, X over $\mathcal{X}$, and φ over $A \longrightarrow A$. The variables x, x', y and y' are fixed and all different.

$$a \cdot x \xrightarrow{a} x$$

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$$

$$\frac{x \xrightarrow{a} x'}{x \| y \xrightarrow{a} x' \| y} \qquad \frac{y \xrightarrow{a} y'}{x \| y \xrightarrow{a} x \| y'}$$

$$\frac{x \xrightarrow{a} x' \;,\; y \xrightarrow{a} y'}{x \times y \xrightarrow{a} x' \times y'} \qquad \frac{x \xrightarrow{a} x'}{\varphi(x) \xrightarrow{\varphi(a)} \varphi(x')}$$

$$\frac{E(X) \xrightarrow{a} y}{X \xrightarrow{a} y}$$

Table 2: The inference rules of PC.

3 Trace and Failure Preorders: the General Case

Before studying trace preorders in a setting with input, output and internal actions, I first consider the simple case in which no special assumptions are made about the nature of actions, and study substitutivity of preorders which are based on finite *traces*, i.e. sequences of actions that are enabled from a given state.

Fix a calculus $P = (\Sigma, A, R)$.

Definition 3.1 *(Trace preorder).* The set of (finite) *traces* of a term $t \in T(\Sigma)$ is

$$traces(t) \triangleq \{\sigma \in A^* \mid \exists t' : P \vdash t \xrightarrow{\sigma} t'\}.$$

The *trace preorder* $\sqsubseteq_T$ on $T(\Sigma)$ is defined by: $t \sqsubseteq_T t'$ iff $traces(t) \subseteq traces(t')$.

The requirement that the trace preorder $\sqsubseteq_T$ is substitutive excludes a lot of calculi. In fact, all the counterexamples that are presented in [13] to show that strong bisimulation equivalence is not substitutive for certain calculi can be used again. Two of these counterexamples will be recalled here.

The first example illustrates that substitutivity is often endangered if a variable occurs more than once in the left-hand-side of the conclusion of a rule: if one adds to PC a rule $x + x \xrightarrow{b} \delta$, then $a \sqsubseteq_T a \| \delta$ but $a + a \not\sqsubseteq_T a + (a \| \delta)$.

The second example shows that problems can arise if function symbols occur in the right-hand-side of a premis: if one adds to PC a rule

$$\frac{x \xrightarrow{a} \delta}{a \cdot x \xrightarrow{b} \delta},$$

then $\delta \sqsubseteq_T \delta \| \delta$ but $a \cdot a \cdot \delta \not\sqsubseteq_T a \cdot a \cdot (\delta \| \delta)$.

Basically, the counterexamples of [13] show that problems can be expected if the rules do not fit the *tyft/tyxt* format introduced in that paper. However, the following counterexamples show that the trace preorder is not substitutive for all calculi in *tyft/tyxt* format.

In general it is not allowed to test whether, from a certain state, more than one action is enabled: if one adds to PC rules

$$\frac{x \xrightarrow{a} y}{B(x) \xrightarrow{a} B(y)} \text{ and } \frac{x \xrightarrow{b} y,\ x \xrightarrow{c} z}{B(x) \xrightarrow{a} \delta},$$

then $a(b+c) \sqsubseteq_T ab+ac$ but $B(a(b+c)) \not\sqsubseteq_T B(ab+ac)$.

Also rules with *lookahead* cause problems: if one adds to PC rules

$$\frac{x \xrightarrow{a} y,\ y \xrightarrow{a} z}{L(x) \xrightarrow{a} L(y)} \text{ and } \frac{x \xrightarrow{b} y}{L(x) \xrightarrow{b} L(y)},$$

then $a(a+b) \sqsubseteq_T aa+ab$ but $L(a(a+b)) \not\sqsubseteq_T L(aa+ab)$.

The search for simple constraints on inference rules which guarantee that the trace preorder is substitutive, leads to a format of rules that is essentially due to De Simone [25]. For any obvious relaxation of the constraints imposed by this format a counterexample in the style of the ones presented above can be found. Without doubt it is possible to come up with a format which is more general than the format of De Simone such that the trace preorder is still substitutive, but the definition of such a format will be more complex.

In the definition below I use the notion of *stuttering steps*. These stuttering steps are technically convenient but not essential and all the results of this paper carry over in a trivial way to a setting without them. Remark 3.3 comments on this issue in more detail.

Definition 3.2 *(De Simone calculi).* Let $\{x_i \mid i \in \mathbb{N}\}$ and $\{y_j \mid j \in \mathbb{N}\}$ be two fixed sets of variables in $\mathcal{V}$ with all x_i and y_j different. Let Σ be a signature, A a set of labels, and $*$ a special label, not in A, denoting *stuttering* or *idling*.

A rule in $Cf(\Sigma, A \cup \{*\})$ is a *De Simone* rule if it takes the form

$$\frac{\{x_i \xrightarrow{a_i} y_i \mid 1 \leq i \leq n\}}{f(x_1, \ldots, x_n) \xrightarrow{a} t}$$

where $(f, n) \in \Sigma$, $a \in A$ and for all i, $a_i \in A \cup \{*\}$, and $t \in \mathbb{T}(\Sigma)$ is a context with variables in $\{y_1, \ldots, y_n\}$ (so variables occur linearly in t). In the above rule, (f, n) is the *type*, a the *action*, t the *target*, and the tuple $\langle a_1, \ldots, a_n \rangle$ is the *trigger*. If $a_i \in A$, then the i-th position is *active* in the rule; otherwise it is *passive*. Each rule is characterized uniquely by its type, action, target and trigger. If r is a rule, these ingredients will be referred to as *type*(r), *action*(r), *target*(r) and *trigger*(r).

A calculus $P = (\Sigma, A \cup \{*\}, R)$ is a *De Simone calculus* if Σ can be partitioned into Σ_1 and Σ_2, and R can be partitioned into R_1, R_2 and R_3 in such a way that

- all the rules in R_1 are De Simone rules with a type in Σ_1;
- there exists a set $\mathcal{X}$ and a mapping $E : \mathcal{X} \rightarrow T(\Sigma)$ such that:

$$\Sigma_2 = \{(X, 0) \mid X \in \mathcal{X}\},$$

$$R_2 = \{\frac{E(X) \xrightarrow{a} y_0}{X \xrightarrow{a} y_0} \mid X \in \mathcal{X} \wedge a \in A\};$$

- R_3 consists of the single *stuttering axiom* $x \xrightarrow{*} x$.

Elements of $\mathcal{X}$ are referred to as *process names* and E is called the *declaration mapping.* If P is a De Simone calculus, then both the set of process names and the declaration mapping are uniquely determined and will be referred to as $\mathcal{X}_P$ and E_P.

Remark 3.3. Given a De Simone rule, the corresponding variant of it without stuttering is obtained by removing all the stuttering premises and replacing in the target y_i by x_i for each stuttering argument i. If P' is the result of applying this procedure to all De Simone rules in a De Simone calculus P and discarding moreover the stuttering axiom, then it is easy to see that for all $a \neq *$, P proves a transition with label a iff P' does. In addition P proves a $*$-transition from each term to itself. When presenting a concrete De Simone calculus, I mostly give the variants of the rules without stuttering, and often use different names for the variables. PC is a De Simone calculus in this sense. Also the definitions of the various preorders (except for the fair trace preorder at the very end) refer to the $*$-less variants.

In process algebra *deadlock* phenomena are often modeled in terms of states without outgoing transitions. The trace preorder does not preserve deadlock behavior in this sense. Therefore one is often interested in the following refinement of the trace preorder.

Definition 3.4 *(Completed trace preorder).* The set of (finite) *completed traces* of a term t is

$$ctraces(t) \triangleq \{\sigma \in A^* \mid \exists t' : P \vdash t \xrightarrow{\sigma} t' \wedge enabled(t') = \emptyset\}.$$

The *completed trace preorder* $\sqsubseteq_{CT}$ on $T(\Sigma)$ is given by

$$t \sqsubseteq_{CT} t' \triangleq t \sqsubseteq_T t' \wedge ctraces(t) \subseteq ctraces(t').$$

All the counterexamples that were presented to show that the trace preorder is not substitutive for all calculi, can be used again to show that also the completed trace preorder is not substitutive in general. Thus it can be argued that a format for which the completed trace preorder is substitutive should be at least as restrictive as the De Simone format. However, it is well known that in general the completed trace preorder is not substitutive for De Simone calculi. The canonical counterexample can be expressed in PC:

$$a{\cdot}b + a{\cdot}c \sqsubseteq_{CT} a{\cdot}(b + c)$$

but

$$a{\cdot}b \times (a{\cdot}b + a{\cdot}c) \not\sqsubseteq_{CT} a{\cdot}b \times (a{\cdot}(b + c)).$$

Instead of imposing additional constraints on the inference rules, the research on algebraic process theory has concentrated on finding substitutive preorders, as coarse as possible, that refine completed trace equivalence. For several individual De Simone calculi a *full abstractness* result has been established in the literature which says that the so-called *failure preorder* is the coarsest substitutive preorder which refines the completed trace preorder (see for instance [10, 4]). Below it will be shown that the failure preorder is substitutive for ***any*** De Simone calculus. The full abstraction result then follows for any calculus which is sufficiently expressive to distinguish between terms that are not related by the failure preorder.

Definition 3.5 *(Failure preorder).* The set of *failure pairs* of a closed term t is

$$failures(t) \triangleq \{\langle\sigma, X\rangle \in A^* \times 2^A \mid \exists t' : P \vdash t \xrightarrow{\sigma} t' \wedge enabled(t') \cap X = \emptyset\}.$$

The *failure preorder* $\sqsubseteq_F$ on $T(\Sigma)$ is defined by

$$t \sqsubseteq_F t' \triangleq failures(t) \subseteq failures(t').$$

The failure preorder refines the completed trace preorder since there is a 1-to-1 correspondence between failure pairs of the form $\langle\sigma, \emptyset\rangle$ and traces σ, and also a 1-to-1 correspondence between failure pairs of the form $\langle\sigma, A\rangle$ and completed traces σ. Let t and t' be PC terms with $t \not\sqsubseteq_F t'$. Then there exists a pair $\langle a_1 \cdots a_n, X\rangle$ which is a failure pair of t but not of t'. Because PC has guarded recursion only, the induced transition relation is finitely branching ([6, 7, 27]), and for this reason we can assume w.l.o.g. that X is in fact finite ([12]). Now consider the context

$$C[x] = (a_1{\cdot}a_2{\cdot}\ldots{\cdot}a_n{\cdot}(\sum_{b \in X} b)) \times x.$$

It is straightforward to check that $a_1 \cdots a_n$ is a completed trace of $C[t]$ but not of $C[t']$. Hence $C[t] \not\sqsubseteq_{CT} C[t']$. In combination with the following theorem these observations imply that, for the language PC, the failure preorder is the coarsest substitutive preorder that refines the completed trace preorder.

Theorem 3.6. *For any De Simone calculus the associated trace and failure preorders are substitutive.*

Due to the generality of this theorem, its proof is quite involved and refers to a number of definitions and technical lemmas, which are not included in this abstract. The proof exploits the strong similarity between proof trees of transitions and the leftmost term in transitions. In order to emphasize this similarity and to facilitate reasoning about proofs, a special syntax for proofs is introduced in the style of [8].

4 Internal Affairs

All the preorders of the previous section are based on notions of observation which assume that all actions are equally visible. Often however it is realistic to assume that certain state transitions in a reactive or concurrent system cannot be observed from the outside and that the corresponding actions are *internal* or *hidden*. This type of assumption naturally leads to different preorders.

Fix a calculus $P = (\Sigma, A, R)$ and a set $I \subseteq A$ of 'internal' actions. Let $E = A - I$.

Definition 4.1 *(External trace preorders).* Relative to P and I, define the sets of *external traces* and *external completed traces* of a closed term t by

$$etraces(t) \triangleq traces(t)\lceil E,$$

$$ectraces(t) \triangleq ctraces(t)\lceil E.$$

The *external trace preorder* $\sqsubseteq_{eT}$ and *external completed trace preorder* $\sqsubseteq_{eCT}$ are given by

$$t \sqsubseteq_{eT} t' \quad \triangleq \quad etraces(t) \subseteq etraces(t'),$$

$$t \sqsubseteq_{eCT} t' \quad \triangleq \quad t \sqsubseteq_{eT} t' \wedge ectraces(t) \subseteq ectraces(t').$$

The $\sqsubseteq_{eT}$ preorder is essentially the $\ll_{\mathrm{MAY}}$-preorder of [14], except that it is defined relative to a set of internal actions instead of just a single one. In the extreme case of calculi with an empty set of internal actions, external trace equivalence coincides with trace equivalence. Thus the counterexamples which were presented in the previous section for trace equivalence can be used again to argue that it is nontrivial to find a format of rules which is more general than De Simone's format such that the external trace preorder is substitutive. Additional restrictions are needed however since both $\sqsubseteq_{eT}$ and $\sqsubseteq_{eCT}$ are not substitutive in general for De Simone calculi. In the case of $\sqsubseteq_{eT}$ the canonical PC example is

$$a \sqsubseteq_{eT} i{\cdot}a \quad \text{but} \quad a \times a \not\sqsubseteq_{eT} a \times i{\cdot}a$$

(here of course it is assumed that i is internal and a is external). For $\sqsubseteq_{eCT}$ the same example applies that was used in the previous section. Below I will propose some syntactic restrictions on De Simone calculi which, when met, guarantee that $\sqsubseteq_{eT}$ is substitutive. There is no obvious way to relax these restrictions without loosing substitutivity. The external completed trace preorder can be refined into a failure type preorder as follows.

Definition 4.2 *(External failure preorder).* The set of *external failure pairs* of t is

$$efailures(t) \triangleq \{\langle \rho\lceil E,\ X\rangle \in E^* \times 2^E \mid \exists t' : P \vdash t \xrightarrow{\rho} t' \wedge enabled(t') \cap (X \cup I) = \emptyset\}.$$

The *external failure preorder* $\sqsubseteq_{eF}$ is defined by

$$t \sqsubseteq_{eF} t' \triangleq t \sqsubseteq_{eT} t' \wedge efailures(t) \subseteq efailures(t').$$

The preorder $\sqsubseteq_{eF}$ turns out to be substitutive for any De Simone calculus which obeys the restrictions to be defined below and moreover, for sufficiently expressive calculi, it is the coarsest substitutive preorder that refines $\sqsubseteq_{eCT}$. The kernel of $\sqsubseteq_{eF}$ is very similar to the *failure semantics with fair abstraction of unstable divergence* of Bergstra, Klop and Olderog [3], but does not handle stability at the root and successful termination.

Definition 4.3 *(Sleeping arguments).* Let $P = (\Sigma, A \cup \{*\}, R)$ be a De Simone calculus, let $(f, n) \in \Sigma$ and let $1 \leq i \leq n$. Then (f, n) *tests* its i-th argument and the i-th argument is called *awake* if there is a rule in R of type (f, n) in which the i-th position is active (i.e. $a_i \neq *$); otherwise the i-th position is *sleeping.* A subterm s of a term t is *sleeping* if it occurs in a subterm which is on a sleeping position. Of course, subterms which are not sleeping are called *awake.*

Example 4.4. The prefixing operators $a\cdot$ of PC all have a single argument, which is sleeping. All the other operators of PC only have arguments that are awake. A very useful auxiliary operator in algebraic process theory is the binary *left-merge* operator $\lfloor\!\lfloor$:

$$\frac{x \xrightarrow{a} x'}{x \lfloor\!\lfloor y \xrightarrow{a} x' \| y}.$$

The first argument of this operator is awake, whereas its second argument is sleeping.

Definition 4.5 *(Internal action sets).* Let $P = (\Sigma, A \cup \{*\}, R)$ be a De Simone calculus and let I be a nonempty subset of A. The set I is called an *internal action set* if P satisfies the following properties:

1. *Non-blocking*: for each $(f, n) \in \Sigma$, for each argument i of (f, n) that is awake, and for each $a \in I$, R contains a *clearing* rule, i.e. a rule of the form
$$\frac{\{x_i \xrightarrow{a} y_i\} \cup \{x_j \xrightarrow{*} y_j \mid j \neq i\}}{f(x_1, \ldots, x_n) \xrightarrow{b} f(y_1, \ldots, y_n)}$$
where $b \in I$;
2. *Redundancy*: for each rule in R that is not a clearing rule, and with a premis $x_i \xrightarrow{a} y_i$ for some $a \in I$, there is another rule in R which is exactly the same except that it has a premis $x_i \xrightarrow{*} y_i$ instead.

As far as I know, the notion of an operator testing an argument, and also the non-blocking constraint are due to Bloom [5]. The notion of clearing rules occurred earlier in the work of Parrow [23] in the setting of 'static' operators. My constraints are much less restrictive than the ones imposed by Parrow, since he allows only for internal steps in clearing rules, and moreover insists on clearing rules for all arguments, not just the ones that are awake. In contrast to this, Bloom imposes less restrictions than I do, because he has no constraints besides the non-blocking requirement. As a consequence he ends up with the somewhat peculiar notion of *firework simulation*, which in

$$x \oplus y \xrightarrow{\tau} x \qquad x \oplus y \xrightarrow{\tau} y$$

$$\frac{x \xrightarrow{\tau} x'}{x + y \xrightarrow{\tau} x' + y} \qquad \frac{y \xrightarrow{\tau} y'}{x + y \xrightarrow{\tau} x + y'}$$

$$\frac{x \xrightarrow{\tau} x'}{x \| y \xrightarrow{\tau} x' \| y} \qquad \frac{y \xrightarrow{\tau} y'}{x \| y \xrightarrow{\tau} x \| y'}$$

$$\frac{x \xrightarrow{\tau} x'}{x \times y \xrightarrow{\tau} x' \times y} \qquad \frac{y \xrightarrow{\tau} y'}{x \times y \xrightarrow{\tau} x \times y'}$$

$$\frac{x \xrightarrow{\tau} x'}{\varphi(x) \xrightarrow{\tau} \varphi(x')}$$

$$\frac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')} \text{ if } a \in I \qquad \frac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{a} \tau_I(x')} \text{ if } a \notin I$$

Table 3: The inference rules of TAU.

my view does not properly capture the concept of internal actions since a term with two consecutive internal transitions is distinguished from a term with only a single internal transition.

Example 4.6. The calculus PC has no internal action set since there are no clearing rules for the operators $+$ and $\times$. Below I describe an extension of PC called PC_τ which does have an internal action set $\{\tau\}$. Here τ is a 'fresh' label not in the label set A of PC. PC_τ is defined as the union of PC and an auxiliary calculus TAU. The signature of TAU consists of the signature of PC augmented with a binary *internal choice* operator $\oplus$ and unary *hiding* operators τ_I for each $I \subseteq A$. The label set of TAU is $A \cup \{\tau\}$ and its rules are displayed in Table 3. By adding the clearing rules for $+$, the choice combinator of PC, which is essentially the same as the choice combinator of CCS and ACP, is turned into the external choice operator $\square$ of CSP [15]. The $\oplus$ combinator of PC_τ is just the internal choice operator $\sqcap$ of CSP (the notation employed here is from [14]). Note that the fact that $\{\tau\}$ is an internal action set of PC_τ in the sense of Definition 4.5, depends crucially on the fact that the presence of clearing rules is required only for arguments that are awake: $\oplus$ does not need to have clearing rules since both its arguments are sleeping.

Using the same context that was used in the case without internal actions, it is an elementary exercise to show that, assuming that $\sqsubseteq_{eF}$ is substitutive, it is in fact the coarsest substitutive preorder on PC_τ that refines $\sqsubseteq_{eCT}$.

The *must preorder* $\ll_{\mathrm{MUST}}$ of [14] carries over trivially to the setting of this paper; for reasons of space I will not recall its definition here. However, the following theorem holds.

Theorem 4.7. *For any De Simone calculus P and for any internal action set of P, the associated external trace, external failure and must preorders are substitutive.*

5 Input / Output Calculi

The I/O automata approach postulates three different types of labels in a transition system: input actions, output actions and internal actions. Technically this trichotomy is reflected in the notion of an action signature.

Definition 5.1. An *action signature* S is a triple $(in(S), out(S), int(S))$ of three disjoint sets of resp. *input actions*, *output actions* and *internal actions*. The derived sets of *external actions*, *locally controlled actions* and *actions* of S are defined resp. by

$$\begin{aligned} ext(S) &\triangleq in(S) \cup out(S), \\ local(S) &\triangleq out(S) \cup int(S), \\ act(S) &\triangleq in(S) \cup out(S) \cup int(S). \end{aligned}$$

5.1 Input Actions

I/O models require that in each state each input action is enabled, since input actions are considered to be under the control of the environment, and not the machine itself. Below I will present two simple syntactic conditions on De Simone calculi which together guarantee input enabling.

Definition 5.2 *(Input action sets).* Let $P = (\Sigma, A \cup \{*\}, R)$ be a De Simone calculus and let $In \subseteq A$. The set In is called an *input action set* of P if for each signature element $(f, n) \in \Sigma$ and for each action $a \in In$, there is a rule in R of type (f, n) with action a, and a trigger with labels in $In \cup \{*\}$.

Definition 5.3 *(Guardedness).* Let P be a De Simone calculus. A term t over the signature of P is *guarded* if all process names in t occur in sleeping subterms. P is *guarded* if all terms in the image of E_P are guarded.

The above notion of guardedness (which generalizes the notion of guardedness for PC) has been introduced in [27]. Some assumptions about the recursive definitions are needed for input enabling: if X is a process name with recursive definition X, then no outgoing transitions can be derived using the standard inference rules for recursion.

Lemma 5.4 *(Input enabling). Let P be a guarded De Simone calculus and let In be an input action set of P. Then for all closed terms t, $In \subseteq enabled(t)$.*

Note that the notions of input and internal actions are independent in the following sense: in order to prove substitutivity of the 'external' preorders one only needs assumptions about the internal action set, and in the proof of the above Lemma 5.4 one only needs the assumptions about the input action set and guardedness.

5.2 Quiescence

In I/O models deadlock phenomena are not modeled in terms of states without outgoing transitions: due to input enabling there are no such states. Instead one uses the notion of a *quiescent state*, a state which enables no locally controlled actions, but only input actions. In a quiescent state the system waits for stimuli from the environment but has no activity of its own.

Fix an action signature S and a guarded De Simone calculus $P = (\Sigma, act(S) \cup \{*\}, R)$ with $in(S)$ an input action set and $int(S)$ an internal action set.

Definition 5.5 *(Quiescent trace preorder).* A term is *quiescent* if it only enables input actions. The set of (finite) *quiescent traces* of t is

$$qtraces(t) \triangleq \{\rho\lceil ext(S) \mid \exists t' : P \vdash t \xrightarrow{\rho} t' \wedge t' \text{ quiescent}\}.$$

The *quiescent trace preorder* $\sqsubseteq_{qT}$ is defined by

$$t \sqsubseteq_{qT} t' \triangleq t \sqsubseteq_{eT} t' \wedge qtraces(t) \subseteq qtraces(t').$$

The quiescent trace preorder is in general not substitutive. In order to see this, consider the simple case in which the set of input actions is empty. In this case the quiescent trace preorder coincides with the external completed trace preorder and the standard example — $a \cdot b + a \cdot c$ versus $a \cdot (b + c)$ — showing that the latter is not substitutive can also be used to show nonsubstitutivity of the former. It turns out that basically all one has to do in order to guarantee that $\sqsubseteq_{qT}$ is substitutive, is to add the following constraint which says that a context has no longer the possibility to block output actions.

δ	0	inaction
e	1	prefixing; for each $e \in A \cup \overline{A}$
$I+J$	2	external choice; for each $I, J \subseteq A$
$\oplus$	2	internal choice
$\|_H$	2	parallel, merge; for each $H \subseteq A$
φ	1	renaming; for each $\varphi : A \to A$
τ_I	1	hiding, abstraction; for each $I \subseteq A$
X	0	process names; for each $X \in \mathcal{X}$

Table 4: The signature of IOC.

Definition 5.6 *(Output blocking).* P has no *output blocking* if for each $(f,n) \in \Sigma$, for each argument i of (f,n) that is awake, and for each action $a \in out(S)$, there is a rule in R with type (f,n), called a *clearing rule*, in which the premis for x_i has label a, all other premises have a label in $in(S) \cup \{*\}$, and the conclusion has a label in $local(S)$.

Note that the conditions on clearing rules for output actions are less restrictive than those for internal actions. The clearing rules for internal actions do not allow for other arguments to take non-stuttering steps, they require the target to be of the form $f(y_1, \ldots, y_n)$, and the label of the conclusion has to be a internal action.

All the restrictions on general transition system specifications introduced thus far come together in the following definition.

Definition 5.7. An *I/O calculus* is a triple $Q = (\Sigma, S, R)$ with S an action signature and $(\Sigma, act(S) \cup \{*\}, R)$ a guarded De Simone calculus with $in(S)$ an input action set, $int(S)$ an internal action set, and no output blocking.

Theorem 5.8. *For any I/O calculus the associated quiescent trace preorder is substitutive.*

Example 5.9. The I/O calculus IOC is parametrized by a set A, ranged over by $a, b, \ldots$, and a set $\mathcal{X}$ of process names, ranged over by $X, Y, \ldots$. Like in CCS, there is also a set $\overline{A}$, with $\overline{a}, \overline{b}, \ldots$ ranging over $\overline{A}$. The action signature of IOC is $(A, \overline{A}, \tau)$. So actions of the form a are input actions, actions of the form $\overline{a}$ are output actions and τ is the only internal action. The signature of IOC is displayed in Table 4, and its rules are given in Table 5. However, the presence of clearing rules for τ is assumed implicitly. Also, if for a certain type (f,n) and a certain input action a, Table 5 mentions no rule with that type and that action, then IOC contains in fact a rule

$$f(x_1, \ldots, x_n) \xrightarrow{a} f(x_1, \ldots, x_n).$$

Due to this convention, the rules for prefixing in Ta-

$e\cdot x \xrightarrow{e} x$	$x \oplus y \xrightarrow{\tau} x$	$x \oplus y \xrightarrow{\tau} y$	
$\dfrac{x \xrightarrow{e} x'}{x\ {}_I{+}_J\ y \xrightarrow{e} x'}$	if $e \in I \cup \overline{A}$	$\dfrac{y \xrightarrow{e} y'}{x\ {}_I{+}_J\ y \xrightarrow{e} y'}$	if $e \in J \cup \overline{A}$
$\dfrac{x \xrightarrow{\overline{a}} x',\ y \xrightarrow{a} y'}{x \parallel_H y \xrightarrow{\overline{a}} x' \parallel_H y'}$	if $a \in H$	$\dfrac{x \xrightarrow{a} x',\ y \xrightarrow{\overline{a}} y'}{x \parallel_H y \xrightarrow{\overline{a}} x' \parallel_H y'}$	if $a \in H$
$\dfrac{x \xrightarrow{\overline{a}} x'}{x \parallel_H y \xrightarrow{\overline{a}} x' \parallel_H y}$	if $a \notin H$	$\dfrac{y \xrightarrow{\overline{a}} y'}{x \parallel_H y \xrightarrow{\overline{a}} x \parallel_H y'}$	if $a \notin H$
$\dfrac{x \xrightarrow{a} x',\ y \xrightarrow{a} y'}{x \parallel_H y \xrightarrow{a} x' \parallel_H y'}$	if $a \notin H$		
$\dfrac{x \xrightarrow{a} x'}{\varphi(x) \xrightarrow{\varphi(a)} \varphi(x')}$		$\dfrac{x \xrightarrow{\overline{a}} x'}{\varphi(x) \xrightarrow{\overline{\varphi(a)}} \varphi(x')}$	
$\dfrac{x \xrightarrow{e} x'}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')}$	if $e \in \overline{I}$	$\dfrac{x \xrightarrow{e} x'}{\tau_I(x) \xrightarrow{e} \tau_I(x')}$	if $e \notin \overline{I}$

Table 5: The inference rules of IOC.

ble 5 look the same as the rules for this operator in the calculus PC. However, the prefixing operators of IOC and PC are really different. Both in IOC and PC the process $\overline{a}{\cdot}\delta$ has two states. In IOC the initial state has a self loop for each input action. Moreover there is a transition $\overline{a}$ to the final state in which also has a self loop for each input action. In PC the process just has the single $\overline{a}$ transition from the initial to the final state and there are no loops. In calculi like CCS and PC, the prefixing and choice operators play a crucial role since they make it possible, provided there is a sufficient amount of recursive power, to specify in the language each finitely branching transition system up to isomorphism. In order to obtain a similar expressiveness, the language IOC incorporates, besides the modified prefixing operators, binary choice operators ${}_I{+}_J$ and hiding operators τ_I. Using recursion, prefixing and ${}_I{+}_J$, it is a simple exercise to specify any finitely branching, input enabled automaton over a given, finite action signature that contains no transitions that are labeled with internal actions. Arbitrary finitely branching, input enabled automata can be obtained by applying a hiding operator on top of this construction. The choice operator $+$ of PC$_\tau$ does fit the I/O calculi format, but the reader can easily convince him/herself that this operator alone does not give the required amount of expressiveness. The parallel combinator $\parallel_H$ is just a proposal which looks interesting. Subsequent research has is needed to see whether this operator is sufficiently expressive. It might be that a simpler calculus can be obtained by parametrizing expressions by an action signature, like in CSP [15] and all I/O models that have been proposed thus far. In this case one would only need a single parallel combinator instead of an operator for each $H \subseteq A$.

In Table 5, the rules for recursion and stuttering are omitted as usual, $a, b, \ldots$ range over A and $e, \ldots$ range over external actions, unless further restrictions are made. The symbols H, I, J range over subsets of A, and φ ranges over $A \rightarrow A$.

5.3 Fairness

All I/O models incorporate some notion of fairness, but they all do it in a different way. All I/O models agree however, that an execution of the composition of automata is fair iff it is fair to each of its components. Furthermore, a key result in all I/O models is that the preorder induced by the traces of fair executions is substitutive for the operations of composition, hiding and renaming, which are the only operations defined on I/O automata thus far. Therefore it seems reasonable to require that the fair trace preorder (or at least a close approximation of it) is also substitutive for any new operator.

In the syntactic setting of this paper, weak process fairness appears to be a natural fairness notion. As it turns out, the resulting preorder is substitutive for any I/O calculus. Unfortunately, due to lack of space, it is not possible to present the precise definition of this preorder in this abstract. For this the reader is refered to the full version of this paper.

Fix an I/O calculus P.

Definition 5.10 *(Fairness).* An execution of P (i.e. an infinite sequence of proofs of consecutive transitions) is *unfair* if it has a suffix and there exists a marking of the first proof such that if this subterm is traced throughout the execution, it remains the same, it stays awake, stutters and enables a locally controlled action. An execution is *fair* if it is not unfair.

In the full paper it will be argued in detail why this notion of fairness is reasonable and useful. The following lemma, whose formulation is due to [18], implies that the fairness notion is *feasible* in the sense of [1].

Lemma 5.11. *Let $\mathcal{T}$ be a finite execution fragment and let σ be a sequence of input actions. Then there exists a $\mathcal{E}$ such that $\mathcal{T}\mathcal{E}$ is a fair execution and $\mathcal{E}\lceil in(S) = \sigma$.*

For nontrivial calculi like IOC it is trivial to show that the above fairness notion is *liveness enhancing* in the sense of [1]: there is some term which has some liveness property which it would not have without the fairness assumption. Since the notion of *independent actions* has not been defined in the setting of this paper, it is not possible to argue that the fairness notion is *equivalence robust* in the sense of [1]. An interesting topic for future work is to define independence of actions (transitions) for arbitrary De Simone calculi in the style of [8]. I expect no problems with equivalence robustness once this work has been carried out, because *conspiracies*, the main source of difficulties with the equivalence robustness of weak process fairness in languages like CCS and CSP, are no longer possible in I/O calculi.

Definition 5.12 *(Fair trace preorder).* The set *ftraces*(t) of *fair traces* of a term t is defined as the projection on the external actions of the traces corresponding to the fair executions of t. The induced *fair trace preorder* $\sqsubseteq_{fT}$ is defined by

$$t \sqsubseteq_{fT} t' \triangleq \mathit{ftraces}(t) \subseteq \mathit{ftraces}(t').$$

The expected theorem is true:

Theorem 5.13. *For any I/O calculus the associated fair trace preorder is substitutive.*

6 Concluding Remarks

"So where are the I/O automata?" the reader may ask. And indeed, not even their definition does occur in the text. The full paper will define in detail how an I/O calculus determines operations on I/O automata. The definition is completely standard for readers who are familiar with SOS and who are used to think about Plotkin-style calculi as a way to define operations on automata, even when these automata are not present explicitly.

A topic for future research is to extend the I/O calculi format to allow for predicates on terms dealing with action signatures. This would make it possible to define the parallel composition operator of the I/O automata models of [16, 19], which appears to be simpler and more intuitive than the parallel combinators of my calculus IOC. Finally, an obvious open question is to give a complete axiomatization of the quiescent trace preorder for the recursion free part of (a variant of) IOC.

References

[1] K.R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.

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

[3] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Failures without chaos: a new process semantics for fair abstraction. In M. Wirsing, editor, *Formal Description of Programming Concepts – III, Proceedings of the* 3th *IFIP WG 2.2 working conference,* Ebberup 1986, pages 77–103, Amsterdam, 1987. North-Holland.

[4] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Readies and failures in the algebra of communicating processes. *SIAM Journal on Computing*, 17(6):1134–1177, 1988.

[5] B. Bloom. Strong process equivalence in the presence of hidden moves. Preliminary report, October 1990.

[6] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced: preliminary report. In *Proceedings* 15th *ACM Symposium on Principles*

of Programming Languages, San Diego, California, pages 229–239, 1988.

[7] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced. Technical Report 90-1150, Department of Computer Science, Cornell University, Ithaca, New York, August 1990.

[8] G. Boudol and I. Castellani. Permutation of transitions: an event structure semantics for CCS and SCCS. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Noordwijkerhout, volume 354 of *Lecture Notes in Computer Science*, pages 411–427. Springer-Verlag, 1989.

[9] G. Costa and C. Stirling. A fair calculus of communicating systems. *Acta Informatica*, 21(5):417–441, December 1984.

[10] R. De Nicola. *Testing Equivalences and Fully Abstract Models for Communicating Processes.* PhD thesis, Department of Computer Science, University of Edinburgh, 1985.

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

[12] R.J. van Glabbeek. The linear time - branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90*, Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.

[13] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings 16th ICALP*, Stresa, volume 372 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 1989. Full version to appear in *Information and Computation.*

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

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

[16] B. Jonsson. A model and proof system for asynchronous networks. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, Minaki, Ontario, Canada, pages 49–58, 1985.

[17] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Department of Computer Systems, Uppsala University, 1987. DoCS 87/09.

[18] N.A. Lynch and E.W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82(1):81–92, July 1989.

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

[20] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

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

[22] J. Parrow. *Fairness Properties in Process Algebra - With Applications in Communication Protocol Verification.* PhD thesis, Department of Computer Systems, Uppsala University, 1985. DoCS 85/03.

[23] J. Parrow. The expressive power of simple parallelism. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Proceedings PARLE'89*, Eindhoven, *Vol. II (Parallel Languages)*, volume 366 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, 1989.

[24] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts - II*, Garmisch, pages 199–225, Amsterdam, 1983. North-Holland.

[25] R. de Simone. *Calculabilité et Expressivité dans l'Algebra de Processus Parallèles* MEIJE. Thèse de 3e cycle, Univ. Paris 7, 1984.

[26] 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.

[27] F.W. Vaandrager. Expressiveness results for process algebras. In preparation, 1991.