A Process Algebraic View of Input/Output Automata

Rocco De Nicola*
Università di Roma "La Sapienza"
Dipartimento di Scienze dell'Informazione
denicola@vm.cnuce.cnr.it

Roberto Segala†
Massachusetts Institute of Technology
Laboratory for Computer Science
segala@theory.lcs.mit.edu

February 28, 1994

Abstract

Input/Output Automata are a widely used formalism for the specification and verification of concurrent algorithms. Unfortunately, they lack an algebraic characterization, a formalization which has been fundamental for the success of theories like CSP, CCS and ACP. We present a many-sorted algebra for I/O Automata that takes into account notions such as interface, input enabling, and local control. It is sufficiently expressive for representing all finitely branching transition systems, hence all I/O automata with a finitely branching transition relation. Our presentation includes a complete axiomatization of the external trace preorder relation over recursion free processes with input and output.

*The first author has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", contract n. 91.00894.69 and by Istituto di Elaborazione dell'Informazione of CNR at Pisa

†The second author has been 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.

1 Introduction

Input/Output Automata [14, 20, 11, 12] are a widely used and deeply investigated formalism for specifying and verifying concurrent systems. Unfortunately, they have never been equipped with an algebraic characterization, a formalization that has been fundamental for the success of theories like CSP, CCS and ACP [10, 15, 8, 1]. The goal of this paper is to improve our understanding of the intricacies of I/O automata by describing them as a process algebraic theory. This will permit algebraic manipulation and will provide an alternative to the commonly used verification method based on possibilities mappings.

We start by designing an algebra that incorporates the fundamental features of I/O automata of Lynch and Tuttle [14] and captures the essential role of concurrent composition, hiding and renaming. Our design aims at maintaining minimality of operators and universal expressivity. We base our characterization on the following basic features of I/O automata:

1. *explicit interfacing*: a transition-invariant interface is associated with each process;
2. *input/output distinction*: a clear distinction is made between, locally controlled, output actions and, externally controlled, input actions;
3. *input enabling*: input actions are enabled in every state;
4. *local control*: each action is under the control of at most one process.

For the sake of simplicity, in spite of the fact that it is an important issue always considered within the I/O automata formalism, we decided to ignore fairness in this paper.

The operators in our calculus associate distinct sets of input and output actions (interfaces) with each process. This permits capturing a critical aspect of I/O automata, namely the distinction between input and output actions. To associate an interface to a process we use many-sorted algebras: each process has a sort which stands for its interface. This choice permits dealing with partial operators in a clean way. As an example consider the parallel composition operator. To comply with the requirement that each action be under the control of at most one process, parallel composition is permitted only between pairs of processes that do not have common output actions. Many-sorted algebras allow us to capture this restriction by defining the parallel operator as a family of sorted operators, one for each pair of compatible interfaces.

A key issue in defining our I/0 calculus is the way input enabling is enforced. We present our choice with the support of an example. Consider process $P = a.e$, which is able to perform an action a and then behave like e. If the system is input enabled, the above process must be able to perform all input actions, also those different from a. Indeed, it is for the unspecified input actions that we have to make a definite choice about how to represent the future behavior. We considered two different possibilities:

1. *Angelic*: Unexpected inputs are ignored. A system copes with unexpected input actions by first receiving them and then returning to the state it was in before the input had taken place (*self-loop*). For example, system $P = a.e$, after accepting any input b different from a, is again ready to accept the a-action.

2. *Demonic*: Unexpected inputs are considered as catastrophic. A system, after any unexpected input, moves to a special state Ω from which any behavior is possible. Thus, $P = a.e$, after any b-action different from a, moves to Ω.

The Angelic choice was made by Vaandrager in [21]; here, we study the impact of the Demonic approach. In our view, for $P = a.e$ above, the prefixing operator specifies its behavior only for action a and says nothing about input actions different from it. By interpreting this in the framework of I/O automata and of the correctness of an implementation with respect to a given specification, we have that an implementation of P should be correct independently of the behavior it exhibits when provided with any input action different from a.

Due to this basic choice, our calculus will be called the *Demonic calculus of I/O Automata* (DIOA).

This demonic approach has been partially influenced by the *Receptive Process Theory* (RPT) of Mark Josephs [13]. However, the semantics of RPT is denotational, and like CSP [10], it is described by means of sets of failures, traces and divergencies. The handling of underspecification within RPT is even more demonic than ours; underspecification is propagated backward, thus if a process P can perform an output action o and move to the equivalent of an Ω state, then the whole P is equivalent to Ω.

The semantics of our many-sorted language is obtained by associating a labeled transition system with each term by means of a set of inference rules in the usual SOS style [16]. This choice paves the way toward a number of possible semantics for I/O automata. Indeed, Labeled Transition Systems have been equipped with a number of behavioral equivalences each aiming at capturing particular aspects (sequences of interactions with the external world, reactions to external experiments, branching structures of the sequences of interactions, etc.) that permit relating different descriptions of a given system. For an overview and a discussion on the relationships between different behavioral relations, the interested reader is referred to [3, 7].

We will study a very simple preorder (and the induced equivalence) over the transition systems associated to the terms representing I/O automata and postpone investigation of more interesting equivalences to further study. The behavioral relation we will consider is the *external trace preorder*; it permits identifying all those automata that can perform the same sequences of external (input or output) actions by ignoring possible differences that could be induced by internal actions. This relation allows us to throw some light on the impact of interfaces, input enabling and local control of actions in an algebra of I/O automata.

For the external trace preorder, we propose a set of sound algebraic laws that are also *complete* with respect to recursion-free DIOA processes. The completeness proof is achieved via techniques of reduction to normal forms, these are significatively different from the usual

ones employed in standard process algebras. Indeed, they are special normal forms that contain also restricted occurrences of the operator for parallel composition. We feel that they should be useful also for behavioral relations that are more accurate than external trace preorder.

Particularly important for our result is an operator representing internal choice. It does not fit Vaandrager's general format [21] and forces us to prove substitutivity of our preorder explicitly.

In [18] another behavioral relation is studied: the *quiescent preorder* of [21]. The main idea behind this relation is that a quiescent trace is a trace which leads to states from which only input actions are enabled. The quiescent preorder is given by external and quiescent trace inclusion; it is a restriction to finite traces of the fair preorder of [14], and it is another stepping stone toward the study of fairness sensitive semantics. It is worth remarking that the complete axiomatization for the quiescent preorder of [18], is similar to that for our external preorder; the main difference arises for the complex side conditions which are needed for dealing with quiescence of the empty trace.

The rest of the paper is organized as follows: Sections 2 and 3 contain an overview of I/O automata and process algebras. Section 4 contains the operational definition of a demonic calculus of I/O automata. It also contain hints on how to define an angelic calculus. Section 5 contains a set of laws which permit sound manipulation of DIOA terms with respect to the external trace preorder. Section 6 contains the proof that the laws of the previous section provide a complete axiomatization of recursion free DIOA expressions without renaming, hiding and parallel composition. Section 7 extends the completeness result to the remaining operators. Section 8 contains some concluding remarks.

2 An Overview of I/O Automata

In this section we briefly introduce I/O automata and their basic formal definition. For a complete account, we refer the reader to [14]. We will stick as much as possible with the original notation, but some notions, such as execution and schedule modules, will be ignored. Later we will homogenize the notation with that of process algebras; here, on few occasions, we will recall some obvious correspondences.

One of the basic concepts is the notion of action signature. It represents the interface of an I/O automaton with the external environment.

Definition 2.1 (Action signature) Given three disjoint sets in, out and int we refer to the triple (in, out, int) as an *action signature* S. The sets in, out and int are respectively denoted by $in(S)$, $out(S)$ and $int(S)$. The entire set of actions $in \cup out \cup int$ is denoted by $acts(S)$. The set of *external actions* $in \cup out$ is denoted by $ext(S)$. Finally the set of *locally controlled actions* $int \cup out$ is denoted by $local(S)$. □

An I/O automaton is formally defined as follows.

Definition 2.2 (Input-Output automaton) An *Input-Output automaton* A is a tuple (Q, Q_0, S, t, P) where

- Q is a set of states and is referred to as $states(A)$,
- $Q_0 \subseteq Q$ is the set of start states and is referred to as $start(A)$,
- S is an action signature and is referred to as $sig(A)$,
- $t \subseteq Q \times acts(S) \times Q$ with the property that $\forall q \in Q, a \in in(S)\, \exists q' \in Q : (q, a, q') \in t$. It is referred to as $steps(A)$, and
- P is a partition of $local(S)$ and is referred to as $part(A)$.

A step $(q, a, q') \in steps(A)$ is conventionally denoted by $q \xrightarrow{a} q'$. □

The difference between classical automata and I/O automata arises for the presence of

1. the action signature that permits introducing different types of actions;
2. the constraint that input actions be always possible and that their transition relation be always defined;
3. the presence of the partition P of the locally controlled actions.

The role of P will be more clear when the notion of fair execution is introduced. The basic idea behind the partitioning of the set of local actions is that a system can be seen as a set of subcomponents and that all actions of a partion are under the control of a single subcomponent. Fairness will then be defined to guarantee that infinitely often all subcomponents take the chance of executing at least one of the actions in the different partitions.

We now introduce *executions* and *schedules*, i.e., sequences of labeled transitions and sequences of labels that in process algebras are known as *runs* and *traces*.

Definition 2.3 (Executions and schedules) Given an I/O automaton A, an *execution fragment* is a finite sequence $q_0 a_0 q_1 \cdots a_k q_k$ or infinite sequence $q_0 a_0 q_1 a_1 q_2 \cdots$ of alternating states and actions such that $(q_i, a_i, q_{i+1}) \in steps(A)$ for every i. An *execution* is an execution fragment beginning with a start state (i.e., $q_0 \in start(A)$). The *schedule* of an execution x is the subsequence of actions appearing in x. It is denoted by $sched(x)$. Executions and schedules of an I/O automaton A are denoted respectively by $execs(A)$ and $scheds(A)$. □

Very often, the main concern is with the visible behavior of a system and thus with sequences of external actions. These are obtained from schedules by dropping the elements of int.

Definition 2.4 (External action signature and schedule) An *external action signature* is an action signature consisting only of external actions. The external action signature of a signature S is $(in(S), out(S), \emptyset)$, i.e., S without internal actions.

Given a sequence y of actions and a set of actions X we denote by $y\lceil X$ the maximal subsequence of y consisting only of actions of X. The set of external schedules of an I/O automaton A, denoted by $escheds(A)$, is given by $\{y\lceil ext(A) : y \in scheds(A)\}$. □

A first notion of observational equivalence for I/O automata is then the following, which identifies all those I/O automata which can perform the same sequences of external actions. It is called unfair equivalence because in [14] it is presented in preparation of a different equivalence relation, namely the fair equivalence.

Definition 2.5 (Unfair equivalence) Two I/O automata A and B are said to be *unfairly equivalent*, $A \equiv_U B$, iff A and B have the same external action signature and $escheds(A) = escheds(B)$. □

It can be easily shown that the above relation is indeed an equivalence relation. We proceed now to introducing three basic operations over I/O automata: hiding, renaming and parallel composition. In passing, we note that the above relation is also a congruence for these operators.

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

- $out(Hide_I(A)) = out(A) \backslash I$, 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 to the external environment. The only difference between the original and the resulting I/O automaton is in the signature. Executions and schedules are exactly the same. It is worth noting that the definition of hiding in [14] does not contain the restriction that $I \cap in(A) = \emptyset$; it is however immediate to observe that the operator would not be closed for I/O automata if we permit hiding of input actions: $part(A)$ would not be a partition of $local(A)$ any more.

Definition 2.7 (Renaming) 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. We introduce now parallel composition of I/O automata, it is of fundamental importance because it exactly characterizes the way I/O automata communicate and/or synchronize.

An important property of I/O automata for defining parallel composition is their compatibility. The compatibility condition requires that internal actions be not used to communicate and that every action be under the control of at most one process.

Definition 2.8 (Compatibility of I/O automata)

1. A set of action signatures $\{S_i : i \in I\}$ are *compatible* iff for all $i, j \in I$
   (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 *compatible* iff their action signatures are compatible.

□

Definition 2.9 (Composition of I/O automata) The composition $A = \prod_{i \in I} A_i$ of compatible I/O automata $\{A_i : i \in I\}$ is defined to be the I/O automaton 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 composition $S = \prod_{i \in I} S_i$ of 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. $steps(A) = \{\ ((q_i)_{i\in I}, a, (q'_i)_{i\in I}) : \forall i \in I$
$a \in acts(A_i)$ implies $(q_i, a, q'_i) \in steps(A_i)$, $a \notin acts(A_i)$ implies $q_i = q'_i\}$

□

The step function above states that all processes must synchronize on common actions. However, due to the input enabling and to the local control conditions, the synchronization condition does not mean that communication is synchronous; only one process "can decide" when a communication should occur.

A side effect of input enabling is that it may prevent a system from performing locally controlled actions by means of an infinite sequence of input actions. This is avoided by restricting observations to fair executions. In the following definition we make use of the partitions of the locally controlled actions for the first time.

Definition 2.10 (Fair executions) A *fair execution* of an I/O automaton A is an execution x such that for all $X \in part(A)$

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

A finite fair execution is also said to be *quiescent*.

A *fair schedule* is the schedule of a fair execution. We denote the set of fair schedules of an I/O automaton A with $fscheds(A)$. □

The notion of fairness defined above reminds weak fairness of [6], but the two concepts are somewhat different. First, in [6] fairness is considered relatively to all actions, while in I/O automata only locally controlled actions are taken into account. Second, and more important, within I/O automata theory, fairness is defined by reasoning about sets of actions rather than about single actions. The idea behind the partition of locally controlled actions is that every element of the partition represents the set of actions under the control of a particular component of the global system. In this way, a notion of fair turn can be expressed, ensuring that each component that is continuously willing to perform some of its local actions will eventually perform one of them.

We can now define a new equivalence relation. Like the unfair equivalence introduced above, the new equivalence is substitutive for the I/O automata operators whenever these operators are defined; i.e. it is a weak congruence for I/O automata.

Definition 2.11 (Fair equivalence) Two I/O automata A, B are *fair equivalent* ($A \equiv_F B$) iff A and B have the same external action signature and $fscheds(A) = fscheds(B)$. □

Based on the notion of fair trace, it is possible to introduce a notion of *implementation*. An I/O automaton A_1 implements an I/O automaton A_2 if they have the same external action signature and $fscheds(A_1) \subseteq fscheds(A_2)$. Trivial implementations are avoided by input enabling and fairness. These two concepts, in fact, state that a process must accept all stimuli from the external environment and must perform its output actions whenever it has the possibility to do so, i.e., it must give an answer when requested.

Beside the fair preorder, that is the basic relation for comparing and contrasting I/O automata [14], other relations have been considered. These relations ignore fairness issues, but are closer to the classical relations defined for process algebras:

- *unfair preorder* is based on (finite and infinite) schedules inclusion, while
- *quiescent preorder* is based on both finite schedules inclusion and quiescent schedules inclusion.

The quiescent preorder is a close attempt to capture the fair preorder using only finite schedules. It was used by Vaandrager in [21] and is reminiscent of the maximal trace congruence used in [2]. In this paper, mainly for the sake of simplicity, we concentrate on the unfair preorder; it corresponds essentially to the language equivalence of classical automata and has been used also for comparing concurrent system [9]. We see work on this simple preorder as a stepping stone toward modeling the more elaborate ones. Indeed, as we mentioned in the Introduction, similar results to those obtained for the unfair preorder have been obtained also for the quiescent one [18].

3 Process Algebras Overview

The basic idea of Process Algebras [15, 10, 1] is the use of a few elementary processes and of some basic operators corresponding to primitive operations such as sequentialization, parallelism, nondeterminism and synchronization to describe concurrent processes. A process is represented as an expression which is built inductively from the elementary processes and the operators.

Process Algebras are generally recognized as a good formalism for describing and studying properties of distributed concurrent systems. Very often, a process algebra is defined by specifying its syntax and the transitional semantics of its terms by means of Structured Operational Semantics (SOS) [16] which associate a labeled transition system (essentially a transition labeled classical automaton) to each term. Although SOS has become a standard tool for specifying basic semantics of process algebras, it was early recognized that it does not yield extensional description of processes. Thus, techniques have been developed to abstract from unwanted details in systems descriptions. Many of these techniques are based on behavioral equivalences or preorders; two terms are identified whenever no observer can notice any difference between their external behaviors.

For many of these equivalences complete axiomatizations exist; these are important because the algebraic laws allow one to manipulate process expressions by applying simple axioms and inference rules. Indeed, they constitute the theoretical basis for a class of verification tools which permit establishing the relationship between expressions by means of pure algebraic analysis.

In this paper we concentrate on the SOS approach, but rather than Labeled Transition Systems, we use I/O automata as underlying model and we analyze particular preorder relations which are connected to the unfair preorder of I/O automata.

Below, the basic notions needed for defining a process algebra based on the LTS approach are introduced. We start with the notion of signature, which is now needed due to the different type of the actions we want to consider. A signature represents the basic processes (constants) and the primitive operators: $(f, s_1 s_2 \cdots s_n, s)$ is an operator taking n processes respectively of sort $s_1 \cdots s_n$ as arguments and giving back a process of sort s. Well known calculi like CCS are one-sorted; the more general notion of many-sorted signature will be useful to naturally model association of interfaces to processes.

Definition 3.1 (Signatures and terms) Let $\mathcal{S}$ be a set of *sorts* ranged over by $s, s_1, s_2, \ldots$

1. A *signature element* is a triple $(f, s_1 s_2 \cdots s_n, s)$ consisting of a function symbol f, a sequence of sorts $s_1 \cdots s_n : s_i \in \mathcal{S}, i = 1, \ldots, n$, and a single sort $s \in \mathcal{S}$. s is called the sort of the signature element and n is its arity. In a signature element (c, λ, s), c is often referred to as a constant symbol of sort s.

2. A *signature* is a pair $\Sigma = (\mathcal{S}, \mathcal{O})$ consisting of a set of sorts $\mathcal{S}$ and a set of signature elements $\mathcal{O}$. We denote sort and function symbols of a signature Σ by $sorts(\Sigma)$ and $op(\Sigma)$.

3. The set of *terms* over Σ, is denoted by $T(\Sigma)$. The set of terms of a particular sort $s \in \mathcal{S}$ are denoted by $T(\Sigma)_s$.

□

The following definition introduces the notions of substitutive relation.

Definition 3.2 (Substitutivity) Let Σ be a signature and $\mathcal{R}$ a relation over $T(\Sigma) \times T(\Sigma)$. $\mathcal{R}$ is *substitutive* iff for each signature element $(f, s_1 s_2 \cdots s_n, s)$ of Σ and each t_i, t_i' of sort s_i,

$$t_1 \mathcal{R} t_1', \ldots, t_n \mathcal{R} t_n' \Longrightarrow f(t_1, \ldots, t_n) \mathcal{R} f(t_1', \ldots t_n').$$

□

We proceed by formally defining how a set of transitions can be associated to terms of a given signature. To this purpose *induction rules* will be used. The interpretation of a rule is that, whenever the transitions of the premises are possible, the transition of the conclusion is possible.

Definition 3.3 (Transition rules and calculi) Let A be a given set of labels and let Σ be a signature.

- A transition rule has the form

$$\frac{t_1 \xrightarrow{a_1} t_1', \ldots, t_n \xrightarrow{a_n} t_n'}{t \xrightarrow{a} t'}$$

  where $t_i, t_i' \in T(\Sigma)$, $t, t' \in T(\Sigma)$, $a_i \in A$ and $a \in A$. The elements $t_i \xrightarrow{a_i} t_i'$ are called the *premises* and $t \xrightarrow{a} t'$ is called the *conclusion*.

- A *calculus*, is a triple $P = (\Sigma, A, R)$ where Σ is a signature, A is a set of labels and R is a set of transition rules.

□

Transitions, which are labeled by a single action, can be extended to sequences of actions in the obvious way:

$$t \xrightarrow{a_1 \cdots a_n} t' \text{ iff } \exists t_1, \ldots, t_{n-1} : t \xrightarrow{a_1} t_1 \longrightarrow \cdots \longrightarrow t_{n-1} \xrightarrow{a_n} t'.$$

Unfair and quiescent preorders of I/O automata can be naturally expressed within the process algebraic framework. The notion of schedule is now replaced with the equivalent notion of trace. The unfair preorder is, for the moment, expressed only through finite traces. Throughout the rest of the paper we identify sorts with action signatures.

Definition 3.4 (Quiescence and traces) Given a many-sorted calculus with input and output actions, the set of *enabled actions* from an expression e is defined as

$$\{a | \exists e' : e \xrightarrow{a} e'\}.$$

An expression e is *quiescent* if it only enables input actions.

The set of (finite) external traces of an expression e of sort S is defined as

$$etraces(e) = \{h \lceil ext(S) | \exists e' : e \xrightarrow{h} e'\}$$

where h denotes a sequence of actions and $h \lceil A$ is the projection of h on A.

The set of quiescent traces of an expression e of sort S is defined as

$$qtraces(e) = \{h \lceil ext(S) | \exists e' : e \xrightarrow{h} e', quiescent(e')\}.$$

□

Definition 3.5 (Preorder relations) Let $P = (\Sigma, A, R)$ be a many-sorted calculus with input and output actions, and e_1, e_2 be two terms of $T(\Sigma)$.

- The *external trace preorder* $\sqsubseteq_{ET}$ is defined as follows:

  $e_1 \sqsubseteq_{ET} e_2$ iff e_1 and e_2 have the same external action signature and $etraces(e_1) \subseteq etraces(e_2)$.

- The *quiescent preorder* $\sqsubseteq_Q$ is defined as follows:

  $e_1 \sqsubseteq_Q e_2$ iff $e_1 \sqsubseteq_{ET} e_2$ and $qtraces(e_1) \subseteq qtraces(e_2)$.

□

Definition 3.6 (Equivalences) The kernels of $\sqsubseteq_{ET}$ and $\sqsubseteq_Q$ are respectively called *external trace equivalence* and *quiescent equivalence*.

- $e_1 \equiv_{ET} e_2$ iff $e_1 \sqsubseteq_{ET} e_2$ and $e_2 \sqsubseteq_{ET} e_1$,
- $e_1 \equiv_Q e_2$ iff $e_1 \sqsubseteq_Q e_2$ and $e_2 \sqsubseteq_Q e_1$.

□

4 A Calculus of Demonic I/O Automata

This section introduces a calculus for I/O automata following the demonic approach; i.e., the approach that considers the presence of an unexpected input as catastrophic. The calculus is many sorted and each sort represents an action signature consisting of input and output actions and a single internal action τ. Generally, due to additional flexibility in expressing fairness with respect to different internal tasks, I/O automata signatures have more than one internal action. In this paper, however, we do not address fairness issues and thus we restrict attention to a calculus with a single internal action. At the end of the section we will discuss possible extensions to handle multiple internal actions.

We proceed now by defining the operational semantics of a pair of basic operators (constants) and a set of primitive operators which permit building new I/O automata from existing ones. This set of operators include the operators, renaming, hiding and parallel composition, of [14] described in the previous section and other operators which are directly inspired by those used in process algebras. For the basic operators we will directly associate an I/O automaton; for the others, we will show by structural induction how a new I/O automaton is obtained from the ones corresponding to the operands. Each formal description below will be preceded or followed by some comments on the intuition behind the operators. In the rest of this paper, when describing expressions and I/O automata, we will omit sort indexes whenever they are not relevant or they are evident from the context.

Quiescent I/O automaton: The quiescent I/O automaton, nil_S, is an I/O automaton not enabling any locally controlled action and not expecting any input. Due to the input enabling condition, the quiescent I/O automaton moves to the unspecified I/O automaton for each input action of its action signature. The move to the unspecified I/O automaton (described below) is the interpretation to the fact that the effect of unexpected inputs is catastrophic. Formally, for each sort S we define an operator nil_S taking no arguments. There is only one transition rule for nil_S:

$$\textbf{nil} \quad nil_S \xrightarrow{a} \Omega_S \quad \forall a \in in(S)$$

Unspecified I/O automaton: The unspecified I/O automaton represents an I/O automaton for which any other I/O automaton has to be an implementation. Since our semantic models deal with external and quiescent trace inclusion, the unspecified I/O automaton has to exhibit all possible external and quiescent traces. Formally, for each sort S, we define an operator Ω_S taking no arguments. The relative transition rules are the following:

$$\textbf{ome}_1 \quad \Omega_S \xrightarrow{a} \Omega_S \quad a \in ext(S) \qquad\qquad \textbf{ome}_2 \quad \Omega_S \xrightarrow{\tau} nil_S$$

Rule $\textbf{ome}_1$ makes every string of $ext(S)^*$ an external trace of Ω_S. Rule $\textbf{ome}_2$ makes any trace a quiescent trace of Ω_S. Note that the use of rule $\textbf{ome}_2$ is the only way to move Ω to a quiescent state.

Prefixing operator: A classical method for specifying how a system should respond to a particular stimulus, or what action should be performed next, consists of using the prefixing operator. For each sort S and for each action $a \in ext(S)$ an operator $a._S$ is defined that takes an argument of sort S and yields an expression of sort S. Its transition rules are the following:

$$\textbf{pre}_1 \quad a ._S e \xrightarrow{a} e \qquad\qquad \textbf{pre}_2 \quad a ._S e \xrightarrow{b} \Omega_S \quad \forall b \in in(S) \backslash \{a\}$$

Here, rule $\textbf{pre}_1$ specifies the intuitive meaning of an I/O automaton $a\,.\,e$, i.e., $a\,.\,e$ can perform action a and then behave like e. Rule $\textbf{pre}_2$ deals with the input actions different from a. According to the demonic approach, $a\,.\,e$ specifies the behavior of a system only in the presence of action a; for all other inputs the behavior of the system is not specified, hence they give rise to a transition to Ω.

External choice operator: Often it is useful to build a system that offers a nondeterministic choice between two different actions and then behaves accordingly. In case the choice is between performing an output action o_1 and then behaving like e_1 or performing an output action o_2 and then behaving like e_2, an *external choice operator* like that of [10, 8] would suffice. However, this leads to problems when dealing with input actions. We can write

$$a\,.\,e_1 + b\,.\,e_2$$

In the presence of input a (resp. b) the above expression should move to e_1 (resp. e_2); in the presence of any other input action the above expression should move to Ω. However, due to the input enabling condition, we would have

$$a\,.\,e_1 \xrightarrow{b} \Omega \text{ and } b\,.\,e_2 \xrightarrow{a} \Omega$$

hence, the above expression would not respect our intuition about external choice.

A possibility for avoiding the above problem is parameterizing the choice operator with two sets of input actions I and J which contain those input actions, of the left and right operand respectively, which are meant to lead to a specified (different from Ω) behavior. For each sort S and each pair of sets $I, J \subseteq in(S)$ an operator ${}_I+^S_J$ is defined which takes two arguments of sort S and yields an expression of sort S. This operator is essentially the sorted version of Vaandrager's choice operator [21]. The transition rules for the new operators are the following:

$$\mathbf{ech}_1 \qquad \frac{e_1 \xrightarrow{a} e_1'}{e_1 \;{}_I+^S_J\; e_2 \xrightarrow{a} e_1'} \qquad \forall a \in I \cup out(S)$$

$$\mathbf{ech}_2 \qquad \frac{e_2 \xrightarrow{a} e_2'}{e_1 \;{}_I+^S_J\; e_2 \xrightarrow{a} e_2'} \qquad \forall a \in J \cup out(S)$$

$$\mathbf{ech}_3 \qquad e_1 \;{}_I+^S_J\; e_2 \xrightarrow{a} \Omega_S \qquad \forall a \in in(S) \backslash (I \cup J)$$

$$\mathbf{ech}_4 \qquad \frac{e_1 \xrightarrow{\tau} e_1'}{e_1 \;{}_I+^S_J\; e_2 \xrightarrow{\tau} e_1' \;{}_I+^S_J\; e_2}$$

$$\mathbf{ech}_5 \qquad \frac{e_2 \xrightarrow{\tau} e_2'}{e_1 \;{}_I+^S_J\; e_2 \xrightarrow{\tau} e_1' \;{}_I+^S_J\; e_2'}$$

Rules $\mathbf{ech}_{1,2}$ make explicit the fact that an argument can perform an input action only if it is in the corresponding parameter; rule $\mathbf{ech}_3$ expresses the fact that each input action not in $I \cup J$ is unspecified within a ${}_I+_J$ context. Rules $\mathbf{ech}_{4,5}$ state that an external choice context is not resolved with internal transitions.

Clearly, dealing with parameterized operators adds significantly to the complexity of our I/O calculus. It would be nice to have an unparameterized external choice operator. However, our attempts have failed; for all candidates we failed to achieve substitutivity for the external trace and quiescent preorders. Below, we give a counterexample for the intuitive external choice operator $(+)$ described at the beginning of this discussion. It is easy to verify that $nil \equiv_Q a\,.\,\Omega$ when a is an input action. However, $a\,.\,nil + nil \not\equiv_Q a\,.\,nil + a\,.\,\Omega$ since, if b is an output action of the signature of both processes above, ab is an external and quiescent trace of the right process but not an external, neither a quiescent, trace of the left one. The left process, in fact, according to its intuitive semantics, can only move to nil with the input action a, while the right one could also move to Ω.

The two basic operators Ω_S and nil_S defined above together with prefixing and internal choice and with recursive definition that we will introduce later, are sufficient for specifying all finitely branching input enabled transition systems. However, as a useful specification tool

and as a useful auxiliary operator for our axiomatization, we introduce an additional internal choice operator again based on [10, 8]. We will then describe the original combinators of [14].

Internal choice operator: For each sort S we define an operator $\oplus_S$ that takes two expressions of sort S and yields an expression of sort S. A process $e \oplus f$ nondeterministically evolves according to e or f. Thus, its external and quiescent traces are the union of the external and quiescent traces of e and f. The transition rules are the following:

$$\mathbf{ich}_1 \quad e_1 \oplus_S e_2 \xrightarrow{\tau} e_1 \qquad\qquad \mathbf{ich}_2 \quad e_1 \oplus_S e_2 \xrightarrow{\tau} e_2$$

$$\mathbf{ich}_3 \quad \frac{e_1 \xrightarrow{a} e_1'}{e_1 \oplus_S e_2 \xrightarrow{a} e_1'} \quad \forall a \in in(S) \qquad\qquad \mathbf{ich}_4 \quad \frac{e_2 \xrightarrow{a} e_2'}{e_1 \oplus_S e_2 \xrightarrow{a} e_2'} \quad \forall a \in in(S)$$

Rules $\mathbf{ich}_{1,2}$ express the fact that $e \oplus f$ nondeterministically behaves like e or f; rules $\mathbf{ich}_{3,4}$ are necessary for ensuring input enabling. Notice that, even if we generalize rules $\mathbf{ich}_{3,4}$ to all the external actions from S, rules $\mathbf{ich}_{1,2}$ cannot be eliminated when dealing with quiescent traces. Their elimination raise the problem that the empty trace be a quiescent trace of one argument but not a quiescent trace of the other.

Hiding operator: The hiding operator is similar to the corresponding operator defined on I/O automata. The main difference is given by the fact that all internal actions are converted into a single action, namely τ. For each sort S and each set $I \subseteq out(S)$ we define an operator τ_I^S taking an expression of sort S and yielding an expression of sort $(in(S), out(S) \backslash I, \{\tau\})$. The transition rules for hiding are the following:

$$\mathbf{tau}_1 \quad \frac{e \xrightarrow{a} e'}{\tau_I^S(e) \xrightarrow{a} \tau_I^S(e')} \quad a \notin I \qquad\qquad \mathbf{tau}_2 \quad \frac{e \xrightarrow{a} e'}{\tau_I^S(e) \xrightarrow{\tau} \tau_I^S(e')} \quad a \in I$$

Renaming operator: The renaming operator simply renames the actions of its argument. For each sort S and each injective mapping ρ with $\rho(\tau) = \tau$, we define an operator ρ_S taking an expression of sort S and yielding an expression of sort $(\rho(in(S)), \rho(out(S)), \{\tau\})$. The transition rule for renaming is the following:

$$\mathbf{rho} \quad \frac{e \xrightarrow{a} e'}{\rho_S(e) \xrightarrow{\rho(a)} \rho_S(e')}$$

Parallel operator: Also the parallel operator is defined according to the original definition of [14]. In particular, we comply with the restrictions on the sorts of its operands aiming at guaranteeing that each action be under the control of at most one process. Moreover, the transition rules agree with the specification given in Definition 2.9. Please notice that we do not have any condition on the internal actions of the arguments. This is because we are not dealing with multiple internal actions, but have a single invisible action. Thus compatibility of sorts reduces to:

Two sorts S_1, S_2 are *compatible* if $out(S_1) \cap out(S_2) = \emptyset$.

For each pair of processes with compatible sorts we define an operator ${}_{S_1}\|_{S_2}$ taking an expression of sort S_1 and one of sort S_2 and yielding an expression of sort S_3 where $out(S_3) = out(S_1) \cup out(S_2)$ and $in(S_3) = (in(S_1) \cup in(S_2))\backslash out(S_3)$. The transition rules for parallel composition are the following:

$$\textbf{par}_1 \quad \frac{e_1 \xrightarrow{a} e_1' \qquad e_2 \xrightarrow{a} e_2'}{e_1 \; {}_{S_1}\|_{S_2} \; e_2 \xrightarrow{a} e_1' \; {}_{S_1}\|_{S_2} \; e_2'}$$

$$\textbf{par}_2 \quad \frac{e_1 \xrightarrow{a} e_1'}{e_1 \; {}_{S_1}\|_{S_2} \; e_2 \xrightarrow{a} e_1' \; {}_{S_1}\|_{S_2} \; e_2} \quad a \in acts(S_1)\backslash ext(S_2)$$

$$\textbf{par}_3 \quad \frac{e_2 \xrightarrow{a} e_2'}{e_1 \; {}_{S_1}\|_{S_2} \; e_2 \xrightarrow{a} e_1 \; {}_{S_1}\|_{S_2} \; e_2'} \quad a \in acts(S_2)\backslash ext(S_1)$$

Recursion: Recursion within DIOA can be obtained in a De Simone style [4, 5]. Existence is assumed of a countable set $\mathcal{X}_S$ of process variables for each sort S and of a declaration mapping E associating a guarded expression of sort S with each process variable of $\mathcal{X}_S$. An expression e is *guarded* if each process variable occurs within the scope of a prefixing operator. Thus, $a\,.\,X$ and $\tau_{\{a\}}(a\,.\,X)$ are guarded expressions. The reason we want guarded expressions in the declaration mapping is to avoid definitions of the form $E(X) = X$ which, according to the transition rule we are introducing, would not be input enabled. The transition rule for process variables is the following:

$$\textbf{rec} \quad \frac{e \xrightarrow{a} e'}{X \xrightarrow{a} e'} \quad \text{if } X \stackrel{\text{def}}{=} e$$

where the notation $X \stackrel{\text{def}}{=} e$ means $E(e) = e$.

To offer a global view of our DIOA calculus, all the operators defined above, their signatures and their operational semantics are grouped in Tables 1 and 2 .

Below, we formalize a couple of basic properties of DIOA, which will be used to prove that the semantics we offer for our calculus provides an adequate interpretation of I/O automata. Indeed, Propositions 4.3 and 4.4 vindicate our choices.

Definition 4.1 (Sort consistency) A many-sorted calculus is *sort consistent* if the sort of every expression is invariant under transition. □

Definition 4.2 (Input enabledness) An expression e is *input enabled* if for each e' such that $e \xrightarrow{h} e'$ for some trace h, $in(e') \subseteq enabled(e')$. A many-sorted calculus with interfaces associated with expressions is *input enabled* if each expression is input enabled. □

Proposition 4.3 *DIOA is sort consistent.*

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	$out(S_1) \cap out(S_2) = \emptyset$ $out(S_3) = out(S_1) \cup out(S_2)$ $in(S_3) = (in(S_1) \cup in(S_2)) \backslash out(S_3)$
hiding	τ_I^S	S	S'	$I \subseteq out(S), S' = (in(S), out(S) \backslash I)$
renaming	ρ_S	S	S'	for each injective $\rho : acts(S) \longrightarrow acts(S')$ $S' = (\rho(in(S)), \rho(out(S)))$
process	X_S	λ	S	$X_S \in \mathcal{X}_S$

Table 1: The signature of DIOA

Proof. Simple induction argument on the depth of the proof tree of a transition. The base case is by cases analysis by considering rules **nil**, $\mathbf{ome}_{1,2}$ and $\mathbf{pre}_{1,2}$; the induction step is also by cases analysis by considering all the other transition rules. □

Proposition 4.4 DIOA is input enabled.

Proof. It is sufficient to show by structural induction that each DIOA expression e satisfies $in(e) \subseteq enabled(e)$. The fact that recursion is given through guarded expressions is crucial. □

Theorem 4.5 (Substitutivity) *External trace preorder and quiescent preorder are substitutive for DIOA.*

Proof. Since the internal choice operator does not fit Vaandrager's general format of [21], the proof of substitutivity must be given explicitly by considering each DIOA operator. We show as an example the case of the external choice operator for the external trace preorder. Let $e_1 \sqsubseteq_{ET} e_2$ and $f_1 \sqsubseteq_{ET} f_2$ and let t be an external trace of $e_1 \; {}_I+_J \; f_1$. If t is the empty trace then t is trivially an external trace of $e_2 \; {}_I+_J \; f_2$ since the empty trace is an external trace of any expression. Let $t = at'$. If a is an input action not in $I \cup J$ then rule $\mathbf{ech}_3$ applies

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$_1$	$a \cdot_S e \xrightarrow{a} e$		**pre**$_2$	$a \cdot_S e \xrightarrow{b} \Omega_S$	$\forall b \in in(S)\backslash\{a\}$
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$_1$	$\dfrac{e_1 \xrightarrow{a} e_1'}{e_1 {}_I{+}^S_J\, e_2 \xrightarrow{a} e_1'}$	$\forall a \in I \cup out(S)$			
ech$_2$	$\dfrac{e_2 \xrightarrow{a} e_2'}{e_1 {}_I{+}^S_J\, e_2 \xrightarrow{a} e_2'}$	$\forall a \in J \cup out(S)$			
ech$_3$	$e_1 {}_I{+}^S_J\, e_2 \xrightarrow{a} \Omega_S$	$\forall a \in in(S)\backslash(I \cup J)$			
ech$_4$	$\dfrac{e_1 \xrightarrow{\tau} e_1'}{e_1 {}_I{+}^S_J\, e_2 \xrightarrow{\tau} e_1' {}_I{+}^S_J\, e_2}$				
ech$_5$	$\dfrac{e_2 \xrightarrow{\tau} e_2'}{e_1 {}_I{+}^S_J\, e_2 \xrightarrow{\tau} e_1' {}_I{+}^S_J\, e_2'}$				
tau$_1$	$\dfrac{e \xrightarrow{a} e'}{\tau_I^S(e) \xrightarrow{a} \tau_I^S(e')}$	$a \notin I$	**tau**$_2$	$\dfrac{e \xrightarrow{a} e'}{\tau_I^S(e) \xrightarrow{\tau} \tau_I^S(e')}$	$a \in I$
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 2: The transition rules for DIOA

to $e_2 \ {}_I{+}_J \ f_2$ and t is trivially an external trace of $e_2 \ {}_I{+}_J \ f_2$. Suppose now without loss of generality that $a \in I \cup out(e_1)$ and rules $\mathbf{ech}_{2,3}$ are not used for the first a-transition of t. Then $e_1 \ {}_I{+}_J \ f_1 \stackrel{\tau^n}{\longrightarrow} e_1' \ {}_I{+}_J \ f_1' \stackrel{a}{\longrightarrow} e_1''$ for some $n \geq 0$ where t' is an external trace of e_1''. t is then an external trace of e_1, hence, by hypothesis, t is an external trace of e_2. In particular $e_2 \stackrel{\tau^m}{\longrightarrow} e_2' \stackrel{a}{\longrightarrow} e_2''$ for some $m \geq 0$ where t' is an external trace of e_2''. t is then an external trace of $e_2 \ {}_I{+}_J \ f_2$ since $e_2 \ {}_I{+}_J \ f_2 \stackrel{\tau^m}{\longrightarrow} e_2' \ {}_I{+}_J \ f_2 \stackrel{a}{\longrightarrow} e_2''$. Note that, alternatively, we could have converted DIOA into an equivalent calculus fitting Vaandrager's format by adding the necessary internal clearing rules to the internal choice operator. Its semantics in terms of external and quiescent traces, in fact, does not change. □

We conclude this section with a few remarks on alternative approaches we could have taken for defining calculi of I/O automata and proving their properties.

Given the calculus of DIOA, it is not difficult to convert it into an angelic calculus. The main changes consist in converting into self-loops all transitions to Ω which were used to capture our intuition about capturing underspecification. The rules to be replaced by self-loops are **nil**, $\mathbf{pre}_2$ and $\mathbf{ech}_3$. When this new approach is followed, the expression Ω can be eliminated from the calculus; it can be defined in terms of the other operators and of recursion.

Another issue we just want to mention is that of multiple internal actions; indeed, DIOA does not completely take into account the I/O automata requirement about this point. Our calculus allow only signatures with a single internal action. This restriction is not a serious one because in this paper we do not consider the problem of fair specifications. It would not be difficult to expand DIOA to permit multiple internal actions; however two main problems would have to be faced.

1. the preorder relations have to be defined over expressions with different sorts (all those sorts with the same external action signature)
2. substitutivity is no longer valid (if $P \equiv Q$ it might happen that there exists a process C such that $P \| C$ is legal while $Q \| C$ is not legal).

This would imply that a weaker variant of substitutivity has to be introduced that asks for preservation of equivalence within only those contexts in which both equivalent processes can be inserted.

The problem of defining a calculus with multiple internal actions is completely addressed in [17] where Vaandrager's work [21] is extended to the many-sorted setting. In [17] also the full details of the extended version of an angelic calculus of I/O Automata (called IOA) are completely worked out.

5 A Set of Laws for the External Trace Preorder

In this section we study a set of laws for the external trace preorder over DIOA expressions containing only nil, Ω, prefixing, external choice and internal choice. The first three operators

$\mathbf{O}_1$	$e \preceq_{ET} \Omega$
$\mathbf{O}_2$	$nil_S \simeq_{ET} a\,.\,\Omega$ if $a \in in(S)$
$\mathbf{O}_3$	$\Omega_S \simeq_{ET} \sum_{a \in out(S)} a\,.\,nil_S$ $\sum_{a \in \{a_1,\ldots,a_n\}} a\,.\,nil$ means $a_1\,.\,nil \,_{\emptyset}{+}_{\emptyset}\, a_2\,.\,nil \,_{\emptyset}{+}_{\emptyset} \cdots {}_{\emptyset}{+}_{\emptyset}\, a_n\,.\,nil.$

Table 3: Omega laws for DIOA.

$\mathbf{Ic}_1$	$e \oplus f \simeq_{ET} f \oplus e$
$\mathbf{Ic}_2$	$(e \oplus f) \oplus g \simeq_{ET} e \oplus (f \oplus g)$
$\mathbf{Ic}_3$	$e \simeq_{ET} e \oplus e$
$\mathbf{Ic}_4$	$a\,.\,(e \oplus f) \simeq_{ET} a\,.\,e \oplus a\,.\,f$
$\mathbf{Ic}_5$	$(e \oplus f) \,_I{+}_J\, g \simeq_{ET} (e \,_I{+}_J\, g) \oplus (f \,_I{+}_J\, g)$
$\mathbf{Ic}_6$	$\tau_I(e \oplus f) \simeq_{ET} \tau_I(e) \oplus \tau_I(f)$
$\mathbf{Ic}_7$	$(e \oplus f) \| g \simeq_{ET} (e \| g) \oplus (f \| g)$
$\mathbf{Ic}_8$	$e \preceq_{ET} e \oplus f$

Table 4: Internal choice laws for DIOA.

together with nil_S and Ω_S form the basic Input/Output Calculus; indeed, as we will see later, the other operators, renaming, hiding and parallel composition of Section 4, can be all described in terms of the basic ones.

We first introduce some simple laws, that are listed in Table 3, describing the main properties of Ω. Note that we do not give the soundness proofs of our laws since they are standard and in many cases follow directly from the definition of external trace.

Proposition 5.1 *The laws of Table 3 are sound for the external trace preorder.* □

The internal choice operator has simple and useful properties (see Table 4). It is commutative, associative, idempotent ($\mathbf{Ic}_1$ - $\mathbf{Ic}_3$); moreover every other DIOA operator distributes over it ($\mathbf{Ic}_4$ - $\mathbf{Ic}_7$). The meaning of $\mathbf{Ic}_8$ is immediate given the intuitive meaning of $\oplus$.

Proposition 5.2 *The laws of Table 4 are sound for the external trace preorder.* □

The last operator we need to axiomatize for providing a complete axiomatization of basic I/O automata is the external choice operator. For actually introducing its laws, we need two

$\mathbf{Ec}_1$	$e \;{}_I{+}_J\; f \simeq_{ET} f \;{}_J{+}_I\; e$
$\mathbf{Ec}_2$	$(e \;{}_I{+}_J\; f) \;{}_{I\cup J}{+}_K\; g \simeq_{ET} e \;{}_I{+}_{J\cup K}\; (f \;{}_J{+}_K\; g)$
$\mathbf{Ec}_3$	$e \simeq_{ET} e \;{}_I{+}_J\; nil$ if $Si(e) \subseteq I$ and $Si(e) \cap J = \emptyset$
$\mathbf{Ec}_4$	$e \;{}_I{+}_J\; f \simeq_{ET} (e \;{}_H{+}_K\; e) \;{}_I{+}_J\; f$ if $I \subseteq H \cup K$
$\mathbf{Ec}_5$	$e \;{}_I{+}_J\; g \preceq_{ET} (e \;{}_H{+}_K\; f) \;{}_I{+}_J\; g$ if $K \cap Si(f) \cap I \subseteq H$
$\mathbf{Ec}_6$	$(e \;{}_H{+}_K\; f) \;{}_I{+}_J\; g \preceq_{ET} e \;{}_I{+}_J\; g$ if $Quiet(f)$, $Si(e) \cap I \subseteq H$ and $K \cap Si(e) \cap I = \emptyset$
$\mathbf{Ec}_7$	$a \,.\, e \;{}_I{+}_J\; a \,.\, f \simeq_{ET} a \,.\, (e \oplus f)$ if $a \in out(e) \cup (I \cap J)$
$\mathbf{Ec}_8$	$e \;{}_I{+}_J\; f \preceq_{ET} e \oplus f$ if $Si(e) \cap Si(f) \subseteq I \cup J$
$\mathbf{Ec}_9$	$e \;{}_I{+}_J\; f \simeq_{ET} e \oplus f$ if $Si(e) \cup Si(f) \subseteq I \cap J$
$\mathbf{Ec}_{10}$	$(a \,.\, e \;{}_I{+}_J\; f) \oplus g \simeq_{ET} (a \,.\, e \;{}_I{+}_J\; f) \oplus (a \,.\, e \;{}_I{+}_K\; g)$ if $\begin{array}{c} Si(g) \subseteq K, \text{ and} \\ \{a\} \cap I \subseteq \{a\} \cap K \end{array}$
$\mathbf{Ec}_{11}$	$e \;{}_I{+}_J\; f \simeq_{ET} e \;{}_{I\setminus\{a\}}{+}_{J\setminus\{a\}}\; f$ if $a \in I \setminus Si(e)$.

Table 5: External choice laws for DIOA.

auxiliary functions which permit testing the behavior of I/O automata. They are defined as follows:

$$Si(e) = \{a \in in(e) | \exists t \in ext(e)^* : at \notin qtraces(e)\},$$

$$Quiet(e) = true \text{ iff } enabled(e) \subseteq in(e).$$

Function Si (*Specified input actions*) yields the set of input actions that an expression e can perform after any, possibly empty, sequence of invisible actions and such that the future behavior is not completely undefined. Intuitively an expression e is specifying a future behavior after an input action a only when not all possible implementations in response to a are correct implementations of the specification e.

Function $Quiet$ is much simpler since it simply checks whether a given expression does not enable locally controlled actions.

Evidence of the need of function Si for axiomatizing ${}_I{+}_J$ is given by the idempotence law for this operator. Indeed, according to our semantics it is in general not true that $e \equiv_{ET} e \;{}_I{+}_J\; e$ since for example, if a is an input action we have that $a \,.\, nil \not\equiv_{ET} a \,.\, nil \;{}_\emptyset{+}_\emptyset\; a \,.\, nil$. The needed side condition for the idempotence law is $Si(e) \subseteq I \cup J$.

Proposition 5.3 *The laws of Table 5 are sound for the external trace preorder.* □

$Quiet(nil)$
$a \in in(e)$ implies $Quiet(a\,.\,e)$
$Quiet(e)$ and $Quiet(f)$ implies $Quiet(e\ {}_I{+}_J\ f)$ and $Quiet(e \oplus f)$
$Quiet(E(X))$ implies $Quiet(X)$

Table 6: Axioms for $Quiet$.

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

Table 7: Definition of si for DIOA.

6 Completeness Proofs for the External Trace Preorder

In order to use the laws of the previous section, we need to determine whether side conditions of the form $Si(e) \cap J \subseteq I$ and $Quiet(e)$ do hold. Thus, to claim completeness of our set of laws we need also a complete set of rules for establishing truth of these two conditions. The proof system for function $Quiet$ is very simple as established by Lemma 6.1 below and the associated Table 6. For the auxiliary function $Si(e)$, we have not been able to find a simple proof system. We can, however, axiomatize a variant of Si, that we call si; this, thanks to a lemma which establishes that $Si(e) \subseteq si(e)$, provides a sound proof system for establishing whether the side conditions involving $Si(e)$ do hold.

Lemma 6.1 *The rules of Table 6 are sound and complete for function Quiet.* □

Table 7 contains the definition of the auxiliary function si; it is given in terms of the syntactic structure of an expression e. The intuitive idea behind its definition is that for all the input actions of e not belonging to $si(e)$ there is a visible transition to Ω, therefore $Si(e) \subseteq si(e)$.

Lemma 6.2 *For each DIOA expression e without renaming, hiding and parallel composition,* $Si(e) \subseteq si(e)$.

Proof. The lemma is a direct consequence of the assertion

$$\text{if } a \in in(e) \text{ and } a \notin si(e) \text{ then } e \stackrel{a}{\Longrightarrow} \Omega.$$

This assertion can be proved by induction on the complexity of guarded expressions e. For unguarded expressions it is sufficient to substitute $E(X)$ for each unguarded occurrence of a process variable X.

The base cases, nil and Ω, are trivial since, for any input action, they both have only transitions to Ω. For the other operators we have the following cases:

Case 1 prefixing:

Let $e \equiv a\,.e'$ and suppose $b \notin si(e)$ where $b \in in(e)$. By definition of si, $b \neq a$, hence the result is trivial since $a\,.e \stackrel{b}{\longrightarrow} \Omega$ for any input action b different from a.

Case 2 internal choice:

Let $e \equiv e_1 \oplus e_2$ and suppose $a \notin si(e)$ where $a \in in(e)$. By definition of si either $a \notin si(e_1)$ or $a \notin si(e_2)$. Suppose without loss of generality that $a \notin si(e_1)$. By induction $e_1 \stackrel{a}{\Longrightarrow} \Omega$. By first using rule $\mathbf{ich}_1$ we have $e_1 \oplus e_2 \stackrel{\tau}{\longrightarrow} e_1 \stackrel{a}{\Longrightarrow} \Omega$.

Case 3 external choice:

Let $e \equiv e_1 \;{}_I{+}_J\; e_2$ and suppose $a \notin si(e)$ where $a \in in(e)$. We distinguish the following cases.

1. $a \notin I \cup J$
   This case is trivial since $e_1 \;{}_I{+}_J\; e_2 \stackrel{a}{\longrightarrow} \Omega$.
2. $a \in I$
   By definition of si, $a \notin si(e_1)$, therefore we apply the induction hypothesis to e_1 and, since $a \in I$, we use rules $\mathbf{ich}_{1,4}$ to derive $e_1 \;{}_I{+}_J\; e_2 \stackrel{a}{\Longrightarrow} \Omega$.
3. $a \notin I$ and $a \in J$
   Similar to the previous case.

□

Function si has two main advantages; it relies on the syntactic structure of its arguments and can replace Si in all the laws of Section 5 without affecting soundness.

Theorem 6.3 *Let e be a DIOA expression without renaming, hiding and parallel composition, and let A, B be two sets of actions. Then*

$$A \cap si(e) \subseteq B \;\; \textit{implies} \; A \cap Si(e) \subseteq B.$$

□

This theorem amounts to saying that the laws of Tables 3, 4, and 5 together with Table 6 for quiet and Table 7 for *si*, provide a sound set of rules for establishing whether two I/O automata are external trace equivalent. Clearly the reader can develop his own sound rules to be used instead of those for *si*. We would like, however, to remark that function *si* as presented by Table 7 is sufficient to achieve completeness. In particular function *si* gives us the possibility of defining an unparameterized choice operator as

$$e + f \stackrel{\text{def}}{=} e \;{}_{si(e)}+_{si(f)}\; f,$$

which is commutative, associative and idempotent. This new operator $+$, although not substitutive, is useful for simplifying DIOA expressions and obtaining terms which do not contain the the parameters of the external choice operator. These are the source of many difficulties in the use of the algebraic laws for DIOA.

The simplified form to which each expression is reducible is called internal prefix form. It is defined below.

Definition 6.4 (Normal forms) A DIOA expression e is in *prefix normal form* if one of the following conditions holds.

1. $e \equiv \Omega$ or $e \equiv nil$ (atomic expressions)
2. $e \equiv a\,.e'$ where e' is in prefix normal form
3. $e \equiv e_1 + e_2$ where e_1 and e_2 are in prefix normal form but not atomic.

We denote an expression in prefix normal form different from Ω with $\sum_{i\in I} a_i\,.e_i$. If $I = \emptyset$ then the denoted expression is nil.

A DIOA expression e is in *internal prefix form* if $e \equiv e_1 \oplus \cdots \oplus e_n$ where each e_i is in prefix normal form. We abbreviate $e_1 \oplus \cdots \oplus e_n$ with $\sum\!\!\!\!\!\!\sum e_i$. □

The following lemmas show that, up to external trace equivalence, the internal normal form is closed under prefixing, internal choice and external choice. A consequence is that each DIOA expression not containing recursion, hiding, renaming and parallel composition is provably equivalent to an expression in internal prefix form. Thus, the completeness proof reduces to proving completeness only for expressions in internal prefix form.

Lemma 6.5 *If e is in internal prefix form then, for each $a \in ext(e)$, there is an expression e' in internal prefix form such that $a\,.e \simeq_{ET} e'$.*

Proof. Let $e \equiv \sum\!\!\!\!\!\!\sum_{i\in I} e_i$. By repeatedly applying $\mathbf{Ic}_4$ we obtain $a\,.(\sum\!\!\!\!\!\!\sum_{i\in I} e_i) \simeq_{ET} \sum\!\!\!\!\!\!\sum_{i\in I} a\,.e_i$. Since each e_i is in prefix normal form, then each $a\,.e_i$ is in prefix normal form, hence $\sum\!\!\!\!\!\!\sum_{i\in I} a\,.e_i$ is in internal prefix form. □

Lemma 6.6 *If e and f are in internal prefix form then $e \oplus f$ is in internal prefix form.*

Proof. Immediate consequence of the definition of internal prefix form. □

In order to easily prove the next closure property, we introduce some derived laws.

Proposition 6.7 *Let e, f and g be DIOA expressions. The following laws can be derived.*

$\mathbf{Ec}_{12}$ $e \preceq_{ET} e \,{}_I{+}_J\, f$ *if* $J \cap Si(f) \subseteq I$

$\mathbf{Ec}_{13}$ $e \,{}_I{+}_J\, f \preceq_{ET} e$ *if* $Quiet(f)$, $Si(e) \subseteq I$ *and* $Si(e) \cap J = \emptyset$

$\mathbf{Ec}_{14}$ $e \simeq_{ET} e \,{}_I{+}_\emptyset\, f$ *if* $Quiet(f)$ *and* $Si(e) \subseteq I$

$\mathbf{Ec}_{15}$ $e \,{}_I{+}_J\, g \simeq_{ET} (e \,{}_I{+}_K\, f) \,{}_I{+}_J\, g$ *if* $Quiet(f)$ *and* $K \cap I = \emptyset$

$\mathbf{Ec}_{16}$ $e \simeq_{ET} e \,{}_I{+}_J\, a\,.\,\Omega$ *if* $Si(e) \subseteq I$, $Si(e) \cap J = \emptyset$ *and* $a \in in(e)$

Proof. The laws $\mathbf{Ec}_{12,13}$ are a consequence of $\mathbf{Ec}_3$ and $\mathbf{Ec}_{5,6}$. The laws $\mathbf{Ec}_{14,15}$ follow from the combination of $\mathbf{Ec}_{5,6}$ with $\mathbf{Ec}_{12,13}$. The law $\mathbf{Ec}_{16}$ follows directly from $\mathbf{Ec}_3$ and $\mathbf{O}_2$. □

Lemma 6.8 *Let $e \equiv \sum_{i \in I} a_i\,.\,e_i$. Then $si(e) = \{a_i : i \in I\} \cap in(e)$.*

Proof. Direct application of the definition of si. □

Lemma 6.9 *If e and f are in prefix normal form then $e \,{}_I{+}_J\, f$ is provably equivalent to an expression e' in prefix normal form.*

Proof. Without loss of generality we can assume that e and f are different from Ω since, in such cases, $\mathbf{O}_3$ can be applied.

Let $e \equiv \sum_i a_i\,.\,e_i$ and let $f \equiv \sum_j b_j\,.\,f_j$.

From repeated applications of $\mathbf{Ec}_{11}$ there are two sets $I' \subseteq si(e)$ and $J' \subseteq si(f)$ such that

$$e \,{}_I{+}_J\, f \simeq_{ET} e \,{}_{I'}{+}_{J'}\, f.$$

We distinguish the following cases:

1. $\exists_i\, a_i \in I' \cup out(e)$ and $\exists_j\, b_j \in J' \cup out(f)$:

   Due to commutativity and associativity, the expressions e and f can be rewritten as $e' \,{}_{I'}{+}_{si(e)\setminus I'}\, e''$ and $f' \,{}_{J'}{+}_{si(f)\setminus J'}\, f''$, where $e' = \sum_{\{i : a_i \in I' \cup out(e)\}} a_i\,.e_i$, $e'' = \sum_{\{i : a_i \in si(e)\setminus I'\}} a_i\,.\,e_i$, $f' = \sum_{\{j : b_j \in J' \cup out(f)\}} b_j\,.\,f_j$ and $f'' = \sum_{\{j : b_j \in si(f)\setminus J'\}} b_j\,.\,f_j$. Note that e' and f' are in

prefix normal form, $si(e') = I'$, and $si(f') = J'$. By repeatedly applying $\mathbf{Ec}_{15}$ (together with $\mathbf{Ec}_2$) we obtain

$$\begin{array}{lll}(e' \,_{I'}{+}_{si(e)\setminus I'}\, e'')\,{}_{I'}{+}_{J'}\,(f' \,_{J'}{+}_{si(f)\setminus J'}\, f'') & \simeq_{ET} & e' \,_{I'}{+}_{J'}\,(f' \,_{J'}{+}_{si(f)\setminus J'}\, f'') \\ & \simeq_{ET} & (f' \,_{J'}{+}_{si(f)\setminus J'}\, f'')\,{}_{J'}{+}_{I'}\, e' \\ & \simeq_{ET} & f' \,_{J'}{+}_{I'}\, e' \\ & \simeq_{ET} & e' \,_{I'}{+}_{J'}\, f'.\end{array}$$

2. $\exists_i\, a_i \in I' \cup out(e)$ and $\not\exists_j\, b_j \in J' \cup out(f)$:

   In this case $J' = \emptyset$ (in fact $J \subseteq \bigcup_j \{a_j\}$) and $Quiet(f)$. From $\mathbf{Ec}_{15}$, $e \,_{I'}{+}_{J'}\, f \simeq_{ET} e' \,_{I'}{+}_{J'}\, f$ for some e' in prefix normal form with $si(e') = I'$. From $\mathbf{Ec}_{14}$ $e' \,_{I'}{+}_{J'}\, f \simeq_{ET} e'$.

3. $\not\exists_i\, a_i \in I' \cup out(e)$ and $\exists_j\, b_j \in J' \cup out(f)$:

   This case is similar to the previous one.

4. $\not\exists_i\, a_i \in I' \cup out(e)$ and $\not\exists_j\, b_j \in J' \cup out(f)$:

   In this case $I' = J' = \emptyset$, $Quiet(e)$ and $Quiet(f)$. From $\mathbf{Ec}_{2,14,15}$,

$$e \,_{\emptyset}{+}_{\emptyset}\, f \simeq_{ET} (e \,_{\emptyset}{+}_{\emptyset}\, nil)\,{}_{\emptyset}{+}_{\emptyset}\, f \simeq_{ET} (nil \,_{\emptyset}{+}_{\emptyset}\, e)\,{}_{\emptyset}{+}_{\emptyset}\, f \simeq_{ET} nil \,_{\emptyset}{+}_{\emptyset}\, f \simeq_{ET} nil.$$

This concludes the proof. □

Lemma 6.10 *If e and f are in internal prefix form then $e \,_I{+}_J\, f$ is provably equivalent to an expression e' in internal prefix form.*

Proof. Let $e \equiv \sum_{h\in H} e_h$ and $f \equiv \sum_{k\in K} f_k$. By a repeated application of $\mathbf{Ic}_5$ and $\mathbf{Ec}_1$ we obtain

$$(\sum_{h\in H} e_h) \,_I{+}_J\, (\sum_{k\in K} f_k) \simeq_{ET} \sum_{h\in H, k\in K} e_h \,_I{+}_J\, f_k.$$

From Lemma 6.9, for each h, k there is an expression $e_{h,k}$ in prefix normal form such that $e_h \,_I{+}_J\, f_k \simeq_{ET} e_{h,k}$, hence

$$(\sum_{h\in H} e_h) \,_I{+}_J\, (\sum_{k\in K} f_k) \simeq_{ET} \sum_{h\in H, k\in K} e_{h,k}.$$

The above expression is in internal prefix form. □

We now need to prove the completeness result for expressions in internal prefix form. Lemma 6.12 introduces a rule which is derivable from the laws and completely characterizes the external trace preorder over expressions in internal prefix form. First we state a simple lemma.

Lemma 6.11 *Let $e \equiv \sum_{i\in I} e_i$. Then $etraces(e) = \cup_{i\in I}\, etraces(e_i)$.*

Proof. Simple consequence of the transition rules for $\oplus$. □

Proposition 6.12 (A rule for inductive reasoning) *Let* $e \equiv \sum_i a_i.e_i$ *and* $f \equiv \Sigma_j\ f_j$ *where* $f_j \equiv \sum_k b_{j\,k}\,.\,f_{j\,k}$. *For each* a, j *let*

$$g_j^a \equiv \begin{cases} \Sigma_{b_{j\,k}=a}\ f_{j\,k} & \text{if } \{k|b_{j\,k}=a\} \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

Then $e \sqsubseteq_{ET} f$ *iff the following two conditions hold:*

1. $\forall i \left(e_i \sqsubseteq_{ET} \Sigma_{g_j^{a_i} \not\equiv \bot}\ g_j^{a_i} \text{ and } \exists j : g_j^{a_i} \not\equiv \bot \right)$ or $(a_i \in in(e) \text{ and } \exists j : g_j^{a_i} \equiv \bot)$
2. $\forall a \in \bigcap(si(f_j)) \setminus si(e) \quad \Omega \sqsubseteq_{ET} \Sigma_j\ g_j^a$

Proof.

If: Suppose conditions 1 and 2 to be valid. We perform the following external trace equivalence preserving transformations on e and f:

1. Using $\mathbf{Ec}_{16}$ add $a\,.\,\Omega$ to each expression f_j such that $a \notin si(f_j)$ and $a \in si(e) \cup si(f)$. Do the same on e.
2. Using $\mathbf{Ec}_{10}$ replicate on all the f_js each summand $a\,.\,f_k'$ of each f_k. For example, if $f \equiv (a\,.\,f_1' + f_1'') \oplus f_2 \oplus \cdots \oplus f_n$ then it becomes $(a\,.\,f_1' + f_1'') \oplus (a\,.\,f_1' + f_2) \oplus \cdots \oplus (a\,.\,f_1' + f_n)$
3. Using $\mathbf{Ec}_7$ group all expressions with a common prefix in each expression f_j of the new f.
4. Reduce each summand of the form $a\,.\,(\Omega \oplus \cdots)$ of each f_j to $a\,.\,\Omega$. This step is possible since it is immediate to prove $\Omega \simeq_{ET} e \oplus \Omega$ by using $\mathbf{O}_1$ and $\mathbf{Ic}_8$.

The above manipulations lead to two expressions e', f' with $e' \equiv_{ET} e$ and $f' \equiv_{ET} f$ where

$$e' \equiv e + \sum_{a \in si(f) \setminus si(e)} a\,.\,\Omega \quad \text{and} \quad f' \equiv \sum_j f_j'$$

for some expressions f_j'. Consider now

$$f'' \equiv (\sum_{a \in si(e) \cup si(f) \cup A} a\,.\,f_a'')$$

where $A = \{a \in out(e) | \exists_{j,k} a = b_{j\,k}\}$ and each f_a'' is

$$f_a'' \equiv \begin{cases} \sum_{g_j^a \not\equiv \bot} g_j^a & \text{if } a \in out(e) \text{ or } (a \in in(e) \text{ and } \nexists j | g_j^a \equiv \bot) \\ \Omega & \text{if } a \in in(e) \text{ and } \exists j | g_j^a \equiv \bot) \end{cases}$$

Consider a generic f_j' of f'. Due to the construction of the latter, each summand of f_j' is a summand of f'' and vice versa. As a consequence the external choice laws prove $f'' \simeq_{ET} f_j$ and, by repeatedly using $\mathbf{Ic}_3$, $f \simeq_{ET} f''$.

We now show that, for each summand $a.e''$ of e', $e'' \sqsubseteq_{ET} f''_a$. The law $\mathbf{Ec}_3$ and substitutivity are then sufficient to conclude $e' \sqsubseteq_{ET} f''$. If $\exists j | g^a_j \equiv \perp$ then $f''_a \equiv \Omega$ and $\mathbf{O}_1$ is sufficient to conclude; if otherwise, then $f''_a \equiv \sum_{g^a_j \not\equiv \perp} g^a_j$. If $a\,.\,e''$ is a summand of e then the conclusion follows from item 1 of the hypothesis; if otherwise then the conclusion follows from item 2 of the hypothesis after observing that $e'' \equiv \Omega$ and that $a \in \bigcap(si(f_j)) \setminus si(e)$.

Only if: Let $e \sqsubseteq_{ET} f$. We show that conditions 1 and 2 are satisfied.

1. Suppose condition 1 to be false and let i be one of the indexes for which the condition is false. We distinguish the following cases:

   (a) a_i is an output action

   In this case the left side of condition 2 must be false. If $\forall j : g^{a_i}_j \equiv \perp$, then no external trace with a_i as first action is an external trace for f, while a_i is an external trace of e. This gives a contradiction, hence $\exists j : g^{a_i}_j \not\equiv \perp$. Since condition 1 is false, it must be $e_i \not\sqsubseteq_{ET} (\sum_{g^{a_i}_j \not\equiv \perp} g^{a_i}_j)$. Let t' be an external trace of e_i but not of $\sum_{g^{a_i}_j \not\equiv \perp} g^{a_i}_j$. Clearly $t = a_i t'$ is an external trace of e. We show that t is not an external trace of f obtaining a contradiction. Suppose $f \stackrel{a}{\Longrightarrow} f'$ where t' is an external trace of f'. From the transition rules, $\exists j,k : f' \equiv f_{j\,k}$ and $a_{j\,k} = a_i$. By definition, $f_{j\,k}$ is a summand of $g^{a_i}_j$, hence t' is an external trace of $\sum_{g^{a_i}_j \not\equiv \perp} g^{a_i}_j$. This gives a contradiction.

   (b) a_i is an input action

   Since the right part of condition 1 must be false, then $\forall j : g^{a_i}_j \not\equiv \perp$. It is then enough to repeat the argument of the previous case to conclude.

2. Suppose condition 2 to be false. Then $\exists a \in \bigcap(si(f_j)) \setminus si(e) : \Omega \not\sqsubseteq_{ET} (\sum_j g^a_j)$. Let t' be an external trace of Ω but not of $\sum_j g^a_j$, and consider $t = at'$. Since from the transition rules and Lemma 6.8 $e \stackrel{a}{\longrightarrow} \Omega$, t is an external trace of e. By using the same argument as in case (b) of the proof for condition 1 we obtain that t is an external trace of $\sum_j g^a_j$. This gives a contradiction.

□

Lemma 6.13

$$e_1 \oplus \cdots \oplus e_n \sqsubseteq_Q f \;\; \textit{iff} \; \forall_{1 \le i \le n} e_i \sqsubseteq_Q f.$$

Proof. Direct consequence of Lemma 6.11. □

Proposition 6.14 (Completeness for expressions in internal prefix form) *Let e and f be expressions in internal prefix form. If $e \sqsubseteq_{ET} f$ then $e \preceq_{ET} f$.*

Proof. From Lemma 6.13 and $\mathbf{Ic}_3$ it is sufficient to analyze the case in which e is in prefix normal form. We show the result by induction on the maximum complexity n of e and f,

where the complexity of an expression is the maximum number of nested prefixing operators. If $n = 0$ then e and the summands of f are atomic expressions. By applying $\mathbf{Ic}_3$ we can suppose f to be nil or Ω or $\Omega \oplus nil$. If f is nil then either e is nil or $out(e) = \emptyset$. In the second case $\mathbf{O}_3$ applies. If f is Ω then $\mathbf{O}_1$ applies. If f is $nil \oplus \Omega$ then $\mathbf{Ic}_8$ applies.

For the induction step suppose $n > 0$. We can assume without loss of generality that e and any summands of f are different from Ω since $\mathbf{O}_3$ can be applied in such cases. By applying the rule of Proposition 6.12 to e and f we have that, for each condition involving the comparison of some expressions, one level of prefixing is eliminated, hence the complexity of the expressions to prove in relation is less than n. By applying the induction hypothesis and successively the rule of Proposition 6.12, we conclude that $e \preceq_{ET} f$. □

The main theorem is then the following.

Theorem 6.15 (Completeness for DIOA) *Let e and f be two recursion-free DIOA expressions without renaming, hiding and parallel composition operators. If $e \sqsubseteq_{ET} f$ then $e \preceq_{ET} f$.*

Proof. By means of Lemmas 6.5, 6.6, 6.10 and a simple induction argument the problem is reduced to the case in which e and f are in internal prefix form. The completeness result is then stated by Proposition 6.14. □

7 Axiomatizing Renaming, Hiding and Parallel Composition

In this section we consider the remaining three operators of DIOA and provide complete axiomatizations for them all.

7.1 Laws for Renaming

Axiomatizing the renaming operator is relatively easy; we can put forward laws that permit removing the renaming operator from any expression (see Table 8). Indeed, its laws show that the renaming operator distributes over every other operator and thus that it can be eliminated from any expression. It is worth remarking that not all the laws are necessary for proving completeness: some like $\mathbf{R}_{5,6,7}$ are reported only for continuity of presentation and for giving a fuller algebraic account of all operators.

Proposition 7.1 *The laws of Table 8 are sound for the external trace preorder.* □

$\mathbf{R}_1$	$\rho(nil) \simeq_{ET} nil$
$\mathbf{R}_2$	$\rho(a\,.\,e) \simeq_{ET} \rho(a)\,.\,\rho(e)$
$\mathbf{R}_3$	$\rho(e \oplus f) \simeq_{ET} \rho(e) \oplus \rho(f)$
$\mathbf{R}_4$	$\rho(e\ {}_I{+}_J\ f) \simeq_{ET} \rho(e)\ {}_{\rho(I)}{+}_{\rho(J)}\ \rho(f)$
$\mathbf{R}_5$	$\rho_1(\rho_2(e)) \simeq_{ET} \rho_1 \circ \rho_2(e)$
$\mathbf{R}_6$	$\rho(\tau_I(e)) \simeq_{ET} \tau_{\rho'(I)}(\rho'(e))$ if ρ' extends ρ
$\mathbf{R}_7$	$\rho(e \| f) \simeq_{ET} \rho(e) \| \rho(f)$
$\mathbf{R}_8$	$\rho(\Omega) \simeq_{ET} \Omega$

Table 8: Renaming laws for DIOA.

7.2 Laws for Hiding

In order to extend the completeness result to the hiding operator we need five additional laws. In this section we introduce eleven laws in order to show some interesting properties of the hiding operator independently of our final purpose. The laws that are used for the completeness proof are $\mathbf{I}_{1,2,3,4,11}$.

The law $\mathbf{I}_4$ uses an auxiliary function

$$So(e) = \{a \in out(e) | a \in etraces(e)\}$$

giving the set of *Specified output* actions of e, i.e., the set of output actions of e that could become enabled after some internal transitions. Note that it is not true in general that $\tau_I(e_H{+}_K\ f) \equiv_{ET} \tau_I(e)\ {}_H{+}_K\ \tau_I(f)$ since performing an action from I resolves the choice context in the left I/O automaton but does not resolve it in the right one.

Proposition 7.2 *The laws of Table 9 are sound for the external trace preorder.* □

Similarly as for function *Si*, function *So* can be provided with a sound proof system. In this paper we only give a rule for expressions in prefix normal form. The interested reader is referred to [18] for other rules.

Lemma 7.3 $So(\sum_{i \in I} a_i\,.\,e_i) = \{a_i | i \in I \text{ and } a_i \text{ is an output action}\}$. □

Lemma 7.4 *If e is in prefix normal form then $\tau_I(e)$ is provably equivalent to an expression e' in internal prefix form.*

$\mathbf{I}_1$	$\tau_I(nil) \simeq_{ET} nil$
$\mathbf{I}_2$	$\tau_I(\Omega) \simeq_{ET} \Omega$
$\mathbf{I}_3$	$\tau_I(a\,.\,e) \simeq_{ET} a\,.\,\tau_I(e)$ if $a \notin I$
$\mathbf{I}_4$	$\tau_I(e\ {}_H{+}_K\ f) \simeq_{ET} \tau_I(e)\ {}_H{+}_K\ \tau_I(f)$ if $So(e) \cap I = So(f) \cap I = \emptyset$
$\mathbf{I}_5$	$\tau_I(\tau_J(e)) \simeq_{ET} \tau_{I \cup J}(e)$
$\mathbf{I}_6$	$\tau_I(e) \| \tau_J(f) \simeq_{ET} \tau_{I \cup J}(e \| f)$ if $I \cap acts(f) = J \cap acts(e) = \emptyset$
$\mathbf{I}_7$	$\tau_I(e) \sqsubseteq_Q \tau_I(f)$ *implies* $\tau_I(a\,.\,e) \preceq_{ET} \tau_I(a\,.\,f)$
$\mathbf{I}_8$	$\tau_I(e) \sqsubseteq_Q \tau_I(g)$ *implies* $\tau_I(e\ {}_H{+}_K\ f) \preceq_{ET} \tau_I(g\ {}_H{+}_K\ f)$
$\mathbf{I}_9$	$\tau_I(e) \preceq_{ET} \tau_I(i\,.\,e\ {}_H{+}_K\ f)$
$\mathbf{I}_{10}$	$\tau_I(i\,.\,e) \simeq_{ET} \tau_I(e)$ if $Si(e) = \emptyset$
$\mathbf{I}_{11}$	$\tau_I(e\ {}_H{+}_\emptyset\ i\,.\,f) \simeq_{ET} \tau_I(e \oplus f)$ if $Si(e) \subseteq H$

Table 9: Hiding laws for DIOA.

Proof. We prove the proposition by induction on the complexity of an expression e. If e has complexity 0 then $\mathbf{I}_{1,2}$ are sufficient to conclude. Suppose now that e has complexity at most $n+1$, i.e., $e \equiv \sum_{k \in K} a_k\,.\,e_k$ where the complexity of each e_k is at most n. Let $K_1 = \{k \in K | a_k \notin I\}$ and $K_2 = K \backslash K_1$. Then

$$\begin{array}{lll} \tau_I(\sum_{k\in K} a_k\,.\,e_k) & \simeq_{ET} & \text{by the external choice laws} \\ \tau_I((\sum_{k\in K_1} a_k\,.\,e_k) + (\sum_{k\in K_2} a_k\,.\,e_k)) & \simeq_{ET} & \text{by } \mathbf{I}_{11} \text{ and } \mathbf{Ic}_6 \\ \tau_I(\sum_{k\in K_1} a_k\,.\,e_k) \oplus \sum\!\!\!\!\!\!\sum_{k\in K_2} e_k & \simeq_{ET} & \text{by } \mathbf{I}_{3,4} \\ (\sum_{k\in K_1} a\,.\,\tau_I(e_k)) \oplus \sum\!\!\!\!\!\!\sum_{k\in K_2} e_k\,. & & \end{array}$$

By induction each $\tau_I(e_k)$ has a provably equivalent expression e'_k in internal prefix form, hence

$$\tau_I(e) \simeq_{ET} (\sum_{k\in K_1} a_k\,.\,e'_k) \oplus \sum\!\!\!\!\!\!\sum_{k\in K_2} e'_k.$$

From Lemmas 6.5, 6.6 and 6.10, the above expression has a provably equivalent one in internal prefix form. □

Proposition 7.5 *If e is in internal prefix form then $\tau_I(e)$ is provably equivalent to an expression e' in internal prefix form.*

Proof. Let $e \equiv \sum\!\!\!\!\!\!\sum_{k\in K} e_k$ where each e_k is in prefix normal form. By $\mathbf{Ic}_6$ $\tau_I(e) \simeq_{ET} \sum\!\!\!\!\!\!\sum_{k\in K} \tau_I(e_k)$. By Lemma 7.4 each e_k has a provably equivalent expression e'_k in internal prefix form, hence $e \simeq_{ET} \sum\!\!\!\!\!\!\sum_{k\in K} e'_k$, which is in internal prefix form. □

$\mathbf{P}_1$	$e\|f \simeq_{ET} f\|e$
$\mathbf{P}_2$	$(e\|f)\|g \simeq_{ET} e\|(f\|g)$
$\mathbf{P}_3$	$\Omega\|\Omega \simeq_{ET} \Omega$

Table 10: Parallel laws for DIOA.

7.3 Laws for Parallel Composition

Some simple laws for the parallel operator in addition to those presented in the previous sections are listed in Table 10.

Proposition 7.6 *The laws of Table 10 are sound for the external trace preorder.* □

Unfortunately, when using the parallel operator, the notion of prefix normal form we used in the previous sections is no longer sufficient. Expressions of the form $\Omega\|nil\|\cdots\|nil$ cannot be reduced to normal form in general. For the above reason it is necessary to change the notion of atomic expression in the definition of the prefix normal form by saying that an atomic expression has the form $\Omega\|nil\|\cdots\|nil$.

The laws $\mathbf{O}_{2,3}$ are not sufficient for the new notion of atomic expression. The following two laws introduce a construction which is typical of interleaving semantic models. The law $\mathbf{E}_1$ is the needed extension of $\mathbf{O}_{2,3}$.

Proposition 7.7 (Expansion laws) *The following laws are sound for the external trace preorder.*

$\mathbf{E}_1$ *Let* $e \equiv \Omega_{S_0}\|nil_{S_1}\|\cdots\|nil_{S_n}$ *be of sort* S. *For each* $a \in out(S_0) \cup in(S)$ *let* e_a *be the unique state that* e *reaches with action* a. *Then* $e \simeq_{ET} (\sum_{a \in out(S_0) \cup in(S)} a\ .e_a)$

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

$$E_a^i = \begin{cases} \{e_{ij} | a_{ij} = a\} & \text{if } a \in acts(e_i) \\ \{e_i\} & \text{otherwise} \end{cases}$$

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

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

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

□

The proof of Proposition 6.14 is basically unchanged in its induction step. The main difference is that, instead of using $\mathbf{O}_3$ to eliminate subexpressions of complexity 0 containing Ω, $\mathbf{E}_1$ is used. The proof for the base of the induction, instead, needs one additional sound and complete rule.

Proposition 7.8 (Parallel law) *The following law is sound and complete for the external trace preorder:*

$\mathbf{P}_4$ *Let $e_i, 0 \leq i \leq n$ be atomic expressions and, for each action a, let f_i^a be the expression that e_i reaches with action a ($\perp$ if no expression exists). Then $e_0 \preceq_{ET} \sum_{1 \leq i \leq n} e_i$ iff, for each action a, either*

1. *$f_i^a \equiv e_i, 0 \leq i \leq n$ or*
2. *$f_0^a \equiv \perp$ or*
3. *$f_0^a \preceq_{ET} \sum_{f_i^a \not\equiv \perp} f_i^a$.*

Proof.

Only if: Suppose, for each action a, one of the conditions 1, 2 or 3 to be valid. Let t be an external trace of e_0. The case for $t = \lambda$ is trivial since λ is an external trace of any expression. Let $t = t_1 t_2$ where t_1 is the longest prefix of t such that each $e_i \xrightarrow{t_1} e_i$ by means of self loop transitions (due to the Ω parts). If $t_2 = \lambda$ then trivially t is an external trace of $(\sum_{1 \leq i \leq n} e_i)$. Suppose $t_2 = a t_3$ for some action a and let $e_0 \xrightarrow{a} f_0^a$. t_3 is then an external trace of f_0^a and, by hypothesis and the definition of t_2, t_3 is an external trace of $(\sum_{f_i^a \not\equiv \perp} f_i^a)$ and $\{f_i^a \not\equiv \perp\} \neq \emptyset$ (in fact conditions 1 and 2 are false). This implies that $\exists j : t_3$ is an external trace of f_j^a. Moreover $(\sum_{1 \leq i \leq n} e_i) \stackrel{\lambda}{\Longrightarrow} e_j \xrightarrow{t_1} e_j \xrightarrow{a} f_j^a$, hence t is an external trace of $(\sum_{1 \leq i \leq n} e_i)$.

If: Let $e_0 \sqsubseteq_{ET} (\sum_{1 \leq i \leq n} e_i)$ and suppose conditions 1, 2 and 3 to be false for some action a. Since, by condition 2, $f_0^a \not\equiv \perp$, we have that $e_0 \xrightarrow{a} f_0^a$. Since condition 3 is false, then either $\{f_i^a \not\equiv \perp\} = \emptyset$ or $f_0^a \not\sqsubseteq_{ET} (\sum_{f_i^a \not\equiv \perp} f_i^a)$. The first case cannot hold, for which otherwise a is an external trace of e_0 but not an external trace of $(\sum_{1 \leq i \leq n} e_i)$. Let $t = a t'$ where t' is an external trace of f_0^a but not an external trace of $(\sum_{f_i^a \not\equiv \perp} f_i^a)$. By definition, t is an external trace of e_0. We show that t is not an external trace of $(\sum_{1 \leq i \leq n} e_i)$. Suppose the contrary. By Lemma 6.11, t is an external trace of e_i for some $i > 0$. In particular $e_i \xrightarrow{a} f_i^a$, hence t' is an external trace of f_i^a, i.e., t' is an external trace of $\sum_{f_i^a \not\equiv \perp} f_i^a$, absurdum. □

The basic case of Proposition 6.14 is then proved by induction on the number of *nil* subexpressions appearing in the expressions to be compared. Note in fact that the number of *nil* subexpressions in the preconditions of $\mathbf{P}_4$ is strictly decreasing.

To deal with the hiding operator, $\mathbf{I}_{1,2}$ have to be extended to the new atomic expressions. The two new laws are the following.

Proposition 7.9 (Hiding laws) *Let e, f, g be DIOA expressions. The following laws are sound for the external trace preorder.*

$\mathbf{I}_{12}$ $\tau_I((\Omega_{S_0}\|nil_{S_1}\|\cdots\|nil_{S_n})\|e) \simeq_{ET} \tau_I(\Omega\|e)$ *if* $\forall_{1\leq j\leq n}(out(S_0)\cap in(S_j)\cap I)\backslash in(e)\neq\emptyset$

$\mathbf{I}_{13}$ $\tau_I(\Omega_{S_0}\|nil_{S_1}\|\cdots\|nil_{S_n}) \simeq_{ET} \Omega_{S_0\backslash I}\|nil_{S_1\backslash I}\|\cdots\|nil_{S_n\backslash I}$ *if* $\forall_{1\leq i\leq n} out(S_0)\cap in(S_i)\cap I=\emptyset$

□

Finally we have to state the closure property of the new internal prefix form under parallel composition. The closure property can be easily proven by induction on the complexity of the arguments of the parallel operator by noting that $\mathbf{E}_2$ reduces their complexity.

7.4 Dealing with Quiescence

The laws for the quiescent preorder are essentially the same as for the external trace preorder. The main problem in the formulation of the new laws is given by the possible quiescence of the empty trace. For the above reasons some of the laws must be restricted through some additional side conditions.

The external choice laws $\mathbf{Ec}_{5,6,9,10}$ need restrictions. The law $\mathbf{I}_{11}$ is no longer valid in general. As a consequence $\mathbf{I}_{7,8}$ are used in the completeness proof. The law $\mathbf{E}_1$ has to be changed in order to allow the empty trace to be a quiescent trace of the expanded expression. Its new form is

$$e \simeq_{ET} (\sum_{a\in out(S_0)\cup in(S)} a\,.\,e_a) \oplus (\sum_{a\in in(S)} a\,.\,e_a).$$

Finally the completeness rule needs an additional condition as follows:

$$Quiet(e) \text{ implies } \exists j|\ Quiet(f_j).$$

The complete axiomatization of the quiescent preorder is given in [18]. This section just gives an idea of how the axioms should be structured. The interested reader is referred to the cited bibliography.

8 Conclusion

We have presented a process algebra, called DIOA for Demonic I/O Automata, with the following main features: explicit interfaces are associated with each expression, a clear distinction is enforced between locally and globally controlled actions, input actions are always enabled, and actions are always under the control of at most one process. Our process algebra is directly related to I/O automata of Lynch and Tuttle [14], a model of distributed systems which has been successfully used for the specification and the verification of algorithms for distributed environments.

We have presented a set of algebraic laws which are sound with respect to the external trace preorder, a preorder which permits ignoring invisible actions and identifying those automata which can perform the same sequences of visible actions. These laws over DIOA have also been proved to be complete for recursion-free processes. We have also discussed possible extensions of the axiomatization to the quiescent preorder, a strengthened version of the external trace preorder that is sensitive to deadlock.

Indeed, further work has to be dedicated to investigating extensions of the completeness results to recursively defined expressions and to other preorder relations which are used in the operational framework of I/O automata. We are confident that the normal forms we have devised and the reduction techniques we have developed for proving completeness are a significative starting point and that they can naturally be extended to other behavioral relations.

Another interesting topic is that of fairness and of its relationship with quiescence; we will investigate when and how quiescence is sufficient for capturing fairness properties. We would also like to see how the quiescent and fair preorders relate to other well studied relations in non input enabled algebras, i.e., we would like to see how it is possible to embed notions such as input enabling, quiescence and fairness in non input enabled algebras.

Finally, we would like to see how a preorder relation can be thought of as an implementation relation. The fair preorder of I/O automata, in fact, is used as an implementation relation.

However, we have not found any formal justification of its use. Many times ad-hoc proof techniques are developed to deal with different verification tasks, and each time arguments have to be provided to convince the reader that the chosen technique corresponds to a correct notion of implementation. A formal understanding of the notion of implementation would avoid the above problem.

Some of the topics mentioned above are addressed in [18, 19] and will be the subject of a forthcoming paper; others are just proposals needing further investigations.

Acknowledgments: The paper started as [17] and is a simplification of chapters 3 and 4 of [18]; it owes its existence to [21]. Many thanks to Frits Vaandrager for suggesting the pursued lines of investigation. We would also like to thank Nancy Lynch and Albert Meyer for helpful comments and criticisms on draft copies.

References

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

[2] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced. In *Conference Record of the* 15^{th} *ACM Symposium on Principles of Programming Languages,* San Diego,

California, pages 229–239, 1988. Full version appeared as Technical Report TR 90-1150, Cornell University, Ithaca, 1990.

[3] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[4] R. De Simone. Calculabilité et expressivité dans l'algebra de processus parallèles MEIJE, 1984. Thèse de 3^e cycle, Univ. Paris 7.

[5] R. De Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[6] Nissim Francez. *Fairness*. Springer-Verlag, Berlin, 1986.

[7] 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.

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

[9] C.A.R. Hoare. A model of communicating systems. Technical report, Oxford University, 1981.

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

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

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

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

[14] 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.

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

[16] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer science Department, Aarhus University, 1981.

[17] R. Segala. Algebre di processi come automi con input e output, 1991. Tesi di laurea, Università di Pisa, Italy.

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

[19] 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.

[20] 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.

[21] 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.