Ithaca, NY

Bard Bloom*

TR 93-1369
August 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* Other members of the committee include: R. Cleaveland, R. van Glabbeek, I. Lee, N. Lynch, and F. Vaandrager.
Proceedings of the
North American
Process Algebra Workshop
1993
Ithaca, NY

PROGRAM COMMITTEE:
B. Bloom (chair)
R. Cleaveland
R. van Glabbeek
I. Lee
N. Lynch
F. Vaandrager
Contents

A Semantic Theory for ML Higher Order Concurrency Primitives
 Dominique Bolignano and Mourad Debabi

An Operational Semantics of Value Passing
 Rance Cleaveland

An Information Flow Security Property for CCS
 Riccardo Focardi and Roberto Gorrieri

Concurrent Kripke Structures
 Vineet Gupta

Specification of Instruction-Level Parallelism
 Ed Harcourt and Jon Mauney and Todd Cook

Specification of Transition Systems with Negation
 Rémi Lissajoux

A Comparison of Simulation and Algebraic Techniques for Verifying Concurrent Systems
 Nancy Lynch and Roberto Segala

A note on Model Checking Context Free Processes
 S. Purushothaman Iyer

State Refinement in Process Algebra
 Andrew Uselton and Scott Smolka

Parametric Preorders for Process Description Languages
 Daniel Yankelevich
A Semantic Theory for ML Higher Order Concurrency Primitives

Dominique Bolignano       Mourad Debabi

Bull Corporate Research Center,
78340 Les Clayes-Sous-Bois, FRANCE
D.Bolignano@frcl.bull.fr, M.Debabi@frcl.bull.fr

Abstract

In this paper we deal with the theoretical foundations of the unification of concurrent, functional and imperative programming paradigms. The intent is to present a denotational model for a language that is aimed to integrate all the three paradigms. Concurrency is supported through the embedding of a CSP-like process algebra into a functional language. Expressions may communicate through unidirectional channels and can also be composed through some concurrency combinators. The notion of reference is used to support imperative aspects. References, channels, and functions are made first-class citizens. Following an informal presentation of the language we will describe its static semantics. This semantics is effect-based i.e. for each expression it yields a type, the side effects and the communication effects. The use of types and effects allows a safe and efficient integration of the three programming paradigms and also solves some technical problems related to the foundations of the dynamic domains. Then the dynamic semantics is presented. It is denotational and rests on an extension of the mathematical model of acceptance trees.

1 Motivation and Background

The intent of this paper is to provide theoretical foundations for the unification of three computational paradigms which we refer to as concurrent, functional and imperative programming. A great deal of interest has been expressed in each of this programming styles and the underlying models have been deeply investigated, albeit generally separately.

Concurrency models have been a focus of interest for a great number of researchers. Accordingly, this gave rise to plenty of calculi and models. Prominent calculi are those that corresponds to process algebra such as: CCS (Calculus for Communicating Systems) [28, 27] and CSP (Communicating Sequential Processes) [17, 18] for which mathematically well-behaved models have been advanced. One can cite the failure-sets model of Brooks, Hoare and Roscoe [7, 8, 9] or the acceptance-trees model of Hennessy [13, 14]. However, in spite of the large activity of the concurrency community, it remains that formalisms and techniques devised for concurrent and distributed systems are generally relevant to pure processes, in other words, they focus on control aspects rather than data aspects. Thus, in such frameworks, there is no data, no communication, no states,...etc. These simplifications are generally adopted in order to put the emphasis on the difficulties inherent to concurrent systems, for instance, nondeterminism, the semantics of combinators,...etc.

On the other hand, functional programming has been extensively studied. Consequently, many powerful, general-purpose programming languages emerged such as ML dialects. The latter rests on secure theoretical foundations that are exemplified by the large body of results on pure and typed λ-calculus. Generally, functional languages are endowed with imperative features for efficiency reasons. Also programming without such facilities becomes quickly tedious and cumbersome in many situations.

There is an increasing need for programming languages and environments that combine all the three paradigms. This need is justified by the growing number of applications that operate in distributed environments. In addition, multiprocessor machines are very common today and the challenge is to develop appropriate languages that can take advantage from the computing power of such systems.

The work that will be presented hereafter, is meant to explore whether programming styles and conve-
nences evolved as part of Concurrent, Functional and Imperative programming could be somehow brought together to coexist in a single language. More accurately, the intention is to experiment our ideas on an ML-like language extended with concurrency features. Thus, the language described here supports polymorphic types. It supports also both functional and process abstractions as in CML [31, 32, 33] and FACILE [10]: functions may be used to describe internal computations of concurrent processes. Functions, processes, references and communication channels are first-class values and thus can be passed along channels. Consequently, the mobility of these values is supported.

At the theoretical level, we will present the static semantics of this language as well as the dynamic semantics. The type inference system is based on an extension of the type and effect discipline: a new approach to implicit typing that can be viewed as an extension of the ML-style type discipline. In addition to that, as shown in [36], effect-based type disciplines are more appropriate for integrating safely and efficiently functional and imperative programming. In this paper we will show that it contributes also significantly to the integration of concurrency features. The dynamic semantics presented here is denotational. It is based on an extension of both the RSL (Raise Specification Language [12]) models and VPLA (Value Passing Language with Assignment) of Hennessy [15, 16]. Thus, the presented model can be viewed as a CCS without τ's version.

The rest of the paper is organized as follows. A comparison with related approaches is given in Section 2. Then an informal description of the language is presented in Section 3. Section 4 is devoted to the denotational model as well as its algebraic properties. A few concluding remarks and a discussion of further research are ultimately sketched as a conclusion in Section 5. Part of the denotational definition of the language is given in the appendix.

2 Related Work

During the last decade, many proposals have been advanced for concurrent programming languages. OC-CAM [19] is a parallel imperative programming language that is an incarnation of the pioneering work of Hoare on CSP [17]. Recently, more modern languages have been proposed that reconcile the functional, concurrent and imperative styles. For instance one can cite CML [31, 32, 33], FACILE [10] and LCS [3]. All the three languages emerged from the idea of combining an SML-like language [30, 29] as a functional and imperative core, with a CCS or a CSP-like process algebra for process abstraction. They supports polymorphism, functional and process abstractions, dynamic behaviors and higher order objects.

These languages are quite expressive and as reported in [1], there is a need to have a semantic theory that enables one to reason about the programs and to grasp the meaning of sophisticated constructions. A structural operational semantics have been proposed for both CML [33] and FACILE [10]. Another description of FACILE semantics has been developed using the CHAM [2] (CHEmical Abstract Machine) framework [21]. In [5] we presented a structural operational semantics for our language.

2.1 Typing

The static semantics (typing semantics) in CML, FACILE and LCS rests on the type inference discipline. It is well known that this discipline, is problematic in the presence of non referentially transparent constructs. More precisely, the problem is relevant to type generalization in the presence of mutable data. Therefore, many extensions of the initial work of Milner [26] have been proposed.

The classical way to deal with this issue, is the imperative type discipline [37]. An extension of this approach has been used in the implementation of Standard ML of New Jersey. It is based on weak type variables: these type variables have an attached strength information, denoting the number of applications needed to get a non trivial effect. In [20], another method is proposed that consists in detecting some so called dangerous type variables (the ones occurring in the types of imperative objects), and labeling function types accordingly.

Later, in [36], the type and effect discipline is introduced. The latter yields as a result of the static evaluation of an expression, not only its principal type, but also all the minimal side effects. It should be noted that the idea of considering the effects as part of the static evaluation of an expression, has been suggested in [22] and adopted in the FX project [11, 23].

In [5] we proposed a new inference typing system that computes in addition to the principal types of expressions and their side effects, the minimal communication effects generated by the concurrent constructs. We have also presented an adequate operational semantics for our language and we proved that our typing system is consistent w.r.t. the static semantics.
2.2 Denotational Models

We are not aware of the existence of any denotational model for the previously mentioned languages, except for OCCAM. A denotational description of the latter is presented in [34]. However, we are interested here in some issues such as polymorphism, implicit typing and higher order objects, that are not supported by OCCAM. Another important issue we are dealing with here consists in providing a semantics for a language that allows process and functional abstractions.

Denotational models are first important to get a well understood foundations of the language. They also allow the semantics designer to extract the proof theory. They finally can serve as a model for the proof theory.

As pointed out before, in the literature, the two prominent denotational models for concurrency are the failure-sets model [7, 8, 9] and the acceptance-trees model [13, 14]. The failure-sets model has been designed as a semantic theory for a quite abstract version of CSP usually referred to as TCSP (Theoretical CSP). It supports only pure processes. The acceptance-trees model is very similar to the failure-sets model and has been devised as a model for TCSP or CCS-like abstract languages.

To deal with data aspects that are value-passing, assignment, return of results, store sharing as well as some control aspects such as sequencing, Robert Milne proposed an extension of the acceptance-trees model [24] as part of the RSL models. The whole RSL denotational semantics is presented in [25]. Inspired by [24], Hennessy and Ingólfsdóttir investigated value-passing and proposed a fully abstract model [15]. The same authors [16] proposed a semantic theory for an imperative language referred to as VPLA that supports value passing and assignments. In both [15] and [16] the authors presented three semantic approaches (i.e. denotational, axiomatic and operational) and proved their equivalence. The foundations as well as the algebraic properties of the RSL denotational models presented in [25] are detailed in [6].

To sum up, the language we present here, compares with CML or FACILE, (1) by the use and the extension of the "type and effect discipline" to obtain a more efficient type system, and (2) by the presentation of a denotational semantics. The primary objective of our work is to build upon the work done essentially by Hennessy, Ingólfsdóttir and Milne, respectively on VPLA and RSL, in order to get a denotational model for a real-life concurrent, functional and imperative language. Compared to RSL and VPLA we add in particular (1) implicit typing, (2) full higher order processes (which allows in particular to communicate references and channel values through channels) (3) and we take advantage of the effects provided by the type inference system for the denotational semantics definition.

3 Informal presentation

The syntactic constructions allowed in our language are close to those allowed in CML and FACILE. The set of expressions includes:

- Literals such as integers, booleans true and false, a distinguished value (), a constant skip which models an expression that immediately terminates successfully.

- Three binding operations that are the λ-abstraction, the recursion and the let definition.

- Imperative aspects are supported through the notion of reference. Expressions of the form ref (E) stands for the allocation of a new reference and assigns to it the value obtained by evaluating the expression E. We will use the unary operator ! for dereferencing and the binary operator := for assignment.

- Expressions may communicate through channels. The expression channel() means allocate a new channel. The expression E!E' means: evaluate E', evaluate E and send then the result of E' evaluation on the channel resulting from the evaluation of E. The whole expression evaluates then to (). The expression E? evaluates to any value received on the channel resulting from the evaluation of E. Notice that the communications are synchronized as in CCS and CSP.

- Three concurrency combinators:
  -[]: Nondeterministic (internal) choice.
  -[]: External choice.
  -[][]: Parallel composition of two expressions.

- A sequencing operator: [; ; ;].

More formally the BNF syntax of our language is:

\[
E ::= () \mid \text{true} \mid \text{false} \mid \text{Number}\ n \mid \text{ident}\ z \mid \text{skip} \mid \\
\lambda z \bullet E \mid E \ E \mid E \ E \mid E \ E \mid E \ E \ E \mid E \ E \ E \ E \mid \text{ref}\ E \mid E \ E \mid \text{if}\ E\ \text{then}\ E\ \text{else}\ E \mid \text{channel}() \mid E? \mid E!E \mid \\
\text{if}\ E\ \text{then}\ E\ \text{else}\ E \mid \text{let}\ z = E \text{ in } E \mid \\
\text{rec}\ z \bullet E
\]

3
In the following, we will use \( \mathcal{P}_f \) to stand for the finite powerset, \( A \rightarrow B \) for the set of all finite mappings (maps for short) from \( A \) to \( B \) and \( m \vdash m' \) for the overwriting of the map \( m \) with the map \( m' \). We will use the notation \([a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]\) to denote the map that associates the elements \( b_i \)'s to \( a_i \)'s.

4 Static semantics

As we pointed out before, we propose hereafter an extension of the type and effect discipline to give a static semantics to our language. The reader may refer to \([5]\) for the main motivations underlying this choice. We define the following static domains:

- **The domain of Reference regions:** The notion of reference regions is introduced to abstract memory locations. Every data structure corresponds to a region. Two values are in the same region if they may share some memory locations. The domain consists in the disjoint union of a countable set of constants and variables noted \( \gamma \). We will use \( \rho, \rho', \ldots \) to represent reference regions.

- **The domain of Reference effects:** Reference effects abstracts the memory side-effects. We define the following basic effects: 0 for the absence of effect, \( \zeta \) for a reference effect variable, \( \text{init}(\rho, \tau) \) for the reference allocation, \( \text{read}(\rho) \) for reading in the region \( \rho \) and \( \text{write}(\rho) \) for assignments of values to references in the region \( \rho \). We introduce also a union operator \( \cup \) for effects.

\[
\sigma ::= 0 | \zeta | \text{init}(\rho, \tau) | \text{read}(\rho) | \text{write}(\rho) | \sigma \cup \sigma
\]

We will write \( \sigma \sqsupseteq \sigma' \iff \exists \sigma'' : \sigma = \sigma' \cup \sigma'' \). Equality on reference effects is modulo ACI (Associativity, Commutativity and Idempotence) with 0 as the neutral element.

- **The domain of Channel regions:** As with reference regions, channel regions are intended to abstract channels. Their domain consists in the disjoint union of a countable set of constants and variables noted \( \delta \). We will use \( \chi, \chi', \ldots \) to represent values drawn from this domain.

- **The domain of Channel effects:** It is defined inductively by:

\[
\kappa ::= 0 | \eta | \text{chan}(\chi, \tau) | \text{in}(\chi) | \text{out}(\chi) | \kappa \cup \kappa
\]

We will use \( \eta \) to stand for a channel effect variable. The basic channel effect \( \text{chan}(\chi, \tau) \) represents the creation of a channel of type \( \tau \) in the channel region \( \chi \). \( \text{in}(\chi) \) denotes the effect resulting from an input on a channel of the channel region \( \chi \) while \( \text{out}(\chi) \) denotes an output on the channel of the region \( \chi \). We will write \( \kappa \sqsupseteq \kappa' \iff \exists \kappa'' : \kappa = \kappa' \cup \kappa'' \). Equality on effects is modulo ACI with 0 as the neutral element.

- The domain of types: It is inductively defined by:

\[
\tau ::= \text{Unit} | \text{Bool} | \text{Int} | \alpha | \text{ref}_\rho(\tau) | \text{chan}_\chi(\tau) \tau \xrightarrow{\sigma, \kappa} \tau
\]

\( \text{Unit} \) is a type with only one element “(”)\( , \alpha \) a type variable, \( \text{ref}_\rho(\tau) \) is the type of references in the region \( \rho \) to values of type \( \tau \), \( \text{chan}_\chi(\tau) \) is the type of channels in the communication region \( \chi \) that are intended to be mediums for values of type \( \tau \), \( \tau \xrightarrow{\sigma, \kappa} \tau' \) is the type of functions that take parameters of type \( \tau \) to values of type \( \tau' \) with a latent reference effect \( \sigma \) and a latent channel effect \( \kappa \). We mean by latent effect, the effect generated when the corresponding expression is evaluated.

We also define type schemes of the form \( \forall v_1, \ldots, v_n \bullet \tau \) where \( v_i \) can be type, reference region, channel region, reference effect and channel effect variable. A type \( \tau' \) is an instance of \( \forall v_1, \ldots, v_n \bullet \tau \) noted \( \tau' \prec \forall v_1, \ldots, v_n \bullet \tau \), if there exists a substitution \( \theta \) defined over \( v_1, \ldots, v_n \) such that \( \tau' = \theta \tau \). Our static semantics contain sequents of the form: \( E \vdash E : \tau, \sigma, \kappa \) which state that under some typing environment \( E \) the expression \( E \) has a type \( \tau \), a reference effect \( \sigma \) and a channel effect \( \kappa \). Notice that type environments \( E \) map identifiers to type schemes.

Type generalization in this type system states that a variable cannot be generalized if it is free in the type environment \( E \) or if it is present in the observed reference effect, or present in the observed communication effect. The first condition is classical while the two others are due to the fact that types are bound to regions in the effects. The reader should refer to \([36]\) for a detailed explanation of this issue.

\[
\text{Gen}(\sigma, \kappa, E)(\tau) =
\begin{align*}
\text{let } & \{ v_1, \ldots, v_n \} = f v(\tau) \backslash (f v(E) \cup f v(\sigma) \cup f v(\kappa)) \\
\text{in } & v_1, \ldots, v_n \bullet \tau \end{align*}
\]

where:

\[
\begin{align*}
fv(E) &= \{ f v(\tau) \mid \exists x \bullet x \mapsto \tau \in E \} \\
f v(\sigma) &= \{ f v(\tau) \mid \exists \rho \bullet \text{init}(\rho, \tau) \in \sigma \}
\end{align*}
\]
\( f_v(\kappa) = \{ f_v(\tau) \mid \exists \chi \cdot \text{chan}(\chi, \tau) \in \chi \} \)
\( f_v(\tau) = \{ x \mid x \text{ appears free in } \tau \} \)

The observation criterion has been introduced in order to report only reference effects that can affect the context of an expression.

\[
\text{Observe}(\mathcal{E}, \tau, \sigma) = \{ \epsilon \in \sigma | \epsilon \in f_v(\mathcal{E}) \cup f_v(\tau) \} \cup \\
\{ \text{init}(\rho, \varrho), \text{read}(\rho), \text{write}(\rho) \in \sigma | \rho \in \text{fr}_r(\mathcal{E}) \cup \text{fr}_r(\tau) \}
\]
\[
\text{Observe}(\mathcal{E}, \tau, \kappa) = \{ \eta \in \kappa | \eta \in f_v(\mathcal{E}) \cup f_v(\tau) \} \cup \\
\{ \text{chan}(\chi, \varrho), \text{in}(\chi), \text{out}(\chi) \in \kappa | \chi \in \text{fr}_c(\mathcal{E}) \cup \text{fr}_c(\tau) \}
\]

where \( \text{fr}_r(\mathcal{E}) \) and \( \text{fr}_c(\mathcal{E}) \) stand respectively for free reference regions and free communication regions in \( \mathcal{E} \):
\[
\text{fr}_r(\mathcal{E}) = \{ \rho \mid \exists x : x \mapsto \tau \in \mathcal{E} \land \rho \in f_v(\tau) \}
\]
\[
\text{fr}_c(\mathcal{E}) = \{ \chi \mid \exists x : x \mapsto \tau \in \mathcal{E} \land \chi \in f_v(\tau) \}
\]

In what follows let us give some examples on the static evaluation of the expressions.

**Example 4.1** Let us consider a process that creates a reference to the number 1, a channel and then put in parallel an output expression and an input expression followed by an assignment.

\[
\begin{align*}
\text{let } & x = \text{ref } 1 \\
\text{in } & \text{let } c = \text{channel } () \\
& \text{in } x = c ? \ || \text{!}7 \\
& \text{end}
\end{align*}
\]

Let us denote by \( P \) the previous process. Without using the observation criterion the type of the previous expression under an empty typing environment is:

\[
[ ] \vdash P : \text{Unit,} \\
\text{init}(\rho, \text{Int}) \cup \text{write}(\rho), \\
\text{chan}(\chi, \text{Int}) \cup \text{in}(\chi) \cup \text{out}(\chi)
\]
After application of the observation criterion the type of $P$ becomes:

$$[] \vdash P : \text{Unit}, \emptyset, \emptyset$$

**Example 4.2** If we consider an expression similar to the previous one that returns the created channel value after the parallel composition:

```haskell
let \(x=\text{ref} 1\)
    in
    let \(c=\text{channel } ()\)
        in \((x:=c? ||c!7);c\)
    end
end
```

Let us denote by $Q$ the previous process. Without using the observation criterion the type of $Q$ under an empty typing environment is:

$$[] \vdash Q : \text{chan}_x(\text{Int}),$$

$$\begin{align*}
\text{init}(\rho, \text{Int}) & \cup \text{write}(\rho), \\
\text{chan}(\chi, \text{Int}) & \cup \text{in}(\chi) \cup \text{out}(\chi)
\end{align*}$$

After application of the observation criterion:

$$[] \vdash Q : \text{chan}_x(\text{Int}),$$

$$\begin{align*}
\emptyset, \\
\text{chan}(\chi, \text{Int}) & \cup \text{in}(\chi) \cup \text{out}(\chi)
\end{align*}$$

5 Denotational Semantics

One of the main novelties of this work, is to provide a denotational model for such a language that can be easily adapted for other concurrent functional languages such as Facile or CML. The technique considered here takes advantage of the existing work on more abstract process algebra. More precisely, by importing the acceptances model\[13][14] which is known to be mathematically well-behaved in the applicative world of process algebra, and extending this model for handling input, output, imperative aspects and higher order objects.

It should be noticed that many adaptations of this model have been proposed in the literature. The first one has been proposed by [24], further explained in [25] whose foundations are detailed and proven in in [6], in order to design the models of RSL [12]. Also inspired by the original RSL denotational description [24], [15] and [16] proposed other accommodations respectively for their languages VPL and VPLA.

### 5.1 Acceptance-trees Model

The intention hereafter is to recall briefly the general model of acceptances trees. Let $\Sigma$ be a set of events which the processes can perform. In the rest of this section, we need the notion of saturated sets. Thus a set $A \subseteq \mathcal{P}_f(\Sigma)$, is said to be saturated if it satisfies the following closures:

1. $\bigcup A \in A \quad \text{union closure}$
2. $A, B \in A$ and $A \subseteq C \subseteq B$ implies $C \in A$ convex closure.

The set of all saturated finite subsets of $\mathcal{P}_f(\Sigma)$, all saturated sets over $\Sigma$, is denoted by $\text{sat}(\Sigma)$. Let $c(A)$ be the saturated closure of a set $A \subseteq \mathcal{P}_f(\Sigma)$, defined as the least set which satisfies:

1. $A \subseteq c(A)$
2. $\bigcup A \in c(A)$
3. $A, B \in c(A)$ and $A \subseteq C \subseteq B$ implies $C \in c(A)$

The process space $D$ is then defined as:

$$D = \{(m, S) | \ (m, S) \in$$

$$(\Sigma_{ref} D \times \text{Sat}(\Sigma)) \bullet$$

$$\text{dom}(m) = \bigcup S$$

$$\{\bot\}$$

Thus $D$ is a set of pairs, each of which represents a process. The first component of a pair is an association (a map) between finitely many members of $\Sigma$ and members of $D$, while the second one is an acceptance set. It should be noted that the definition given above is recursive, but such a domain exists and satisfies elegant algebraic properties such as the Scott-domainhood property [13].

Now, let us come back to the underlying intuition. A process can be modeled as a pair $(m, S)$ which stipulates that the process is waiting to engage in one event, say $e$, chosen from $\text{dom}(m)$, and once chosen, the process continues progressing as the process $m(e)$. We will refer to $m(e)$ as the sequel of the process.
The choice of the event is governed by the corresponding acceptance set. In fact $S$ stands for the set of the possible internal states that can be reached nondeterministically. The actions in a set $A$ from $S$, are those which the process can perform when in that state.

The special element $\perp$ corresponds to the most nondeterministic process, known in the literature as chaos. In other words it models divergent processes. We consider here a bounded non-determinism, so at any given time, a process can be waiting to participate in one of only finitely many events. That it is why $dom(m)$ and the relevant internal states of an accept ance set are restricted to be finite.

5.2 The Adapted Model

In the sequel of this section, the aim is to recall briefly this model. In the latter, the processes are able to perform one-to-one communications through uni-directional channels. The adapted model makes distinction between events in order to handle input and output. Thus, events are not simply assumed to range over an alphabet of actions, but are split into two categories according to their direction. So input and output events are distinguished and correspond respectively to inputs and outputs from certain channels.

Another adaptation, due to communication and sequencing handling, consists of two kinds of sequel processes (or just sequels). In fact sequels are not simply identified, as previously, with processes, but are also classified according to the nature of the relevant event that just occurred. The sequel attached to an input communication is modeled as a $\lambda$-abstraction that takes a value of the associated channel's type, together with some store, to some process that represents the continuation process. The value received on the channel is accompanied with some store, simply because the value may contain some references that are bounded to some values in the store. An output sequel is modeled as a map that associates to the values that may be communicated (together with their corresponding stores) some continuation processes.

We will consider as a result of a process execution a pair composed of a value and a store. But since processes may be nondeterministic, then we will consider the set of possible outcomes i.e. the set of possible results. The semantic technique used in the construction of the dynamic domains makes a dependence between the static and the dynamic semantics. Making the dynamic domains depend on the static domains means that dynamic domains are henceforth typed according to the hierarchy laid down by the static domains. Thus, dynamic spaces are restricted such that, for example, dynamic value domains are not obliged to contain as elements, functions that take members of these domains as parameters. In addition to that, the static constraints relevant to the well-formedness of an expression will be used in the dynamic semantics. This ensures that the dynamic domains are all that are needed. In what follows, we present the construction of the dynamic spaces, from the static ones.

The value dynamic domain is inductively defined by table 2. All the types used in these dynamic domains are ground i.e. without any type, region or effect variable.

We define the following functions on effects:

$$rw(\sigma) = \{ref_\rho(\tau) | (\text{read}(\rho,\tau) \text{ or write}(\rho,\tau) \in \sigma) \text{ and } \text{init}(\rho,\tau) \notin \sigma\}$$

$$wr(\sigma) = \{ref_\rho(\tau) | \text{write}(\rho,\tau) \in \sigma \text{ and } \text{init}(\rho,\tau) \notin \sigma\}$$

$$\text{in}(\kappa) = \{\text{chan}_\chi(\tau) | \text{in}(\chi,\tau) \in \kappa\}$$

$$\text{out}(\kappa) = \{\text{chan}_\chi(\tau) | \text{out}(\chi,\tau) \in \kappa\}$$

where $rw(\sigma)$ and $wr(\sigma)$ stand respectively for the read/written and written references of the reference effect $\sigma$, $\text{in}(\kappa)$ and $\text{out}(\kappa)$ stand respectively for the input and output channels of the communication effect $\kappa$.

Now let us turn to the process dynamic domain: A process can be viewed as a 4-uple, where the first component is the set of possible results. The second component maps an input event to the corresponding continuation process (a $\lambda$-abstraction) while the third component maps an output event to the corresponding continuation process. At any given time, a process can be waiting to participate in one of only finitely many events. The choice of the event is governed by the fourth component (acceptance set). In fact it is a set of subsets of events and stands for the set of possible internal states that can be reached nondeterministically.

$$D_{\tau,\sigma,\kappa} = (P_f(V_{\tau} \times S_{rw(\sigma)}) \times \text{Input}\_\text{map}_{\tau,\sigma,\kappa} \times \text{Output}\_\text{map}_{\tau,\sigma,\kappa} \times \text{Acceptance}\_\text{set}_{\tau,\sigma,\kappa}) \cup \{\perp\}$$

where:
with the following invariant:
\[ \forall s \in S \cdot \forall \tau \rangle \exists i \in \text{dom}(s) \cdot s(\tau, \tau, i) \in V. \]

Table 2: The value dynamic domain

- The domain of input maps \( Input_{\text{map}}_{\tau, \sigma, \kappa} \) is defined as the set of all maps:
  \[
  (\bigcup_{\text{chan}_{\tau}(x) \subset \text{in}(\kappa)} V_{\text{chan}_{\tau}(x)}) \parallel
  (\bigcup_{\text{chan}_{\tau}(x) \subset \text{in}(\kappa)} (V_{\tau} \times S) \rightarrow D_{\tau, \sigma, \kappa})
  \]
  which satisfies:
  \[
  \forall I \in Input_{\text{map}}_{\tau, \sigma, \kappa} \cdot \forall c(\chi, \tau', i) \in \text{dom}(I) \cdot
  I(c(\chi, \tau', i)) \in (V_{\tau} \times S) \rightarrow D_{\tau, \sigma, \kappa}
  \]
  where \( S = \bigcup_{\tau} S_{\tau} \).

- The domain of output maps \( Output_{\text{map}}_{\tau, \sigma, \kappa} \) is defined as the set of all maps:
  \[
  (\bigcup_{\text{chan}_{\tau}(x) \subset \text{out}(\kappa)} V_{\text{chan}_{\tau}(x)}) \parallel
  (\bigcup_{\text{chan}_{\tau}(x) \subset \text{out}(\kappa)} V_{\tau} \times S \parallel D_{\tau, \sigma, \kappa})
  \]
  which satisfies:
  \[
  \forall O \in Output_{\text{map}}_{\tau, \sigma, \kappa} \cdot \forall c(\chi, \tau', i) \in \text{dom}(O) \cdot
  O(c(\chi, \tau', i)) \in (V_{\tau} \times S) \parallel D_{\tau, \sigma, \kappa} \land
  \text{dom}(O(c(\chi, \tau', i))) \neq \emptyset
  \]

- The domain of acceptance sets \( Acceptance_{\text{set}}_{\tau, \sigma, \kappa} \) is defined as the set of all sets:
  \[
  \mathcal{P}_{f}(\mathcal{P}_{f}(\bigcup_{\text{chan}_{\tau}(x) \subset \text{in}(\kappa)} V_{\text{chan}_{\tau}(x)}) \times
  \mathcal{P}_{f}(\bigcup_{\text{chan}_{\tau}(x) \subset \text{out}(\kappa)} V_{\text{chan}_{\tau}(x)}) \times
  \mathcal{P}_{f}(\{x\})
  \]

The latter complicated condition on the acceptance set is just the combination of the convex and the union closures. Notice that \( \bot \) corresponds to the most nondeterministic process, known in the literature as chaos. It models divergent processes. We introduced also a special event denoted by "\( \check{\vee} \)" (pronounced tick) to signal the successful termination of a process. When such an event occurs, the process immediately terminates and then returns some value.

We define the following preorder on the dynamic process domain:

1. \( \forall P \in D_{\tau, \sigma, \kappa} \cdot \bot \leq_{\tau, \sigma, \kappa} P \)
2. \( (R, I, O, A) \leq_{\tau, \sigma, \kappa} (R', I', O', A') \) iff:
   (a) \( R' \subseteq R \land A' \subseteq A \)
   (b) \( \text{dom}(I') \subseteq \text{dom}(I) \land \text{dom}(O') \subseteq \text{dom}(O) \)
(c) \( \forall x \in \text{dom}(I') \cdot I(x) \leq I'(x) \)

to: \( \forall (v, s) \in V \times S \cdot I(x)(v, s) \subseteq_{\tau, \sigma, \kappa} I'(x)(v, s) \)

(d) \( \forall y \in \text{dom}(O') \cdot O(y) \leq O'(y) \)

to: \( \text{dom}(O'(y)) \subseteq \text{dom}(O(y)) \land \forall (v, s) \in \text{dom}(O'(y)) \cdot O(y)(v, s) \subseteq_{\tau, \sigma, \kappa} O'(y)(v, s) \)

The process domain is reflexive. Its existence as well as its algebraic nature can be easily derived from the proof of the process domain presented in [6]. The latter uses the general theory of domains [35]. In fact, we have a recursive domain specification of the form \( D = F(D) \). In order to show both the existence as well as the algebraicity, we have to show that our tree constructor “F” can be turned into a continuous functor in the category of cpo’s CPO and then define the process domain \( D \) as the least fixpoint of the functor \( F \) (i.e. the initial object in the category of F-algebras). Afterwards, we have to show that \( F \) preserves algebraicity.

**Proposition 5.1** \( \forall \tau, \sigma, \kappa \cdot (D_{\tau, \sigma, \kappa}, \subseteq_{D_{\tau, \sigma, \kappa}}) \) is an algebraic cpo whose compact elements are the processes \( d \in D_{\tau, \sigma, \kappa} \) satisfying the following conditions:

1. \( d \) is \( \perp \) or,

2. \( d \) is of finite depth, and all the input mappings that are subterms of \( d \) associate non \( \perp \) processes to only a finite number of elements.

5.3 Handling Polymorphism

The presence of polymorphism in the studied language introduces additional complexity. In the absence of polymorphism the semantics of an expression can be modeled as a function that takes an environment, and some current store to the process that represents the computation associated with the expression. For instance, the monomorphic expression \( 1 + 2 \) can be modeled as:

\[
\lambda \Gamma \cdot \lambda s \cdot ((3, s), [], [], \{(0, \emptyset, \{\emptyset\})\})
\]

This means that the expression \( 1 + 2 \) can be viewed as a function that takes an environment, and some initial store and then returns the value 3 without changing the initial store and without communicating. It terminates afterwards immediately.

Since expressions may be polymorphic, then the semantics depends on the considered instance. To express such a dependence, the technique we used, borrows one form of dependent types "general product" noted \( \Pi \) from constructive type theory. In doing this we associate a standard denotation to each completely instantiated type.

5.4 Semantics Presentation

In order to describe type and effect instantiation we will use ground substitutions. \( GS_\cdot(X) \), where \( X \) is a set of type and effect variables, is meant to be the set of ground substitutions whose domains include \( X \). \( GS_\cdot(\emptyset) \) will stand for the identity substitution written \( \theta_{id} \).

Then for an expression \( E \) of type \( \tau, \sigma, \kappa \), under some static environment \( E \), the signature of the semantic function that performs its dynamic denotational evaluation is:

\[
\begin{align*}
\text{DynEnv}_\cdot(\theta E) & \rightarrow S \rightarrow D_{\theta^\tau_\theta \sigma, \theta^\sigma_\theta, \theta^\kappa_\theta} \\
\end{align*}
\]

with,

\[
\begin{align*}
\text{DynEnv}_\cdot(\theta E) = \\
\{ \Gamma | \text{dom}(\Gamma) = \text{dom}(\theta E) \land \\
\forall x \in \text{dom}(\Gamma) \cdot \\
\text{let} \\
\forall v_1, \ldots, v_n \cdot \tau = \theta E(x) \\
\text{in} \\
\Gamma(x) \in \prod \theta' \in GS_\cdot(\{v_1, \ldots, v_n\}) \cdot V_{\theta'^\tau} \\
\end{align*}
\]

which means that we consider a ground instance \( \theta^\tau_\theta \sigma, \theta^\sigma_\theta, \theta^\kappa_\theta \) of the principal type \( \tau, \sigma, \kappa \) and then we take an environment and an initial store to some process that represents the computation associated with the expression.

As pointed before, the description technique used relies on a dependence between the static and the dynamic semantics. Accordingly, the semantic rules hereafter will be structured in two parts. The first part expresses the static evaluation i.e. it gives the typing of both the expression and the corresponding subexpressions.

The second part of the semantic rules is devoted to the denotational dynamic evaluation.

6 Conclusion

We have reported in this paper the complete definition of an implicitly strongly typed polymorphic con-
current and functional language that supports data accepting in-place modification. We have presented a complete static semantics that rests on an extension of the type and effect discipline to handle communication effects. Afterwards we have presented a denotational interpretation by accommodating the model of acceptance trees.

It should be noted that an operational semantics was also defined for this language in [5]. The full abstractness of the denotational model w.r.t. the operational semantics is under way. As future research, we plan to investigate the structuring and modularity mechanisms. We are particularly interested in experimenting some new approaches in modularity from the algebraic specification world such as the loose stratified semantics proposed by [4].

References


Appendix A: Some Semantic Function

Hereafter some semantic functions that are used in the semantic description and that contribute to the concision of the presentation.

Some Useful Semantic Functions

/* The function “pass” takes a value and a store, and returns a process that has the two arguments as a result. The process does not communicate in input or in output, it just terminates immediately and returns the result */

\[
\text{pass} : V \rightarrow S \rightarrow D \\
\text{pass} = \lambda v \in V \cdot \lambda s \in S \\
\quad \{(v, s), [], [], \{0, 0, \{\sqrt{\}}\}\}
\]

/* The function “process_set” takes a process to a set of deterministic processes, thus with only one acceptance state, such that the application of \(\|\) on this set yields the original process */

\[
\text{process_set} : D_{r,\sigma,\kappa} \rightarrow \mathcal{P}_f(D_{r,\sigma,\kappa}) \\
\text{process_set}(\bot) = \emptyset \\
\text{process_set}(R, I, O, A) = \{ \\
\quad (R, \\
\quad \{e \mapsto I(e) \mid e \in \text{dom}(I) \cap \pi_1(x)\}, \\
\quad \{e \mapsto O(e) \mid e \in \text{dom}(O) \cap \pi_2(x)\}, \\
\quad \text{acceptance}(x) \\
\quad ) \mid x \in A \}
\]

\[
\text{acceptance} : \mathcal{P}_f(\bigcup_{x \in \text{dom}(I)} V_{\text{chan}_x(r^*')}) \times \\
\mathcal{P}_f(\bigcup_{x \in \text{dom}(O)} V_{\text{chan}_x(r^*')}) \times \\
\mathcal{P}_f(\{\sqrt{\}\}) \\
\rightarrow \text{Acceptance_set}_{r,\sigma,\kappa}
\]

\[
\text{acceptance}(x) = \\
\text{case } x \text{ of} \\
\quad (\ldots, \{\sqrt{\}\}) \rightarrow \{x\} \cup \{(0, 0, \{\sqrt{\}\}\}\} \\
\quad \{x\}
\]

/* The function “\(\Rightarrow\)” can be seen as a piping operator. It evaluates its first argument and passes the value and the store that are the result of the evaluation to the second argument. It is like a sequencing operator. */

\[
\Rightarrow : (S \rightarrow D_{r,\sigma,\kappa}) \times \\
(V_r \rightarrow (S \rightarrow D_{r',\sigma',\kappa'})) \\
(S \rightarrow D_{r',\sigma' \cup \sigma,\kappa' \cup \kappa'})
\]

\[
\Rightarrow : D_{r,\sigma,\kappa} \times D_{r',\sigma',\kappa'} \rightarrow D_{r,\sigma' \cup \sigma,\kappa' \cup \kappa'}
\]

\[
P \parallel Q = \text{case } P, Q \text{ of}
\]
\( (R_1, I_1, O_1, A_1), \)
\( (R_2, I_2, O_2, A_2) \rightarrow \)
\( (R_i \cup R_2, \)
\( I_1 \parallel I_2, \)
\( O_1 \parallel O_2, \)
\( A_1 \parallel A_2 \)
\)

else \( \bot \)
end

\[ A \parallel B = \begin{cases} [e \mapsto I_1(e) | e \in \text{dom}(I_1)] & \text{if } A = \text{Input}\_\text{map}_{\tau, \sigma, a} \\ [e \mapsto I_2(e) | e \in \text{dom}(I_2)] & \text{if } A = \text{Input}\_\text{map}_{\tau, \sigma, a'} \end{cases} \]

\[ B \parallel C = \begin{cases} [e \mapsto O_1(e) | e \in \text{dom}(O_1)] & \text{if } B = \text{Output}\_\text{map}_{\tau, \sigma, a} \\ [e \mapsto O_2(e) | e \in \text{dom}(O_2)] & \text{if } B = \text{Output}\_\text{map}_{\tau, \sigma, a'} \end{cases} \]

\[ A = \text{Acceptance}\_\text{set}_{\tau, \sigma, a} \]

\[ B = \text{Acceptance}\_\text{set}_{\tau, \sigma, a'} \]

\[ A \parallel B = \{ x | x \in \mathcal{P}_f(\mathcal{P}_f(\bigcup_{c \in \text{in-chan}(a)} V_c) \times \mathcal{P}_f(\bigcup_{c \in \text{out-chan}(a)} V_c) \times \mathcal{P}_f(\{\top\}) \} \]
\[ \pi_1(x) \subseteq \{ \pi_1(y) | y \in A \cup A' \} \land \]
\[ \pi_2(x) \subseteq \{ \pi_2(y) | y \in A \cup A' \} \land \]
\[ (\exists y \in A \cup A' \bullet \]
\[ \pi_1(x) \supseteq \pi_1(y) \land \]
\[ \pi_2(x) \supseteq \pi_2(y) \]
\]

---

**Appendix B: Some Semantic Rules**

/* In what follows the denotational semantics of some syntactical constructions allowed in the language.*/

### Constants

\[ \mathcal{E}() : \text{Unit}, 0, 0 \]

\[ \mathcal{E}[{}: \mathcal{E}\_\text{true} : \text{Bool}, 0, 0 \]

\[ \mathcal{E}[{}: \mathcal{E}[\text{true}] = \begin{cases} \lambda \delta \in \mathcal{G}_S.(f u(\epsilon)) & \text{if } \lambda \delta \in \mathcal{G}_S.(0) \\ \lambda \Gamma \in \text{DynEnv}(\theta \epsilon) & \text{if } \lambda \Gamma \in \text{DynEnv}(\theta \epsilon) \\ \lambda s \in S \bullet \text{pass}(())(s) \end{cases} \]

\[ \mathcal{E}[{}: \mathcal{E}\_\text{skip} : \text{Unit}, 0, 0 \]

\[ \mathcal{E}[{}: \mathcal{E}[\text{skip}] = \begin{cases} \lambda \delta \in \mathcal{G}_S.(f u(\epsilon)) & \text{if } \lambda \delta \in \mathcal{G}_S.(0) \\ \lambda \Gamma \in \text{DynEnv}(\theta \epsilon) & \text{if } \lambda \Gamma \in \text{DynEnv}(\theta \epsilon) \\ \lambda s \in S \bullet \text{pass}(())(s) \end{cases} \]
**Abstraction**

\[ \forall \xi \vdash [x \mapsto \tau]^+ E : \tau', \sigma, \kappa \]

\[ \forall \xi \vdash \lambda x \cdot E : \tau \xrightarrow{\sigma, \kappa} \tau', \emptyset, \emptyset \]

\[ [ \lambda x \cdot E ] = \lambda \theta \in GS_{\xi}(f\upsilon(\xi)) \]
\[ \lambda \theta' \in GS_{\xi}(f\upsilon(\theta \sigma) \cup f\upsilon(\theta \kappa) \cup f\upsilon(\theta \tau)) \]
\[ \lambda \Gamma \in DynEnv_{\xi}(\theta \xi) \]

let

\[ v = \lambda \upsilon' \in V_{\psi \theta} \cdot [ E ](\gamma)(\Gamma \uplus [x \mapsto \upsilon']) \]

in

\[ \lambda s \in S \cdot pass(\upsilon)(s) \]

end

\[ \forall \xi \vdash E_1; E_2 : \tau_1, \sigma_1, \kappa_1 \]
\[ \forall \xi \vdash E_2 : \tau_2, \sigma_2, \kappa_2 \]

\[ [ E_1; E_2 ] = \lambda \theta \in GS_{\xi}(f\upsilon(\xi)) \]
\[ \lambda \theta' \in GS_{\xi}(f\upsilon(\theta(\sigma_1 \cup \sigma_2)) \cup f\upsilon(\theta(\kappa_1 \cup \kappa_2)) \cup f\upsilon(\theta(\tau_1)) \cup f\upsilon(\theta(\tau_2)) \]
\[ \lambda \Gamma \in DynEnv_{\xi}(\theta \xi) \]

\[ [ E_1 ](\theta)(\theta')(\Gamma) \Rightarrow \lambda \upsilon \in V_{\psi \theta \tau_1} \cdot [ E_2 ](\theta)(\theta')(\Gamma) \]

**Output**

\[ \forall \xi \vdash E_1 : \text{chan}_\chi(\tau) \sigma_1, \kappa_1 \]
\[ \forall \xi \vdash E_2 : \tau, \sigma_2, \kappa_2 \]

\[ \forall \xi \vdash E_1!E_2 : \text{Unit}, \sigma_1 \cup \sigma_2, \kappa_1 \cup \kappa_2 \cup \text{out}(\chi) \]

\[ [ E_1!E_2 ] = \lambda \theta \in GS_{\xi}(f\upsilon(\sigma_1 \cup \sigma_2)) \cup f\upsilon(\theta(\kappa_1 \cup \kappa_2)) \cup f\upsilon(\theta(\text{chan}_\chi(\tau))) \]
\[ \lambda \Gamma \in DynEnv_{\xi}(\theta \xi) \]

\[ [ E_2 ](\theta)(\theta')(\Gamma) \Rightarrow \lambda \upsilon \in V_{\psi \theta \tau_1} \cdot [ E_1 ](\theta)(\theta')(\Gamma) \Rightarrow \lambda \upsilon \in V_{\psi \theta \tau_1} \cdot [ E_1 ](\theta)(\theta')(\Gamma) \Rightarrow \lambda \upsilon \in V_{\psi \theta \tau_1} \cdot \upsilon (f \upsilon) \]

**Sequencing**
\[ \text{Lookup} \]

\[ \varepsilon \vdash x : \tau, \emptyset, \emptyset \]

\[
\begin{align*}
\text{[ } x \text{ ] } &= \\
&= \lambda \theta \in \mathcal{GS}_\tau(f_{\nu}(\varepsilon)) \cdot \\
&\quad \text{let } \forall v_1, \ldots, v_n \cdot \tau' = \theta \varepsilon(x) \\
&\quad \text{in } \\
&\quad \lambda \theta' \in \{ \theta_2 \mid \theta_2 \in \mathcal{GS}_\tau(f_{\nu}(\theta \tau)) \land \exists \theta_1 \cdot \theta_2 \theta \tau = \theta_1 \tau' \} \cdot \\
&\quad \text{let } \theta'' \text{ such that } \theta' \theta \tau = \theta'' \tau' \\
&\quad \text{in } \lambda \Gamma \in \text{DynEnv}_\tau(\theta \varepsilon) \cdot \\
&\quad \lambda s \in S \cdot \text{pass}(\Gamma(x)(\theta''))(s) \\
&\quad \text{end}
\end{align*}
\]

\[ \text{Internal choice} \]

\[ \varepsilon \vdash E_1 : \tau, \sigma_1, \kappa_1 \]
\[ \varepsilon \vdash E_2 : \tau, \sigma_2, \kappa_2 \]

\[ \varepsilon \vdash E_1 \mid E_2 : \tau, \sigma_1 \cup \sigma_2, \kappa_1 \cup \kappa_2 \]

\[
\begin{align*}
\text{[ } E_1 \mid E_2 \text{ ] } &= \\
&= \lambda \theta \in \mathcal{GS}_\tau(f_{\nu}(\varepsilon)) \cdot \\
&\quad \lambda \theta' \in \mathcal{GS}_{\tau_1 \cup \tau_2}(f_{\nu}(\theta_1 \sigma_1 \cup \sigma_2)) \cup \\
&\quad f_{\nu}(\theta_2 (\kappa_1 \cup \kappa_2)) \cup \\
&\quad f_{\nu}(\theta \tau) \cdot \\
&\quad \lambda \Gamma \in \text{DynEnv}_\tau(\theta \varepsilon) \cdot \\
&\quad \lambda s \in S \cdot \\
&\quad \left( \left[ E_1 \right](\theta)(\Gamma') \Rightarrow \lambda v \in V_{\theta \sigma} \cdot \lambda s' \in S \cdot \text{pass}(v)(s') \right) \text{S} \\
&\quad \left[ E_2 \right](\theta)(\Gamma) \Rightarrow \lambda v \in V_{\theta \sigma} \cdot \lambda s' \in S \cdot \text{pass}(v)(s') \text{S} \\
&\text{end}
\end{align*}
\]
An Operational Semantics of Value Passing*

Rance Cleaveland
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
USA
tel: +1 919 515 7862
fax: +1 919 515 7896
e-mail: rance@csc.ncsu.edu

July 16, 1993

Abstract

This paper develops an operational semantics for concurrent languages with value passing. An operation analogous to substitution in the λ-calculus is given, and an operational equivalence is defined and shown to coincide with Milner's bisimulation equivalence. In contrast with existing process-algebraic approaches to value passing, this semantics does not reduce data exchange to pure synchronization over (potentially infinite) families of ports indexed by data.

1 Introduction

Process algebras such as CCS [13], CSP [10] and ACP [2] have been proposed as theories that highlight fundamental notions in concurrency in the same way that the λ-calculus [3] does for functional programming. Although the latter essentially focuses on only one programming construct, function application, it has proven very useful as a framework for studying numerous issues, both implementational as well as theoretical, in functional programming languages. In the search for similarly foundational operations in concurrency, process algebra researchers have typically focused on modeling synchronization between, and the simultaneous execution of, processes. Appropriate equivalences and preorders [4, 5, 6, 7, 11, 12, 16] have been defined that relate concurrent systems on the basis of their externally visible

*Research supported by NSF/DARPA grant CCR-9014775, NSF grant CCR-9120995, ONR Young Investigator Award N00014-92-J-1582, and NSF Young Investigator Award CCR-9257963.
behavior, and the theories have been successfully applied to the verification of concurrent systems. However, researchers in process algebra have devoted relatively little attention to the operational aspects of the exchange of data between processes. This attention to "flow of values" is central in the $\lambda$-calculus, where notions of $\beta$-reduction and substitution form the basis for understanding function application. When value passing has been modeled in process algebra, it has typically been done via a reduction to simple synchronization, with data values in essence encoded in the "name" of the ports that synchronizing processes use. While adequate at an abstract level, this approach obscures details associated with ensuring that the correct transmission of data from one process to another. Consequently, it is difficult to use these semantics to guide the development of implementations—specifically, interpreters—of value-passing languages.

The goal of this paper is to formulate an implementable semantics for process algebras with value passing. To this end, we present a semantics that accounts for operational concerns associated with the exchange of data between processes while maintaining a degree of abstraction from specific implementational details. The central difficulty to overcome is that, in contrast to the situation in the $\lambda$-calculus, where the syntactic structure of a reducible expression (redex) determines the subexpressions involved in a step of $\beta$-reduction, the emitters and consumers of values in concurrent programs may be arbitrarily separated, syntactically. Consequently, exchanging a value requires, at some level, the use of some kind of routing information to determine where the datum is to be delivered.

The remainder of the paper is structured as follows. The next section describes more precisely the intuitive motivation for the work presented in this paper. The section following then outlines a version of Milner's CCS with value-passing, defines its operational semantics according to the standard approach, and defines an associated behavioral equivalence, bisimulation equivalence. Section 4 then remarks on a shortcoming of the traditional semantics, gives a semantics for value-passing designed to remedy this difficulty, and presents a new notion of semantic equivalence. The section following establishes that the semantic equivalences coincide, and the final section presents our conclusions and directions for future research.

2 Reduction and the Flow of Values

The aim of this paper is to give a semantics for concurrent processes that pass values; the semantics should be implementable, and therefore account for issues associated with the transmission of data, while remaining abstract with respect to more concrete implementational details. In this section we outline aspects peculiar to concurrent languages that complicate the development of such a semantics. We begin by first reviewing relevant intuitions from the $\lambda$-calculus, some familiarity with which will be assumed on the part of the reader.

The $\lambda$-calculus is built on the notion of $\beta$-reduction, which in turn is based on the operation of substitution. $\beta$-reduction can be thought of as specifying how an argument to
a function is to be distributed inside the body of the function; in the reduction step

$$(\lambda x. M)N \rightarrow M[N/x],$$

the $\rightarrow$ relation in essence states that the expression $N$ should be “sent” to all (free) occurrences of $x$ in $M$. It should be noted that this “flow” of data is local to the redex $(\lambda x. M)N$ in the sense that places in the term where datum $N$ is to be “sent” can only be within $M$; any context that might surround the term cannot alter the possible destinations of $N$. This affords a syntactic characterization of redexes—a term can be a redex if and only if it is of the form $M \, N$ (i.e. $M$ and $N$ are syntactically adjacent) and $M$ has the form $\lambda x. M’$. Different operational accounts of the $\lambda$-calculus exploit this definition of redex to define evaluation relations that have direct implementations as interpreters.

In concurrent languages, by way of contrast, the locality of data flow present in the $\lambda$-calculus is, by design, absent, since these languages are meant to model systems consisting of entities that exchange data without necessarily being adjacent to one another. For example, consider the following term in a putative process algebra.

$$\cdots || \alpha X.P || \cdots || \overline{\alpha} E.Q || \cdots$$

This system consists of some number of subsystems running in parallel. Process $\overline{\alpha} E.Q$ wishes to output expression $E$ on port $\alpha$ and then behave like $Q$, while $\alpha X.P$ is awaiting an input on port $\alpha$ which it will then “assign” to $X$ throughout $P$. What is notable here is the nonlocal flow of information; $\overline{\alpha} E.Q$ and $\alpha X.P$ in some sense form a “redex” even though they are not syntactically adjacent and may in fact be separated arbitrarily far apart in the term. Additionally, other processes (subterms) may also wish to communicate using $\alpha$, and these may interfere with the exchange between the two processes above. Both of these features complicate the development of an “implementable” semantics for languages supporting concurrency and value passing; the work reported in this paper aims at circumventing these difficulties.

3 CCS and Value Passing

This section presents the syntax and traditional semantics for a language with value passing. In the interests of notational simplicity, values and processes will be distinct (in contrast with the language found in [14]), and a common mechanism, recursive definition, for building processes will be omitted. Including these is not problematic, but we have chosen to exclude them in the interests of notational simplicity.

The language—CCS with Value Passing (CCS-VP)—is parameterized with respect to several sets of syntactic entities. Let $V$ be a (countably infinite) set of variables ranged over by $V, V', V_1, \ldots$ and $E$ be a set of value expressions ranged over by $E, E', E_1, \ldots$; these expressions may contain occurrences of variables $V_i$. Also let $\Lambda$ be a (countably infinite) set of ports, or labels, ranged over by $\alpha, \beta, \ldots$. We use $L, L', L_1, \ldots$ to represent finite subsets
of \Lambda, and \( f \) to denote a (relabeling) function in \( \Lambda \rightarrow \Lambda \). The syntax of CCS-VP may now be given using the following grammar.

\[
P := \text{nil} \mid \alpha V. P \mid \alpha E. P \mid P + P \mid P|P \mid PL \mid P[f]
\]

In the sequel we use \( \mathcal{P} \) to denote the set of all terms in CCS-VP, with \( P, Q, P', Q', P_1, Q_1, \ldots \) ranging over \( \mathcal{P} \). We sometimes refer to elements of \( \mathcal{P} \) as processes.

In order to give the standard operational semantics of CCS-VP we first define the following notions. We call expressions of the form \( \alpha E \) and \( \alpha E \), where \( \alpha \in \Lambda \) and \( E \in \mathcal{E} \), actions. We also let \( \tau \notin \Lambda \) denote a special internal action and use \( \mathcal{A} = \{ \alpha E \mid \alpha \in \Lambda \land E \in \mathcal{E} \} \cup \{ \alpha E \mid \alpha \in \Lambda \land E \in \mathcal{E} \} \cup \{ \tau \} \) to represent the set of all actions. \( a, a', a_1, \ldots \) are used to range over \( \mathcal{A} \). Intuitively, a process can perform \( \alpha E \) when it is capable of receiving \( E \) as an input on \( \alpha \) from its environment, while it can perform \( \alpha E \) when it may output \( E \) on \( \alpha \) to its environment. We extend \( \alpha \) to actions as follows: if \( a = \alpha E \) then \( \alpha = \alpha E \), and if \( a = \alpha E \) then \( \alpha = \alpha E \). The function \( \pi \in \mathcal{A} \rightarrow \Lambda \cup \{ \tau \} \) returns the “port component” of an action; so \( \pi(\alpha E) = \pi(\alpha E) = \alpha \), and \( \pi(\tau) = \tau \). Also, if \( f \in \Lambda \rightarrow \Lambda \) and \( \alpha \in \mathcal{A} \), then \( \text{apply}(f, a) \) “relabels” the port component of \( a \) by \( f \); formally, this is given by

\[
\text{apply}(f, a) = \begin{cases} 
  f(\alpha)E & \text{if } a = \alpha E \\
  \overline{f}(\alpha)E & \text{if } a = \overline{\alpha}E \\
  \tau & \text{otherwise (i.e. } a = \tau) 
\end{cases}
\]

We also assume that \( \mathcal{E} \) is equipped with notions of free variable and substitution of expressions for variables; we use \( \text{fv}(E) \subseteq \mathcal{V} \) to represent the free variables in \( E \) and \( E_1[V := E_2] \in \mathcal{E} \) to denote the expression \( E_1 \) with all occurrences of \( V \) replaced by \( E_2 \). We further assume that for any \( E \in \mathcal{E}, |\text{fv}(E)| < \infty \) and that there exists a function \( \eta \in 2^\mathcal{V} \rightarrow \mathcal{V} \) with the property that for any finite \( \mathcal{V}' \subseteq \mathcal{V}, \eta(\mathcal{V}') \notin \mathcal{V}' \). We now define substitution for CCS-VP as follows.

**Definition 3.1** Let \( V \in \mathcal{V} \) and \( E \in \mathcal{E} \). Then \( P[E/V] \) for \( P \in \mathcal{P} \) is defined inductively as follows.

- \( \text{nil}[E/V] = \text{nil} \).

- \((\alpha V'.P)[E/V] = \begin{cases} 
  \alpha V'.P & \text{if } V = V' \\
  \alpha V'.(P[E/V]) & \text{if } V \neq V' \text{ and } V \notin \text{fv}(E) \\
  \alpha V''.(P[V''/V'][E/V]) & \text{otherwise}
\end{cases}\)

where \( V'' = \eta(\text{fv}(E) \cup \{V\}) \). Notice that the final clause handles the problem of variable capture in the usual fashion by renaming bound variables as necessary.

- \((\alpha E'.P)[E/V] = \alpha(E'[V := E]).(P[E/V])\).

- \((P_1 + P_2)[E/V] = (P_1[E/V]) + (P_2[E/V])\).
\[
\alpha V.P \xrightarrow{\alpha E} P[E/V] \quad \text{for any } E \in \mathcal{E}
\]
\[
\bar{\alpha}E.P \xrightarrow{\bar{\alpha}E} P
\]
\[
P \xrightarrow{a} P' \quad \Rightarrow \quad P + Q \xrightarrow{a} P'
\]
\[
Q \xrightarrow{a} Q' \quad \Rightarrow \quad P + Q \xrightarrow{a} Q'
\]
\[
P \xrightarrow{a} P' \quad \Rightarrow \quad P|Q \xrightarrow{a} P'|Q
\]
\[
Q \xrightarrow{a} Q' \quad \Rightarrow \quad P|Q \xrightarrow{a} P'|Q'
\]
\[
P \xrightarrow{a} P', Q \xrightarrow{a} Q' \quad \Rightarrow \quad P|Q \xrightarrow{a} P'|Q'
\]
\[
P \xrightarrow{a} P', \pi(a) \not\in L \quad \Rightarrow \quad P\backslash L \xrightarrow{a} P'\backslash L
\]
\[
P \xrightarrow{a} P' \quad \Rightarrow \quad P[f] \xrightarrow{\text{apply}(f,a)} P'[f]
\]

Figure 1: The Traditional Semantics of CCS-VP.

- \((P_1|P_2)[E/V] = (P_1[E/V])|(P_2[E/V]).\)
- \(P\backslash L[E/V] = (P[E/V])\backslash L.\)
- \(P[f][E/V] = (P[E/V])[f].\)

The traditional semantics of CCS-VP is given as a relation \(\to \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P},\) where, \(P \xrightarrow{a} P'\) holds if \(P\) is capable of executing action \(a\) and evolving to \(P'\) as a result. The formal definition of \(\to\) appears in Figure 1. Intuitively, \(\alpha V.P\) is a process waiting for an input on channel \(\alpha;\) upon receiving an expression it evolves to \(P\) with occurrences of \(V\) replaced by this value. \(\bar{\alpha}E.P\) first outputs \(E\) on \(\alpha\) and then behaves like \(P\). Operator + is a choice operator; \(P_1 + P_2\) is initially capable of the actions of either \(P_1\) or \(P_2\), with the process thereafter behaving like the process whose initial action was chosen. \(P_1|P_2\) models the parallel composition of \(P_1\) and \(P_2\) by interleaving the actions available to each; in addition, if one of the processes is capable of outputting a value on a port that the other is capable of receiving on the same port, then the two processes synchronize, with an internal action resulting. In \(P\backslash L\) the ports in \(L\) become local; no interaction can take place between \(P\) and its environment using these channels. Finally, in \(P[f]\) \(f\) is used to rename the labels that \(P\) interacts on.

Having defined the operational semantics of CCS-VP processes, we now present the definition of bisimulation equivalence [13], which is designed to related processes on the basis of the behavior they exhibit to their environments. The definition is given in terms of bisimulations, which are relations that require related processes to have "matching transitions".

**Definition 3.2** A relation \(\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}\) is a bisimulation if, whenever \(P \mathcal{R} Q\), then the following holds for all \(a \in \mathcal{A}\).

1. If \(P \xrightarrow{a} P'\) then there is a \(Q'\) such that \(Q \xrightarrow{a} Q'\) and \(P' \mathcal{R} Q'\).
2. If \(Q \xrightarrow{a} Q'\) then there is a \(P'\) such that \(P \xrightarrow{a} P'\) and \(P' \mathcal{R} Q'\).

Bisimulation equivalence, \(\sim \subseteq \mathcal{P} \times \mathcal{P}\), is defined by: \(P \sim Q\) if and only if there is a bisimulation \(\mathcal{R}\) such that \(P \mathcal{R} Q\).
4 An Effective Semantics for Value Passing

While elegant, the semantics of CCS-VP is not directly implementable, in a certain sense. To see this, consider how one might try to implement an interpreter for CCS-VP on the basis of the rules given in Figure 1. Given a process term $P$, such an evaluator would need first to compute the set

$$\{ \langle a, P' \rangle \mid P \xrightarrow{a} P' \}.$$ 

However, this set will in general be infinite when $P$ is capable of any input action and the set of value expressions is infinite, and thus the interpreter would not, in general, terminate.

The problem with the semantics is that it is in a certain sense too abstract with respect to modeling the exchange of values. In essence, the semantics works by replacing a given input port name by a number of port names indexed by the possible values that may be received on it. For example, suppose that $E = \text{Nat}$, the set of natural numbers, and consider a process of the form $P \equiv aV.P'$. From the operational semantics given in the previous section it is the case that $P \xrightarrow{\alpha} P'[i/V]$ for any natural number $i$. The same behavior could be deduced for the following process in a non-value-passing calculus: $\Sigma_{i \in \text{Nat}} \alpha_i.P'[i/V]$. This has the effect of obscuring how the transfer of values between $P$ and $Q \equiv \alpha.E.Q'$ should be achieved in the system $P|Q$.

In this section, we develop an alternative operational semantics for CCS-VP that is computable; it also aims to expose what we feel is a main operational consideration in the implementation of communicating concurrent systems: the routing of data from senders to receivers. The new semantics relies on the introduction of a new syntactic category of input-enabled terms and an operation analogous to substitution that "channels" an input into one of these terms. Using the new semantics, we define a notion of behavioral equivalence that, as the next section shows, corresponds to bisimulation equivalence.

The set $I$ of input-enabled terms is given by the following grammar.

$$I ::= V.P \mid I\downarrow P \mid P[I] \mid I\backslash L \mid I[f]$$

Here $P \in \mathcal{P}$ is as defined in Section 3. As was the case there, $L \subseteq \Lambda$ and $f \in \Lambda \rightarrow \Lambda$. We use $I, I', I_1, \ldots$ to range over $I$ and $K, K', K_1, \ldots$ to range over $\mathcal{P} \cup I$.

Intuitively, an input-enabled term results when a process has indicated that it is ready to receive an input on a particular channel and is awaiting the value that it will substitute for the appropriate variable occurrences. In $V.P$, for example, all occurrences $V$ in the process $P$ are to be replaced by the value that eventually arrives. $I\downarrow P$ results when process $Q$ in $Q|P$ signals a desire for an input; the $\downarrow$ indicates that when the value arrives, it should be routed into the left component. Operator $[\ ]$ plays the symmetric role to $\downarrow$. Finally, $\backslash L$ and $[f]$ are as in Section 3.

In order formally to specify the new semantics, we first introduce the new set of actions we consider. The set $\mathcal{A}_m = \Lambda$ and represents the set of input actions. $\mathcal{A}_o = \{ \alpha E \mid \alpha \in \Lambda \land E \in E \} \cup \{ \tau \}$ contains the "noninput actions"; then $\mathcal{A}' = \mathcal{A}_m \cup \mathcal{A}_o$. We use $a, a', a_1, \ldots$ to range over $\mathcal{A}'$. The functions $\pi' \in \mathcal{A}' \rightarrow \Lambda \cup \{ \tau \}$ and $\text{apply}' \in (\Lambda \rightarrow \Lambda) \times \mathcal{A}' \rightarrow \mathcal{A}'$ are now
defined by:

\[
\pi'(a) = \begin{cases} 
\alpha & \text{if } a = \alpha \\
\alpha & \text{if } a = \bar{\alpha}E \\
\tau & \text{otherwise}
\end{cases}
\]

\[
\text{apply}'(f, a) = \begin{cases} 
f(\alpha) & \text{if } a = \alpha \\
f(\bar{\alpha}E) & \text{if } a = \bar{\alpha}E \\
\tau & \text{otherwise (i.e. } a = \tau)\end{cases}
\]

We now define the routing operation that “delivers” a value to the appropriate places in an input-enabled term.

**Definition 4.1** Let \( I \in \mathcal{I} \) and \( E \in \mathcal{E} \). Then \( I \cdot E \) is defined inductively as follows.

- \((V.P) \cdot E = P[E/V] \).
- \((I/P) \cdot E = (I \cdot E)[P] \).
- \((P[I] \cdot E = P[I \cdot E] \).
- \((I \backslash L) \cdot E = (I \cdot E) \backslash L \).
- \((I[f]) \cdot E = (I \cdot E)[f] \).

It is straightforward to establish the following.

**Lemma 4.2** Let \( I \in \mathcal{I} \) and \( E \in \mathcal{E} \). Then \((I \cdot E) \in \mathcal{P} \).

The operational semantics is given in terms of a relation \( \rightarrow \subseteq (\mathcal{P} \times \mathcal{A}' \times (\mathcal{P} \cup \mathcal{I})) \). The formal definition appears in Figure 2; the rough intuition underlying the rules is the following. The operations \(+, \backslash, \text{ and } [f] \) behave essentially as they do the old semantics, as does \( \bar{\alpha}E.P \) (i.e. prefixing by an output action). A process of the form \( \alpha V.P \) signals its desire for an input on \( \alpha \) by executing the action \( \alpha \) and then evolving to the input-enabled term \( V.P \), where it awaits a value to be routed to it from the environment. The rules for \( | \) are slightly more complicated than before, since terms must be decorated with appropriate routing information. Thus, if \( P \) requests an input on \( \alpha \) and evolves to input-enabled term \( I \), then \( P|Q \) must evolve to the input-enabled term \( I|Q \) to indicate that its left subterm is awaiting a value. If \( P \) wishes to perform an output without synchronizing with \( P \), then \( P|Q \) behaves as in the old semantics. Finally, if \( P \) signals a desire for an input on \( \alpha \) and \( Q \) is willing to perform an output on the same port, then the two may synchronize, with a \( \tau \) resulting and the appropriate value expression being routed to \( P \). The dual situation holds for \( Q \).

As an illustration of semantics, consider the following term.

\[
P \equiv (\alpha X.\bar{\beta}X.nil|\beta Y.nil)|\bar{\alpha}E.nil
\]
\[ \alpha V.P \vdash_{\alpha} V.P \]

\[ \alpha E.P \vdash_{\alpha E} P \]

\[ P \vdash_{\alpha} K \quad \Rightarrow \quad P + Q \vdash_{\alpha} K \]

\[ Q \vdash_{\alpha} K \quad \Rightarrow \quad P + Q \vdash_{\alpha} K \]

\[ P \vdash_{\alpha} I \quad \Rightarrow \quad P|Q \vdash_{\alpha} I|Q \]

\[ P \vdash_{\alpha E} P' \quad \Rightarrow \quad P|Q \vdash_{\alpha E} P'|Q \]

\[ Q \vdash_{\alpha} I \quad \Rightarrow \quad P|Q \vdash_{\alpha} P[I \]

\[ Q \vdash_{\alpha E} Q' \quad \Rightarrow \quad P|Q \vdash_{\alpha E} P|Q' \]

\[ P \vdash_{\alpha} I, Q \vdash_{\alpha E} Q' \quad \Rightarrow \quad P|Q \vdash_{\alpha} (I \triangleleft E)|Q' \]

\[ P \vdash_{\alpha E} P', Q \vdash_{\alpha} I \quad \Rightarrow \quad P|Q \vdash_{\alpha} P'|(I \triangleleft E) \]

\[ P \vdash_{\alpha} K, \pi'(a) \notin L \quad \Rightarrow \quad P\repl{a}{L} \vdash_{\alpha} K\repl{a}{L} \]

\[ P \vdash_{\alpha} K \quad \Rightarrow \quad P[f] \vdash_{\alpha,\text{apply}(f, a)} K[f] \]

---

Figure 2: A New Semantics for Value Passing.

By applying the above rules, we may deduce the following transitions for \( P \).

\[ P \vdash_{\alpha} (X.|\beta X.nil|\beta Y.nil)|\alpha E.nil \]

\[ P \vdash_{\beta} (\alpha X.|\beta X.nil|Y.nil)|\alpha E.nil \]

\[ P \vdash_{\alpha E} (\alpha X.|\beta X.nil|\beta Y.nil)|nil \]

\[ P \vdash_{\alpha \triangleleft} (X.|\beta X.nil|\beta Y.nil) \triangleleft E|nil \]

\[ \equiv ((X.|\beta X.nil) \triangleleft E|\beta Y.nil)nil \]

\[ \equiv (\beta E.nil|\beta Y.nil)|nil \]

The last of these transitions illustrates how the \( \triangleleft \) and \( \repl{}{} \) operators to guide incoming values to the appropriate subterm, where they are then substituted for the indicated variable. Notice also that in the transfer of data in this transition, the routing procedure does not “touch” the process \( \beta Y.nil \); no attempt is made to substitute \( E \) into it.

4.1 Implementing the Semantics

That the semantics implementable stems from the following result.

**Theorem 4.3** Let \( P \in \mathcal{P} \), and let \( T = \{ \langle a, K \rangle \mid P \vdash_{\alpha} K \} \). Then \( |T| < \infty \).

On the basis of this theorem, it is relatively straightforward to implement a routine that computes the transitions available to a term. Given a term an input, the procedure would recursively compute the transitions of its subterms and combine them appropriately, as indicated by the applicable rules, to obtain the transitions of the over-all term. As the theorem guarantees that these sets are finite, the routine is effectively computable.
4.2 A Semantic Equivalence

As transitions are not defined for elements of $\mathcal{I}$, we cannot directly apply the definition of $\sim$ given in Section 3. Instead, we define the following.

**Definition 4.4** A relation $\mathcal{R} \subseteq (\mathcal{P} \times \mathcal{P}) \cup (\mathcal{I} \times \mathcal{I})$ is an input-sensitive bisimulation if, whenever $K_1 \mathcal{R} K_2$ then the following hold.

1. If $(K_1, K_2) \in \mathcal{P} \times \mathcal{P}$ then for all $a \in \mathcal{A}'$:
   
   (a) $K_1 \xrightarrow{a} K'_1$ implies there is a $K'_2$ such that $K_2 \xrightarrow{a} K'_2$ and $K'_1 \mathcal{R} K'_2$.

   (b) $K_2 \xrightarrow{a} K'_2$ implies there is a $K'_1$ such that $K_1 \xrightarrow{a} K'_1$ and $K'_1 \mathcal{R} K'_2$.

2. If $(K_1, K_2) \in \mathcal{I} \times \mathcal{I}$ then for all $E \in \mathcal{E}, (K_1 \triangleleft E) \mathcal{R} (K_2 \triangleleft E)$.

$K_1 \simeq K_2$ if there is an input-sensitive bisimulation $\mathcal{R}$ such that $K_1 \mathcal{R} K_2$.

It is straightforward to establish that $\simeq$ is itself an input-sensitive bisimulation, and is in fact the largest such relation.

5 The Equivalences Are Equivalent

This section establishes that, when restricted to $\mathcal{P} \times \mathcal{P}$, $\simeq$ relates exactly the same processes as $\sim$. The importance of this result lies in the fact that it in some since licenses the use of either form of operational semantics for CCS-VP. If one is not interested in the specific mechanisms of data exchange, then one may use the traditional, more abstract semantics; if on the other hand one is interested in more implementational considerations, then one may use the new semantics. In either case, the equivalences deduced between processes at one level will hold at the other.

The correspondence between equivalences rests on the following lemmas.

**Lemma 5.1** Let $P \in \mathcal{P}, \alpha \in \Lambda,$ and $E \in \mathcal{E}$. Then $P \xrightarrow{\alpha E} P'$ if and only if there exists an $I \in \mathcal{I}$ such that $P \xrightarrow{\alpha} I$ and $P' \equiv I \triangleleft E$.

**Proof.** Proceeds by induction on the derivations of $P \xrightarrow{\alpha E} P'$ and $P \xrightarrow{\alpha} I$. □

**Lemma 5.2** Let $P \in \text{Proc}$ and $a \in \mathcal{A}_o$. Then $P \xrightarrow{a} P'$ if and only if $P \xrightarrow{a} P'$.

**Proof.** Proceeds by induction on the derivations of $P \xrightarrow{a} P'$ and $P \xrightarrow{a} P'$. □

Using these lemmas, we can now prove the following theorem.

**Theorem 5.3** Let $P_1, P_2 \in \mathcal{P}$. Then $P_1 \sim P_2$ if and only if $P_1 \simeq P_2$.

**Proof.** Using Lemmas 5.1 and 5.2, one can show that every input-respecting bisimulation is also a bisimulation and that every bisimulation may be extended into an input-respecting bisimulation. The proof then follows. □
6 Conclusions

In this paper we have presented an alternative semantics for a concurrent language with value passing. Inspired by the operational considerations that are evident in the reduction-style semantics given to the $\lambda$-calculus, this semantics attempts more faithfully to reflect abstractly the operations concerns associated with the exchange of values between processes distributed in space. A notion of operational equivalence is defined and shown to coincide with the well-known bisimulation equivalence of CCS. Thus the new semantics maintains consistency with the traditional semantics given to value-passing languages while providing more information about how data is passed between processes.

There are a number of directions in which this work can be extended. In this paper we only considered binary synchronous communication in the style of CCS. One could also alter the framework slightly to support synchronous multi-party communication; one way to do so would involve the introduction of an additional "routing" operator for input-enabled terms that would indicate that a data value should be routed to both subterms. It would also be interesting to examine the semantics in this paper in light of the criteria proposed in [15] for an operational semantics to be effective. This could lead to a formalization of the informal argument of implementability given here. Another direction for future research would involve extending the syntax of process terms to permit explicit routings and substitutions in the same way that the $\lambda$-calculus has been extended with explicit substitutions in, e.g., [1]. In the $\lambda$-calculus, explicit substitutions permit a finer study of the cost of reducing one term to another. In addition to enabling a more detailed study of "communication costs" à la the $\lambda$-calculus, explicit routings could also be used as a basis for giving a reduction-like semantics for asynchronous communication in which senders need not wait for receivers. To see how this might be done, consider the process

$$T \equiv (\alpha X.P|\beta Y.Q)|\bar{a}E.R.$$  

With terms of the form $I \triangleleft E$ allowed as processes, one could imagine defining a semantics in which $T$ could have the following transition.

$$T \xrightarrow{T} ((\alpha X.P|\beta Y.Q) \triangleleft E)|R$$

Thus, before the $\triangleleft E$ operation is fully resolved, $R$ could engage in another transition. A number of details would need to be worked out, but the approach appears promising.

Finally, it would also be interesting to extend the language in this paper by considering different types of values and permitting data variables to have types ascribed to them. One could alter the reduction semantics for value-passing by introducing a sensitivity to type: input actions could be labeled by the type of the variable they bind, and a reduction would take place only if the type of the output expression matches that of the variable. In this way one could obtain a (first-order) typed reduction system for concurrent processes. Several interesting directions for further inquiry would then present themselves. One would be to investigate the development of a higher-order typed language for processes. Another would be the extraction of "pure" (finite-state) process algebra terms from typed terms that would
retain some of the behavioral properties of the original (infinite-state) terms. For example, one could imagine taking a term of the form $\alpha V : \text{Int}.P$ and generating a term of the form $\alpha_{\text{Int}}.T(P)$, where $T(P)$ is the “type” for $P$. To the extent that such a procedure would yield a term reflecting at an abstract level the behavior and types of passed values of the original term, useful behavioral information could be automatically computed for concurrent programs.

Related Work

Traditionally, researchers in process algebra have defined the semantics of value-passing by “reducing” it to pure synchronization in the fashion described in Section 3 [11, 13, 14]. One approach to avoiding the attendant infinite number of transitions that this entails can be found in [8, 9]. There the semantics dictates that processes such as $\alpha X.P$ may engage in the single transition $\alpha X.P \xrightarrow{\alpha} P$; when processes synchronize, then, data transfer is handled via substitution in the usual sense. The approach is similar to ours, although it differs in that unnecessary substitutions can be performed, with an attendant need to rename bound variables on occasion. For example, consider the term $\alpha X.P|\beta X.Q$; note that $X$ is free in the right-hand processes. If this system is allowed to engage in the transition

$$\alpha X.P|\beta X.Q \xrightarrow{\alpha X} P|\beta X.Q$$

then $X$ is now free in both the left-hand and right-hand sides. However, since their semantics requires that a value arriving to this term as the result of a synchronization on $\alpha$ be substituted $X$ into both processes, the free occurrence of $X$ in the right-hand side process would incorrectly be replaced. To ensure this does not happen, the $X$ in the left-hand process must be renamed. In contrast, our semantics would not require the renaming of $X$ in here, because the routing operation would ensure that substitution is only performed for occurrences of $X$ on the left.

Acknowledgements. I would like to thank Bard Bloom and S. Purushothaman for their insightful comments on an earlier version of this paper.

References


An Information Flow Security Property for CCS

(Extended Abstract)

Riccardo Focardi          Roberto Gorrieri

Università di Bologna, Dipartimento di Matematica
Piazza di Porta San Donato 5, I – 40127 Bologna (Italy)*
e-mail:gorrieri@cs.unibo.it

Abstract

Multilevel security has been introduced to limit the activity of Trojan Horses – malicious programs which try to broadcast secret information. Every information is classified with a security level and the multilevel secure system must guarantee that information can not flow from a level to a lower one. So a Trojan Horse, which operates at a certain level, has no way to downgrade information and the effect of its execution is restricted into such a level.

In this paper, a formal property of security, called Non Deducibility on Compositions (NDC), is proposed and defined on CCS agents. We assume, for the sake of simplicity, to have two security levels only: high and low. In this context, a Trojan Horse is a high level process which tries to pass information to the low level. NDC is based on the following intuition: a system is NDC secure if, when connected to all possible high level processes, the computations of the low level users are not affected. So, such a property guarantees that no information flow from the high level to the low one can occur. An alternative formulation of NDC, which exploits only local information, is presented. It has the merits of being algorithmically testable for finite-state systems. Moreover, it is useful for modular verification of secure systems: indeed, NDC is a composable property w.r.t. the CCS operators of parallel composition and restriction. Moreover, through an example, it is shown how NDC can be applied to reason about realistic systems. Finally, we show that NDC, based on trace semantics, may be insufficient in some cases. Hence, NDC is formulated also assuming the finer (weak) bisimulation as basic underlying semantics.

1 Introduction

Security is a crucial property of system behaviour. It requires that there is a strict control over the information flow among parts of the system. The main problem is to limit, and possibly avoid, the damages produced by malicious programs, usually called Trojan Horses, which tries to broadcast

*This work has been partially supported by Esprit Basic Research project BROADCAST n. 6360 and by Italian CNR, grant n. 92.00069.CT12.115.25585
secret information. There are several approaches to security, most of which based on some access control policy.

In the Discretionary Access Control security (DAC for short), every subject (i.e., an active agent such as a user), decides the access properties of its objects (i.e., passive agents such as files). An example of DAC is the file management in Unix where a user can decide the access possibilities of her/his files. So, if a user executes a Trojan Horse program, this can modify the security properties of user's objects (usually, a program inherits executor's capabilities).

A solution to this problem is the Mandatory Access Control (MAC for short), where access rules are imposed by the system. An example of MAC is Multilevel Security [1]: every object is bound to a security level, and so every subject is; information can flow from a certain object to a certain subject only if the level of the subject is greater then the level of the object. So a Trojan Horse, which operates at a certain level, has no way to downgrade information and its action is restricted inside such a level. There are two access rules (see figure 1):

**No Read Up**: A subject cannot read data from an upper level object.

**No Write Down**: A subject cannot write data to a lower level object.

However these access rules do not guarantee an absence of information flow from upper levels to lower ones. It could be possible to indirectly transmit information using some system side effect. For example, if two levels – 'high' and 'low' – share some finite storage resource (e.g., a hard disk), it is possible to transmit data from level 'high' to level 'low' by using the 'resource full' error message. It is sufficient, for a high level transmitter, to alternatively fill or empty the resource in order to transmit a '1' or a '0' data. Simultaneously, the low level receiver tries to write on the resource, decoding every error message with a '1' and every succesfull write with a '0'.

It is clear that such indirect ways of transmission, called covert channels, do not violate the two multilevel access rules. Therefore, to guarantee a correct information flow control, it is necessary to integrate a MAC discipline with a covert channel analysis [9,16,10,14].

An alternative, more general approach to security requires to control directly the whole flow of information, rather than the accesses of subjects to objects. To make this, it is necessary to choose a formal model of system behaviour and to define the information flow on such a model. By imposing some information flow rule, we can control any kind of transmission, be it direct or indirect.

Our proposal is based on labelled transition systems [8] (LTS for short), a very general and simple model, used to give semantics to many concurrent languages. In particular, a variant of CCS [13] is used in this paper to specify the behaviour of concurrent systems.

We assume, for the sake of simplicity, to have two security levels\(^1\) only: high and low. Each action will be bound to a security level; hence, it will be sufficient to partition the CCS action set in two sets Act\(_L\) (the low level actions) and Act\(_H\) (the high level ones). Agents built with high (low)

---

\(^1\)The generalization of the approach to \(n\) levels can be easily done.
level actions only are often called high (low) level agents. In this context, an information flow from the high level to the low one takes place when the effects of high level actions are visible by low level users. So, a Trojan Horse is a high level agent which tries to modify the low level executions.

In the literature, there are already many different definitions of this kind, based on several system models [4,15,11,7,17,12]. In [3] we have unified them under the its model, then compared and classified, leading to the general notion presented in this abstract: Non Deducibility on Compositions (NDC, for short).

NDC is a transposition on its’s of Non Deducibility on Strategies (NDS), originally defined [17] over a very restrictive model. It assumes two security levels and is based on the following intuition: a system is NDC secure if, when connected to all possible high level agents (or Trojan Horses), the computations of the low level users are not affected. So, such a property guarantees that no information flow from the high level to the low one can take place. We also show that NDC is closed with respect to system composition. This property is very useful in verification and permits to write a new system as a composition of subsystems satisfying NDC. This is the subject of Section 3.

An example of application of the theory is reported in Section 4. It presents an acess monitor, which is proved to be NDC secure.

It is then shown – Section 5 – that a system which satisfies the NDC property may show up a non secure behaviour. This is due to the fact that NDC has been defined exploiting a rather weak semantics, trace semantics, which equates systems when they have the same set of execution traces. We extend NDC to the finer equivalence of weak bisimulation [13], hence obtaining a much finer security property which, in our opinion, seems to be a reasonable candidate to express formally security constraints over CCS-like languages.
2 Preliminary Definitions

We assume the reader familiar with the CCS language and the lts model, shortly presented in the following (see [13] for a comprehensive treatment).

Definition 2.1 A labelled transition system (lts) is a triple \((S, T, \rightarrow)\) such that:

- \(S\) is a set of states
- \(T\) is a set of labels (actions)
- \(\rightarrow \subseteq S \times T \times S\) is a set of labelled transitions

\((S_1, \alpha, S_2) \in \rightarrow\) (or equivalently \(S_1 \xrightarrow{\alpha} S_2\)) means that the system can move from the state \(S_1\) to the state \(S_2\) through the action \(\alpha\). CCS syntax is based on the following elements:

- A set \(I = \{a, b, \ldots\}\) of input actions, a set \(O = \{\bar{a}, \bar{b}, \ldots\}\) of output actions, and a set \(L = I \cup O\) of visible actions.
- A function \(\gamma : L \rightarrow L\) such that \(\forall a \in A, \bar{a} \in \bar{A}\) and \(\forall \bar{a} \in \bar{A}, \bar{a} = a, a \in A\).
- A set \(\text{Act} = L \cup \{\tau\}\) (\(\tau\) is the internal action) of actions or signals.
- A set \(K\) of constants

The syntax of a CCS agent is defined as follows:

\[
E ::= 0 \mid \mu. E \mid E + E \mid E | E \mid E \setminus L \mid E[f] \mid Z
\]

where \(\mu \in \text{Act}, L \subseteq I, Z \in K\) and \(f : \text{Act} \rightarrow \text{Act}\) is such that \(f(\bar{a}) = \bar{f(a)}\), \(f(\tau) = \tau\). Moreover, for every constant \(Z \in K\) there must be the corresponding definition: \(Z \stackrel{\text{def}}{=} E\). Intuitively, 0 is the empty process, which cannot do any action; \(\mu. E\) can perform action \(\mu\) and then behaves like \(E\); \(E_1 + E_2\) is a process which can nondeterministically choose to behave like \(E_1\) or \(E_2\); \(E_1|E_2\) is the parallel composition of \(E_1\) and \(E_2\), where the executions of the two systems are interleaved, possibly synchronised on complementary input/output actions; \(E \setminus L\) can execute all the actions \(E\) is able to do, provided that they do not belong to \(L \cup \bar{L}\); finally, if \(E\) can execute action \(\mu\), then \(E[f]\) performs \(f(\mu)\).

Let \(E\) be the set of CCS agents, ranged over by \(E, F\) (possibly indexed); in general if we say ‘a certain system \(E\)’ we mean the CCS agent \(E\). Let \(L(E)\) denote the sort of process \(E\), i.e., the set of actions executable by \(E\). Then, the operational semantics of CCS is the lts \((\mathcal{E}, \text{Act}, \rightarrow)\) where the transition relation \(\rightarrow \subseteq \mathcal{E} \times \text{Act} \times \mathcal{E}\) is not reported for lack of space (see [13]).

A widely used system composition operator requires that two systems are composed in parallel by forcing the synchronisation on their common (complementary) actions. This can be implemented in CCS by means of parallel composition (which permits synchronizations) and restriction (which forbids asynchronous moves, hence forcing synchronizations):
**Definition 2.2** The system composition of two agents $E$ and $F$ is \((E|F) \setminus (L(E) \cap L(F))\).

Now we firstly introduce execution traces and trace equivalence on CCS process terms; then, we report the definition of the standard notion of equivalence on CCS agents: observational equivalence, based on weak bisimulation.

**Definition 2.3** Let $t = \alpha_1 \ldots \alpha_n \in \text{Act}^*$ be a sequence of actions; then $E \xrightarrow{t} E'$ if and only if $E(\xrightarrow{t}) \xrightarrow{\alpha_1} \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} \xrightarrow{t} E'$. For all $E \in \mathcal{E}$ the set $T_E \subseteq (I \cup O)^*$ of traces associated with $E$ is defined as follows: given a sequence $\gamma = \alpha_1 \ldots \alpha_n$,

$$\gamma \in T_E \iff \gamma \in (I \cup O)^*, \exists E_1, \ldots, E_n \in \mathcal{E} \text{ such that } E \xrightarrow{\alpha_1} E_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} E_n$$

Two systems are trace equivalent if they have the same set of traces: $E \approx_T F \iff T_E = T_F$.

**Definition 2.4** If $t \in \text{Act}^*$, then $\hat{t} \in (I \cup O)^*$ is the sequence gained by deleting all occurrences of $\tau$ from $t$. A binary relation $B \subseteq \mathcal{E} \times \mathcal{E}$ over agents is a weak bisimulation if $(P, Q) \in B$ implies, for all $\alpha \in \text{Act}$,

(i) If $P \xrightarrow{\alpha} P'$ then, $\exists Q' : Q \xrightarrow{\hat{\alpha}} Q' \land (P', Q') \in B$

(ii) If $Q \xrightarrow{\alpha} Q'$ then, $\exists P' : P \xrightarrow{\hat{\alpha}} P' \land (P', Q') \in B$

We say that $P$ is observation equivalent to $Q$, denoted $P \approx_B Q$, if there exists a weak bisimulation $B$ such that $(P, Q) \in B$.

A few auxiliary definitions will be useful in the following.

**Definition 2.5** Let $A$ be a set. Let $\gamma, \gamma' \in A^*$ be two sequences of elements in $A$. $\gamma'$ is a subsequence of $\gamma$ ($\gamma' \prec \gamma$) if and only if $\gamma$ and $\gamma'$ are in the form:

$$\gamma = \alpha_1 \ldots \alpha_n, \quad \gamma' = \alpha_{k(1)} \ldots \alpha_{k(m)} \quad \text{with } m \leq n$$

where $k : [1, m] \to [1, n]$, is an increasing monotone function.

**Notation:** With abuse of notation $\alpha_i \in \gamma$ denotes that $\alpha_i$ is an element of the sequence $\gamma$. Let $\gamma, \gamma'$ be two sequences such that $\gamma' \prec \gamma$, then $k_{\gamma', \gamma}$ denotes the subsequence function in definition 2.5.

A set $C$ of observable actions is complete iff for every input $i \in I$ in $C$, the corresponding output $\bar{i} \in O$ is still in $C$. The set of all the complete sets of actions is denoted with $\mathcal{U} \overset{\text{def}}{=} \{U \subseteq (I \cup O) \mid U \text{ is complete} \}$.

The following functions will be used to define NDC. They are connected to the idea of security levels and assume that the set of actions $\text{Act}$ has been partitioned in two complete sets $\text{Act}_H, \text{Act}_L \in \mathcal{U}$ which represent the sets of high and low level actions, respectively.

The first function, $\text{low}$, extracts from a sequence of actions the subsequence composed only of low level actions. The second function, $\text{lowviews}$, returns the set of low traces of a certain system.
Definition 2.6 Let $\text{Act}_H, \text{Act}_L \in \mathcal{U}$. The functions low and lowviews are defined as follows:

- $\text{low}: \text{Act}^* \to \text{Act}_L^*$; for every $\gamma \in \text{Act}^*$, $\gamma' \in \text{Act}_L^*$, $\gamma' = \text{low}(\gamma)$ is such that:
  $$\gamma' \prec \gamma \land \forall \alpha_i \in \gamma, \alpha_i \in \text{Act}_L \Rightarrow \exists j: i = k_{\gamma', \gamma}(j)$$

- $\text{lowviews}: \mathcal{E} \to \mathcal{P}(\text{Act}_L^*)$. For every agent $E \in \mathcal{E}$:
  $$\text{lowviews}(E) \triangleq \{\gamma \in \text{Act}_L^* \mid \exists \gamma' \in T_E, \gamma = \text{low}(\gamma')\}$$

In the following, we assume that $\text{Act}_H$ e $\text{Act}_L$ are a partition of input and output actions.

3 Non Deducibility on Compositions

Non Deducibility on Compositions (NDC) is a transposition on the lts model of Non Deducibility on Strategies (NDS) [17]. The basic idea of NDS is the following: A Trojan Horse is viewed as a high level input “strategy” [17] (or high level process) II which chooses a new input for the system depending on the sequence of previous input/output actions; hence, a system is NDS if, for a certain strategy II, all the low level views are still possible, i.e, if II does not restrict the possible low executions (hence, it does not “interfere” with low level).

The original definition [17] is based on a very restrictive system model in which, at every clock signal, a system takes an input from every input channel and emits an output to every output one. In such a model, a strategy is simply a function which at time $n$ computes a new input using a $n - 1$ long sequence of previous input/output actions. In a CCS-like setting, a strategy cannot be represented as a function. In fact, when a system emits an output, it must wait for a corresponding external input which could never arrive: a deadlock may arise on a system output, because of the synchronous nature of CCS communication. A strategy can be represented as a function of previous input/output signals only if all such outputs have been accepted by the external user (in particular by a Trojan Horse).

A strategy may be seen as a process II which uses only high level actions. To fix a certain strategy on a system $E$, it is sufficient to consider the system $(E | II) \setminus \text{Act}_H$. So a system $E$ is NDC if for every high level process II the low level view sets of $E$ and $(E | II) \setminus \text{Act}_H$ are the same. In other words, a system $E$ is NDC if the composition of $E$ with all possible Trojan Horses does not change its set of low level views.

Definition 3.1 NDC$_H$ (Non Deducible on Compositions for actions in $H$).
Let $H \subseteq \text{Act}_H$. Then,

$$E \in \text{NDC}_H \iff \text{lowviews}(E) = \text{lowviews}((E | II) \setminus H),$$
$$\forall II \in \mathcal{E}, \mathcal{L}(II) \subseteq \text{Act}_H \cup \{\tau\}$$

NDC is defined as NDC$_H$ where $H = \text{Act}_H$: $\text{NDC} \triangleq \text{NDC}_{\text{Act}_H}$. 
This definition states that there is no way of altering the low level executions of a system $E$ by composing with it a high level agent $H$ which communicates with $E$ only through actions in $H$. Of course, this definition is difficult to use in practice: it is not possible to verify the security of a system connecting to it all Trojan Horses and observing their effect on low level computations (it is an exhaustive procedure which does not terminate, in general). We need an alternative formulation of NDC, which exploits local information, only. It is based on the following theorem, which asserts that the property of ‘transparency with respect to all Trojan Horse attacks’ corresponds to a static property of an agent $E$, namely, to a property that can be verified by examining the traces of system $E$ only.

**Theorem 3.1 (Static characterization of NDC$_H$)**

$$E \in \text{NDC}_H \iff \text{lowviews}(E) = \text{lowviews}(E \setminus H)$$

This characterisation of NDC$_H$ has the merits of being algorithmically testable (at least, for finite state systems) and useful for modular verification of secure systems. Indeed, NDC is a composable property w.r.t. the CCS operators of parallel composition and restriction; hence, a new NDC secure system may be described as a composition of NDC secure subsystems.

**Theorem 3.2** Let $E$ and $E'$ be two NDC$_H$ agents, $S \subseteq H$ and $S' \subseteq \text{Act}_L$. The following hold:

i) $E | E' \in \text{NDC}_H$,

ii) $E \setminus S \in \text{NDC}_H$,

iii) $E \setminus S' \in \text{NDC}_H$,

(iv) $(E | E') \setminus (\mathcal{L}(E) \cap \mathcal{L}(E')) \in \text{NDC}_H$, if $(\mathcal{L}(E) \cap \mathcal{L}(E') \cap \text{Act}_H) \subseteq H$.

Finally, the following theorem states that the NDC$_H$ sets are partially ordered w.r.t. set-theoretic (reversed) inclusion over the sets $H$.

**Proposition 3.1** $H, H' \subseteq \text{Act}_H : H' \subseteq H \implies \text{NDC}_H \subseteq \text{NDC}_{H'}$.

## 4 An Example: Access Monitor

In this section, the NDC property is illustrated through an example: an Access Monitor (AM for short, see figure 2), which is a process managing accesses to a certain resource. As a resource can be viewed as a set of objects containing information, then an AM is a very important element in the implementation of a multilevel security policy (it is sufficient to know all security levels of all subjects and objects to correctly manage every access).

Let us consider the usual multilevel policy (Read-down, Write-up). Let $SL$ be the set of security levels and $\leq \subseteq SL \times SL$ be a total order relation. Let $\text{ID}$, ranged over by id, be the set of user
identifiers; let also $O$, ranged over by $x$, be the set of object identifiers. Finally, let $lvl : ID \cup O \rightarrow SL$ be a function which returns the security level of a certain user $id$ or object $x$. If objects are variables (every object contains a value), then AM can be implemented as follows:

$$Monitor \overset{def}{=} \sum_{id \in ID} access_{r_{id}}(x) \cdot (\text{if } lvl(x) \leq lvl(id) \text{ then } read_{x}(y) \cdot (\overline{value}_{id}(y) \mid Monitor) \text{ else } (\overline{value}_{id}(\text{ERROR}) \mid Monitor))$$

$$+ \sum_{id \in ID} access_{w_{id}}(x, z) \cdot (\text{if } lvl(x) \geq lvl(id) \text{ then } write_{x}(z)) \cdot Monitor$$

Signals read and write are used by the Monitor for accesses to objects. They are parametrized w.r.t. the names of the objects. Signal $access_{r_{id}}(x)$ represents a request for a read operation to object $x$ made by the user with identifier $id$. The request is always accepted and a value is always returned to the user. However, the value stored in the object $x$ is returned only if the security level of $x$ is less than, or equal to, the security level of the user $id$; otherwise, an error message is transmitted.\(^2\) Note that the Monitor is ready to accept new signals before delivering the value to the user, because of the (inner) parallel composition. Signal $access_{w_{id}}(x, z)$ represents a request, made by a user $id$, to write the value $z$ into object $x$. This request is always accepted, but the write operation is really performed only if the level of $x$ is greater than the level of $id$.

The objects can be implemented as follows:

$$Obj_{x}(y) \overset{def}{=} \overline{read}_{x}(y) \cdot Obj_{x}(y) + write_{x}(z) \cdot Obj_{x}(z)$$

If there are $n$ variables then the system is:

$$Access\_monitor \overset{def}{=} (Monitor \mid Obj_{1}(0) \mid \ldots \mid Obj_{n}(0)) \\setminus L$$

where

$$L = \bigcup_{x \in \{1, n\}} Acs(Obj_{x})$$

\(^2\)Obviously, a user must get the replay to a certain read request before asking monitor for another read: a restriction we consider for the sake of simplicity.
\[ \text{Acs}(\text{Obj}_E) \triangleq \{\text{read}_E, \text{write}_E\} \]

By restricting the executions on set \( L \), we make objects not directly accessible by a user, but filtered through the interactions with the Monitor. So the user \( id \) which interacts with the Monitor can execute this set of actions:

\[ \text{Acs}_{id}(\text{Access} \_ \text{monitor}) = \{\text{access}_{r \_ id}, \text{access}_{w \_ id}, \text{value}_{id}\} \]

The set of all the actions accepted by the monitor is:

\[ \text{Acs}(\text{Access} \_ \text{monitor}) = \bigcup_{id \in ID} \text{Acs}_{id}(\text{Access} \_ \text{monitor}) \]

Since NDC is closed with respect to composition of systems then there are two different approaches to verify if \( \text{Access} \_ \text{monitor} \) is NDC secure:

(i) Verify if the whole system is NDC, or

(ii) decompose system in subsystems and verify if everyone is NDC (a possible decomposition is suggested by system definition itself: \( \text{Monitor} \) and \( \text{Obj}_E \)'s).

In the full paper (and in [2]), we prove that AM is NDC (following the first approach), while \( \text{Monitor} \) is not secure for every kind of object. In particular, we report an example of finite storage device management in which a user can transmit information using the \textit{resource full} error message. However, the compositionality property of NDC is important when the monitor is used: as a user is an agent which executes only actions of the same level, then it is NDC. So, by connecting to AM any number of users, we obtain an enlarged system which is still NDC secure.

5 Extending the Approach to Bisimulation Semantics

NDC is based on trace semantics: it requires that a system, to be secure, gives to a low user the same low trace sets for every high execution. More precisely, this statement is formalized through the following theorem.

**Theorem 5.1**

\[ E \in \text{NDC} \iff E \setminus \text{Act}_H \approx_T (E \mid \text{II}) \setminus \text{Act}_H, \forall \tau \in \mathcal{E}, \mathcal{L}(\text{II}) \subseteq \text{Act}_H \cup \{\tau\} \]

Unfortunately, trace semantics is rather weak; it is unable to distinguish systems which give different observations to a user. For instance, systems \( A = ab + a \) and \( B = ab \) are trace equivalent but the first can deadlock after a signal \( a \). Hence, a low user could distinguish two low trace equivalent systems and so have information about high level users. As a simple example, let us
implement the monitor as follows:

\[
Monitor \overset{\text{def}}{=} \left( \sum_{id \in ID} \text{access}_{r}(x). (\text{if } lvl(x) \leq lvl(id) \text{ then } \text{read}_{\pi}(y), \text{value}_{id}(y) \text{ else } \text{value}_{id}(\text{ERROR})) \right.
\]

\[+ \sum_{id \in ID} \text{access}_{w}(x, z). (\text{if } lvl(x) \geq lvl(id) \text{ then } \text{write}_{\pi}(z)) \]

\].

Monitor

Surprisingly enough, it still satisfies the NDC property, but, of course, it is not multilevel secure. Indeed, a high level user can block the monitor – and so interfere with low level users – by making a read request and then refusing to accept the corresponding answer. Trace equivalence does not detect this kind of 'attacks' and so it is necessary to substitute it with a stronger one. One possibility is weak bisimulation [13] which is the 'standard' CCS equivalence and implies other important equivalences such as testing [5] and failure [6]. We denote it with \( \approx_B \) (see section 2). The new security notion is called BNDC.

**Definition 5.1**

\[
E \in BNDC \iff E \setminus Act_H \approx_B (E \mid II) \setminus Act_H, \ \forall II \in \mathcal{E}, \mathcal{L}(II) \subseteq Act_H\cup \{\tau\}
\]

One can easily verify that the second version of Monitor does not enjoy the BNDC property.

A static characterization of BNDC – which does not involve composition with every processes II – is not immediate. A first idea could be to elaborate on the same condition we proposed for NDC. The resulting proposal is the following:

\[
E/Act_H \approx_B E \setminus Act_H
\]

where '/' denotes a hiding operator which transforms all high actions in internal \( \tau \)'s. Unfortunately, it is easy to see that it is only a necessary condition for \( E \) to be BNDC. At present, we have not yet found a static characterization of BNDC. We think this is a very interesting problem, left for future research.

To conclude this paper we propose a sufficient condition to BNDC, which is static and compositional (with \( E \Rightarrow E' \) we denote that \( E' \) is reachable from \( E \), that is \( \exists \gamma \) such that \( E \xrightarrow{\gamma} E' \)):

**Definition 5.2** Static Bisimulation NDC (SBNDC):

\[
E \in SBNDC \iff \forall E' : E \Rightarrow E', \forall E'' : \exists h \in Act_H, E' \xrightarrow{h} E'' \text{ then } E' \setminus Act_H \approx_B E'' \setminus Act_H
\]

**Proposition 5.1** The following hold:

(i) \( SBNDC \subseteq BNDC \),

(ii) \( E, F \in SBNDC \Rightarrow (E \mid F) \in SBNDC \),

(iii) \( E \in SBNDC \Rightarrow E \setminus S \in SBNDC, \text{ if } S \subseteq I \).
6 Conclusions

This paper presents a formal approach to information flow security. NDC is a security property which guarantees that no information flow from high level to low level can occur (no covert channels). It is closed with respect to system composition and can be used to verify if a CCS agent represents a multilevel secure system.

It has been observed that such a property may be too weak in some cases, essentially because it is related to trace equivalence; hence, a stronger property, based on bisimulation [13], has been proposed. Future research is concerned with a static characterization of BNDC and further extensions to other equivalences (e.g., testing [5], failure [6], ...).

The final goal of the present research is the implementation of a verification tool, in order to test such extensions on a number of significative examples.

References


Concurrent Kripke Structures

Vineet Gupta*
Dept. of Computer Science
Stanford University, CA 94305
vgupta@cs.stanford.edu

July 15, 1993

Abstract

We consider a class of Kripke Structures in which the atomic propositions are events. This enables us to represent worlds as sets of events and the transition and satisfaction relations of Kripke structures as the subset and membership relations on sets. We use this class, called event Kripke structures, to model concurrency. The obvious semantics for these structures is a true concurrency semantics. We show how several aspects of concurrency can be easily defined, and in addition get distinctions between causality and enabling, and choice and nondeterminism. We define a duality for event Kripke structures, and show how this duality enables us to convert between imperative and declarative views of programs, by treating states and events on the same footing. We provide pictorial representations of both these views, each encoding all the information to convert to the other.

We define a process algebra of event Kripke structures, showing how to combine them in the usual ways—parallel composition, sequential composition, choice, interaction and iteration. Various properties of these connectives like associativity and distributivity are proved. We then show that Winskel's event structures can be embedded in the class of event Kripke structures, and define partial synchronous composition, the primary connective for event structures, for event Kripke structures, and show its equivalence to Winskel's definition.

Note: This work was done jointly with Vaughan Pratt, Stanford University.

1 Event Kripke Structures

In the past decade, various desirable properties of models of concurrent processes have been discussed: branching time, true concurrency, action refinement, disjunctive enabling, conflict, interchangeable state and event based views, real time consistency etc. Various models, having many of these properties have been proposed: Petri nets [Pet62], synchronization trees [Mil80], Mazurkiewicz traces [Maz77], pomsets [Gra81, Pra82], event structures [NPW81, Win86], causal automata [Gun91, Gun92] etc.

A careful examination of these models reveals that most of them exhibit concurrent behavior by explicitly inserting concurrency into a basically sequential model, such as by having many tokens

*This work was supported by ONR under grant number N00014-92-J-1974, a gift from Mitsubishi and an Accel Partners Fellowship.
in a network or multiple strings being read concurrently. Thus they end up defining concurrency instead of modelling it, and one can never be sure whether the definition is good enough. For example, in event structures, conflict and enabling are primitives, but other primitives could be necessary to capture concurrency fully. In this paper we attempt to discover what true concurrency is from a model of concurrency, rather than defining it a priori, in the same vein as discovering logic as the theory of Boolean algebras rather than trying to characterize it with a long list of equations. We start from a well known model for temporal logic, a Kripke structure, and use it to derive our model.

A Kripke structure \((W, \models, \Pi_0, \rightarrow)\) consists of a binary transition relation \(w \rightarrow w'\) between states \(w, w' \in W\), and a binary satisfaction relation \(w \models p\) between states in \(W\) and atomic propositions \(p \in \Pi_0\), extended by induction on the height of formulas to compound propositions \(p \land q, \neg p, \ldots \in \Pi\).

We study a special class of Kripke structures which looks more like event structures [NPW81]. The relevant property of an event here is that once it becomes true, it stays true. We restrict the elements of \(\Pi_0\) to events, that is \(\forall w, w' \in W, w \models p\) and \(w \rightarrow w'\) implies \(w' \models p\). Now the transition relation between states can be derived from the satisfaction relation by \(w \rightarrow w'\) iff \(\forall p, w \models p \Rightarrow w' \models p\). In fact we can treat each state as a set of the atomic propositions which are true in it (we identify states which have the same sets of propositions true in them, by extensionality). Thus we define an event Kripke structure, referred to as an eks, as a pair \((E, Q)\), where \(E\) is a set of events and \(Q \subseteq 2^E\) a set of states. Then \(q \models e\) iff \(e \in q\) and \(q \rightarrow q'\) iff \(q \subset q'\). An equivalent way of describing an eks is as a \(|E| \times |Q|\) boolean matrix, with each column being the characteristic function of a state.

In addition to having all the desirable properties of models of concurrency mentioned above, eks's can distinguish between enabling an action and causing an action, a distinction which has not been made in any of the above models. In fact in some of the above models, causality has been deliberately erased by insisting on algebraic lattices [Dro89], presumably to match Petri nets and domain theory. Our model shows that not only is this unnecessary, it actually simplifies mathematics to allow this distinction. One has only to compare the definition of an event domain in [Dro89] with the definition of an eks to be convinced of this.

In the next sections, we will study some properties of event Kripke strutures. We show how aspects of concurrent behavior are naturally present in an eks, and give a process algebra to combine eks's in various ways, showing how these operations correspond to the usual operations for concurrent processes. Finally we connect them with event structures, showing how any event structure can be represented as an eks.

These structures have previously been studied by Barr-Bar97, as instances of Chu's construction on sets, and by Lafont and Streicher-LS91 as games over 2. Brown and Gurr-BG90 have used similar structures to study Petri nets.

1.1 Theory of an eks

An eks \((E, Q)\) is a set \(Q\) of subsets of \(2^E\). We can therefore write it out as a formula in Disjunctive Normal Form in propositional logic, with variables \(E\) and clauses \(Q\), with each clause corresponding to a \(q \in Q\) containing all the variables in \(E\), negating all the ones that are not present in \(q\). We call this formula \(\Gamma\). For example, \(\Gamma((\{a, b, c\}, \{\phi, \{a\}, \{ab\}, \{abc\})) = abc \lor ab \lor \neg ab \lor abc\), writing
\( \bar{a} \) for \( \neg a \) and \( abc \) for \( a \land b \land c \). Since there is no bound on the cardinality of the sets \( E \) and \( Q \), we will allow infinite conjunctions and disjunctions in our formulas. So an eks can be equivalently represented as an infinitary boolean propositional formula \( \Gamma \). This is usually the most compact representation of an eks, though the matrix representation is computationally the most efficient.

The theory of an eks is now defined to be all infinitary propositional logic formulae \( \phi \) such that \( \Gamma \to \phi \) is a tautology. For example, the formula \( \neg c \lor b \) is in the theory of the eks mentioned above. If a formula \( \phi \) is in the theory, we write \( \Gamma \models \phi \).

In later sections we shall show how the theory of an eks expresses all the interesting facts about the eks. In particular, we can determine whether an event precedes or follows another, whether two events are in conflict etc.

2 Duality

In the definition, we represented an event structure as a set of events, which were primitive, and a set of states formed from those events. However, there is nothing special about events—we could as well take the states as primitive, leading to a duality between states and events, which we discuss in this section.

Pictorial Representation of an EKS. An eks \((E, Q)\) is any subset \( Q \) of \(2^E\). Thus we can draw it as an \( E \) dimensional cube, and then mark the elements of \( Q \) on the cube as dots. This is called the partial boolean algebra representation of an eks, as the set \( 2^E \) forms a boolean algebra. In the infinite case this is a complete atomic boolean algebra, a CABA\(^1\). For example, the eks mentioned above, \((\{a, b, c\}, \{\phi, \{a\}, \{ab\}, \{abc\})\) can be represented as follows

```
    abc
   /   \
ab  \   \a
  /     \
 0
```

Partial Boolean Algebra

```
  abc
  \    \\
 ab  \   \\
a  \    \\
 0
```

Partial Distributive Lattice

Figure 1

From the above description of a partial boolean algebra, the dots may occupy only a few dimensions of the boolean algebra. Since the dots are all the useful information of the eks, there seems no point in carrying around the other dimensions, which we dump by insisting that all eks's we consider must have the \( T_0 \) property\(^2\)—the formula \( a \leftrightarrow b \) must not be present in the theory of the eks for any pair of events \( a, b \), since \( a \leftrightarrow b \) means these are equivalent events which can be identified. As a result of this identification, the boolean algebra cannot be very large, because it must be generated from the dots using the logical operations\(^3\). We will assume the \( T_0 \) property for all eks's henceforth.

\(^1\)In the finite case, “complete atomic” is automatic and hence redundant.

\(^2\)This terminology comes from topology, by considering \( E \) as the set of points and \( Q \) as the set of open sets. Eks's are a generalization of topological spaces, in that they do not impose any conditions on the open sets.

\(^3\)If the eks is \( T_0 \), then for each event in some state, we can take the intersection of all the states containing this event and the complements of all the states not containing it to get a singleton set containing this event. There may
The information given in the partial boolean algebra can be more succinctly and transparently given in the figure on the right. That figure was obtained by taking the dots and retaining only those holes which can be formed from the dots using unions and intersections but not complements (in this case no holes could be thus generated). The resulting structure is called a partial distributive lattice (pdlat), by analogy with partial boolean algebras. The lattice made up of the dots and holes is a profinite distributive lattice.\footnote{In the finite case this is just a distributive lattice. In the infinite case it is most easily understood as a lattice isomorphic to the lattice of order filters or upward-closed subsets of a partial order. This is an even stronger requirement than being a complete and completely distributive lattice, witness the unit interval $[0,1]$ of reals standardly ordered, which meets that condition but is too connected to be profinite. Dedekind cuts as order filters in the unit interval of rationals however is by definition profinite, the difference being that the rationals appear twice. Eliminating the extra copies connects up the real interval and eliminates profiniteness, along with the means for recovering the rationals.} Note that we do not always achieve such succinctness, for example, when the partial boolean algebra is made up of dots only the pdlat is just the partial boolean algebra.

It is possible for us to use pdlats to represent eks's because of the following proposition:

**Proposition 1** Partial Boolean algebras are in 1-1 correspondence with partial distributive lattices.

**Proof:** The distributive lattice may be obtained from the CABA as the lattice of order filters of the set of dots as partially ordered by the CABA, which is embedded in the CABA; equivalently, the complete sublattice of the CABA generated by the set of dots. Conversely the CABA may be obtained from the profinite distributive lattice as the power set of its dimensions or primes, which embeds that lattice by embedding each element as the set of primes below it; equivalently, generate the Boolean complements of each dot and close up under meets and joins.

**Duality between States and Events.** Thus far we have considered events as primary and states as sets of events. However we can alternatively consider states as atomic, and events as sets of states, with a state belonging to an event iff the event belongs to that state. If we are willing to be more adventurous with foundations of set theory, we may do both—the resulting circularity in the membership relation can be coped with by dropping the Foundation Axiom of ZF set theory and replacing it by e.g. Aczel's Anti-Foundation Axiom. This results in states and events being treated on the same footing, something which has not been done in previous models.

**Schedules and Automata.** The operational interpretation of an eks $A = (E, Q)$ is as an automaton, which does events in $E$ and advances from one state in $Q$ to another, according to the transition relation (which is just the subset relation, as stated above). We call the states in $Q$ the real states of $A$ and the holes in the underlying lattice the forbidden states, namely the states in which the automaton should never find itself. Thus the automaton can rest and make choices only in the real states, while it must rush through the forbidden states. We take the view the any choices made in the forbidden states are made by the environment. The dimensions of the lattice, defined as equivalence classes by Droste[Dro89], are the events of $E$.

We can dualise $A = (E, Q)$ by defining $A^\perp = (Q, E)$, where $Q$ is the set of states, and $E$ is the set of events each of which is taken to be a set of states. We give a few examples of this in figure 2,
drawing the eks’s as pdlats. Drawn as its dual\(^5\), the eks is a schedule. The dots now represent the events which have to be performed, and the lattice structure represents the constraints on these events, some of which we will discuss in the next section.

In fact the schedule and automaton are different ways of looking at the same eks. We can also write the theory of an eks in terms of the state variables, and the two theories have exactly the same information because they are derived from the same boolean formula written in terms of different variables. However, in a schedule, the direction of time is downwards, as indicated by the little arrow on the side, whereas in an automaton, time flows upwards. This is the only way to determine whether an unlabeled pdlat is a schedule or an automaton: any pdlat can be interpreted in either way.

We can label the events of an eks by a labelling function \(\lambda : E \rightarrow Act\) going from events to actions. This allows us to consider the case when multiple instances of the same action occur. We will not go into this any further except to remark that we can have complementary action labels and silent actions, allowing synchronization, hiding and relabelling as in CCS[Mil90].

**Time and Information Duality.** The duality between states and events enables us to see a similar duality between time and information. Events are regarded as instantaneous, occurring at a particular time, and they add to the total accumulated information. Dually, no information is accumulated in a state, which however adds to the total amount of time that has passed since the beginning of the process. Thus we can plot the progress of a behavior on an graph with time on the x-axis and information on the y-axis. The states are then horizontal line segments, and the events are vertical. The graph is monotonically increasing, and we shall have more to say about this in the conclusion. The graph is also piecewise horizontal and vertical, but as the number of events grows to infinity it starts looking like a monotonically increasing curve.

### 3 Modelling Concurrent Behavior

We now show how various concurrent phenomena can be represented in an eks. The different forms of formulae in the theory of the eks \(A\) represent different forms of constraints on the events. We can also use the theory of the dual eks to represent constraints on states, and some constraints are easier to comprehend this way. We use \(\Gamma^+\) for the formula of the dual eks \(A^+\).

**Temporal precedence.** For any two events \(a, b\), if \(\Gamma \models b \rightarrow a\), then no state contains \(b\) and not \(a\). This means that \(b\) can be executed only after \(a\) has been executed. This means that \(a\) precedes \(b\) in time, and the collection of all such constraints defines a temporal order on the events of \(G\). We write \(a \leq b\) for any such pair of events. Note that the temporal order is the converse of the logical order, and this is the reason for the downwards flow of time for the schedule. (We always have the logical order going up.)

**Conflict.** If \(\Gamma \models \neg(a \land b)\), then no state contains both \(a\) and \(b\). This means that we can never execute both \(a\) and \(b\), which is interpreted as a conflict between \(a\) and \(b\). This can be generalised to conflict between arbitrary numbers of events. While this notion has been explicitly introduced in event structures [NPW81, Win86], it follows quite naturally from our definition. It is illustrated

\(^5\)In terms of the matrix representation, forming the dual corresponds to transposing the matrix. This is true because in taking the dual we interchange the states with events, and take the converse of the membership relation.
in Figure 2(c-d), where $b$ and $c$ are in conflict.

![Automata Diagrams](image)

**Figure 2**

*Causality and enabling.* If $\Gamma \models a \leftrightarrow b \land c$ then any state containing $b$ and $c$ must also have $a$. We interpret this as $b$ and $c$ together cause $a$. This can be generalized to an arbitrary formula on the right side of the implication. Note that the $T_0$ condition does not allow an event to be caused by a single event, as the two events are identified.

The above eks is distinct from the eks in which $\Gamma \models a \rightarrow b \land c$. Then $b$ and $c$ just enable $a$, which means that while it is necessary for them to be done to do $a$, it is not sufficient, as $a$ need not be done immediately. This can also be generalized to an arbitrary condition, including the singleton event, when it becomes temporal precedence. The difference between these is like the difference between the necessary and sufficient conditions for a theorem, while the enabling formula represents the necessary conditions, the causing formula is both necessary and sufficient.

*Nondeterminism.* Dual to the above notion is the notion of guarded vs. blind choice. If $\Gamma \models a \leftrightarrow b \lor c$, then as soon as $a$ is done, one of $b$ or $c$ must be done immediately. This means that any information obtained by doing $a$ could not be used in selecting between $b$ and $c$, making this a blind choice. This is more clearly visible from the automaton side in Figure 2(c). The choice between states $v$ and $w$ has to be made at the forbidden state above $u$. We interpret this as being made by the environment, making this a nondeterministic choice for the automaton.

Just having $\Gamma \models b \lor c \rightarrow a$ would make it a guarded choice, whence any information obtained from $a$ can be used to make a deliberate choice. On the automaton side, the choice between $v$ and $w$ is made in the state $u$, as $\Gamma^\perp \models u \leftrightarrow v \land w$, and this state contains the information gathered by doing $a$ to decide which way to go. This is like a conditional branch in a program, where the event $a$ checks some condition, and then either one of two alternatives is chosen.

If $b$ and $c$ were not in conflict here, then this would not be a real choice. The hole at the top of the automaton would be a permitted state, and then $b$ and $c$ could be executed concurrently, showing that this model has true concurrency. In fact interleaving semantics would not work here at all, as is clear from Figure 2(a), when $a$ has to be done concurrently with either $b$ or $c$. 
Remark. The reader may have noticed that when a meet is present in the schedule, a corresponding join is absent in the automaton, and vice versa, and similarly for joins. For example, \( a \) is the join of \( b \) and \( c \) in Figure 2(c), and the meet of \( \nu \) and \( \omega \) is not present (it is a hole). This is a general principle, so the dual of a set (which has no structure) is a highly structured object, a CABA, and the dual of a meet semilattice is another meet semilattice, as it can have no joins.

Disjunctive Enabling. If \( \Gamma \models c \rightarrow a \lor b \), then in order for \( c \) to happen, at least one of \( a \) or \( b \) must have happened. For example, in a candy machine, inserting either a dollar bill or 4 quarters will enable the machine to supply a candy bar. This is called disjunctive enabling [Win86, Gun91, Gun92]. The dual behavior is postponed concurrency, given by \( \Gamma \models a \land b \rightarrow c \), whereby \( a \) and \( b \) can be done concurrently only after \( c \) is completed.

If we draw an automaton for disjunctive enabling with three events, i.e. one which has \( \Gamma \models c \rightarrow a \lor b \), then it looks like a 3-dimensional cube, with all vertices marked by dots except the vertex \( c \), which is the only vertex which does not satisfy the formula \( c \rightarrow a \lor b \). This illustrates a general method for programming an automaton on \( n \) events from a logical specification—from the \( n \)-cube, mark with dots only those vertices which satisfy all the equations. The resulting structure gives a partial boolean algebra for the specification, and this can be converted into a pdat or eks.

Action Refinement. Given an eks \((E,Q)\), and another eks \((E_1,Q_1)\) where \( \phi \in Q_1 \) and \( E_1 \subseteq Q_1 \), (i.e. it is conflict free), we can refine any event \( e \) in \((E,Q)\) by \((E_1,Q_1)\) to get a new eks \((E',Q')\) as follows: \( E' = E - \{e\} \cup E_1 \), and the set \( Q' \) is obtained from \( Q \). If \( q \in Q \), and \( e \notin q \), then \( q \in Q' \). If \( e \in q \), then \( q - \{e\} \cup E_1 \subseteq Q' \). Also, for these \( q \), if \( \forall a \in q, a \neq e \Rightarrow e \notin a \), then for each \( q' \in Q_1 \), \( q - \{e\} \cup q' \subseteq Q' \).

What this means is that instead of doing \( e \), we do the eks \((E_1,Q_1)\). The third clause assures that the intermediate states of \((E_1,Q_1)\) are added iff no event in the state is preceded by \( e \), since otherwise the event \( e \) must have been completed in some previous state. Thus if we looked upon events as intervals, a dot means that the event before it and the event after it are disjoint in time, whereas a hole means that they overlap.

4 Process Algebra

We now define a process algebra for combining various eks’s. The operations considered are parallel composition \( A|B \), sequential composition \( AB \), choice or summation \( A\cup B \), iteration \( A^\omega \) and interaction \( A \circ B \). In order to define sequential composition and iteration, we add the concept of final states to the definition of an eks. An eks now is \((E,Q,F)\), where \( F \subseteq Q \). If \( A = (E,Q,F) \) is an eks, we define functions \( E(A) = E, Q(A) = Q \) and \( F(A) = F \), giving respectively the events, states and final states of \( A \). Note: in the following expressions, the set union operator has the lowest precedence.

Parallel composition \( A|B \) of \( A \) and \( B \) is the independent execution of \( A \) and \( B \). We assume \( E(A) \cap E(B) = \phi \). Define \( E(A|B) = E(A) \cup E(B), Q(A|B) = \{ q \mid q = q_1 \cup q_2, q_1 \in Q(A), q_2 \in Q(B) \} \) and \( F(A|B) = \{ q \mid q = q_1 \cup q_2, q_1 \in F(A), q_2 \in F(B) \} \). This is an eks as \( F(A|B) \subseteq Q(A|B) \subseteq 2^E(A|B) \). Note that any events which can be executed from a state in \( A \) can be executed from any corresponding state in \( A|B \), and similarly for \( B \), showing that this represents independent execution of \( A \) and \( B \). A state in the composition is final iff both \( A \) and \( B \) are in their final states.
The symbol for parallel composition used here is to conform to the usage in the process algebra community. However, our preferred symbol for it is $+ [GP93]$, since this is a coproduct in an appropriate category. Consequently, we prefer $0$ for the eks $1$ defined below, since it is an initial object in that category.

Sequential composition $A; B$ represents the execution of $B$ after completing the execution of $A$, i.e., after $A$ has reached a final state. Once $B$ starts executing, $A$ may not execute any further. Define $E(A; B) = E(A) \cup E(B) \times F(A)$, i.e., have one copy of $B$ for each final state of $A$. Then $Q(A; B) = Q(A) \cup \{q \mid q = q_1 \cup q_2 \times \{q_1\}, q_1 \in F(A), q_2 \in Q(B)\}$, saying that the automaton can be in a state of $A$, or having reached a final state of $A$, execute events in the copy of $B$ associated with that final state of $A$. $F(A; B) = \{q \mid q = q_1 \cup q_2 \times \{q_1\}, q_1 \in F(A), q_2 \in F(B)\}$, saying that a final state is one in which $A$ in a final state and $B$ has also reached a final state afterwards.

Choice or summation $A \sqcup B$ is the execution of $A$ or $B$ but not both. We define $E(A \sqcup B) = E(A) \cup E(B), Q(A \sqcup B) = Q(A) \cup Q(B), F(A \sqcup B) = F(A) \cup F(B)$. If $E(A)$ and $E(B)$ are disjoint, this executes either one of $A$ or $B$ but not both, and once having started $A$, no event in $B$ can be executed, since there is no state containing both events of $A$ and events of $B$. However, we do not insist on the disjointness, since we wish to include the possibility of choosing between two copies of $A$ etc.

Define the null eks $\Phi$ to be $(\phi, \phi, \phi)$, the eks which does nothing. Also define the eks $1$ to be $(\phi, \{\phi\}, \{\phi\})$, which will be the identity for $A; B$ and $A|B$.

**Proposition 2** The following properties hold

1. $A \sqcup A = A$
2. $A \sqcup B = B \sqcup A$
3. $A \sqcup (B \sqcup C) = (A \sqcup B) \sqcup C$
4. $A \sqcup \Phi = A$
5. $A \sqcup 1 \cong A$
6. $A \sqcup B \cong B \sqcup A$
7. $A \sqcup (B|C) \cong (A \sqcup B)|C$
8. $A; 1 \cong 1; A \cong A$
9. $\Phi; A = \Phi$
10. $A; (B; C) \cong (A; B); C$
11. $(A \sqcup B)|C \cong A|(C \sqcup B); C$
12. $(A \sqcup B); C \cong A; C \sqcup B; C$
13. $C; (A \sqcup B) \cong C; A \sqcup C; B$

**Proof:** The first seven properties follow from the properties of set union. The eighth follows as the Cartesian product of any set with the singleton is isomorphic to itself. The ninth holds as the product of an empty set with anything is empty. In the next 4 properties, the arguments for the equality or isomorphism of $F$ are similar to the respective arguments for $Q$, and so are omitted.

10. $E(A; (B; C)) = E(A) \cup E(B; C) \times F(A) = E(A) \cup E(B) \times F(A) \cup E(C) \times F(B) \times F(A)$. But $F(B) \times F(A) \cong F(A; B)$, so we get $E(A; (B; C)) \cong E((A; B); C)$. 

8
\[ Q(A; (B; C)) = Q(A) \cup \{ q \mid q = q_1 \cup q_2 \times \{q_1\}, q_1 \in F(A), q_2 \in Q(B; C) \} \]
\[ = Q(A) \cup \{ q \mid q = q_1 \cup q_2 \times \{q_1\}, q_1 \in F(A), q_2 \in Q(B) \} \]
\[ \cup \{ q \mid q = q_1 \cup q_2 \times \{q_1\} \times \{q_2\} \times \{q_1\} \times \{q_2\}, q_1 \in F(A), q_2 \in F(B), q_3 \in Q(C) \} \]
\[ \cong Q(A; (B; C)) \cup \{ q \mid q = q_1 \cup q_2 \times \{q_1\}, q_1 \in F(A; B), q_2 \in Q(C) \} \]
\[ \cong Q((A; B); C) \]

(11) \[ E((A \cup B); C) = E(A) \cup E(B) \cup E(C) = E((A; C) \cup (B; C)). \]

\[ Q((A \cup B); C) = \{ q \mid q = q_1 \cup q_2 \cup q_3 \in Q(A \cup B), q_2 \in Q(C) \} \]
\[ = \{ q \mid q = q_1 \cup q_2, q_1 \in Q(A), q_2 \in Q(C) \} \]
\[ \cup \{ q \mid q = q_1 \cup q_2, q_1 \in Q(A \cup B), q_2 \in Q(C) \} \]
\[ = Q((A; C) \cup (B; C)) \]

(12) \[ E((A \cup B); C) = E(A) \cup E(B) \cup E(C) \times F(A \cup B) = E((A; C) \cup (B; C)), \]

as \[ F(A \cup B) = F(A) \cup F(B). \]

\[ Q((A \cup B); C) = Q(A \cup B) \cup \{ q \mid q = q_1 \cup q_2 \cup q_3 \in Q(A \cup B), q_2 \in Q(C) \} \]
\[ = Q(A) \cup \{ q \mid q = q_1 \cup q_2, q_1 \in F(A), q_2 \in Q(C) \} \]
\[ \cup Q(B) \cup \{ q \mid q = q_1 \cup q_2 \times \{q_1\}, q_1 \in F(B), q_2 \in Q(C) \} \]
\[ = Q(A; C) \cup Q(B; C) \]
\[ = Q((A; B) \cup (B; C)) \]

(13) Similar to (12). These distributive laws can be extended to an infinite summation also. ■

The iteration of \( A \), denoted \( A^* \), represents the repeated execution of \( A \) finitely many times, as proved below. Define \( E(A^*) = \{(a, i) \mid a \in E(A), i \in \mathbb{N}\} \), where \( \mathbb{N} \) is the set of natural numbers.

Then \( Q(A^*) = \{\phi\} \cup \{ q \mid q = \bigcup_{0 \leq i \leq n} (q_i, i) \cup (q_n, n), q_i \in F(A) \text{ for } i < n, q_n \in Q(A), n \in \mathbb{N}\} \),

where \( (q, i) = \{(a, i) \mid a \in q\} \). The set of final states is defined similarly, \( F(A^*) = \{\phi\} \cup \{ q \mid q = \bigcup_{0 \leq i \leq n} (q_i, i), q_i \in F(A) \text{ for } i \leq n, n \in \mathbb{N}\} \).

**Proposition 3** \( A^* \cong 1 \cup A \cup A; A \cup A; A; A \cup \ldots \)

**Proof:** Since the event sets \( E(A) \) and \( E(B) \) must be disjoint for \( A; B \), we index the events of \( A; A; A \ldots \) with the number of the occurrence of the number. Then the event sets on both sides of the equation are identical.

Consider the states in \( A^n = A; A; \ldots; A \), i.e. \( A \) repeated \( n \) times. For \( n = 1 \) these are just \( Q(A) = \{ q \mid q = \bigcup_{0 \leq i \leq n} (q_i, i) \cup (q_n, n), q_i \in F(A) \text{ for } i < n, q_n \in Q(A), n = 1\} \) and \( F(A) = \{ q \mid q = \bigcup_{0 \leq i \leq n} (q_i, i), q_i \in F(A) \text{ for } i \leq n, q_n \in Q(A), n = 1\} \).

Assume this is true for \( n = m \). Then \( Q(A^{m+1}) = Q(A^m) \cup \{ q \mid q = q_1 \cup (q_{m+1}, n, m+1), q_1 \in F(A^m), q_{m+1} \in A\} \) by definition. But \( F(A^m) = \{ q \mid q = \bigcup_{i \leq m} (q_i, i), q_i \in F(A)\} \). So we get the required form for \( Q(A^{m+1}) \), i.e. \( \{ q \mid q = \bigcup_{0 \leq i \leq n} (q_i, i) \cup (q_n, n), q_i \in F(A) \text{ for } i < n, q_n \in Q(A), n \leq m\} \), and similarly for \( F(A^{m+1}) \).

Since \( Q(A^*) \) is the union of all these state sets, and \( F(A^*) \) is the union of all the final state sets, we get an isomorphism between the state sets and the final state sets on both sides. ■

This gives us all the necessary properties of \( A^* \). In particular, we can prove that \( (A^*)^* \cong A^*; A^* \cong A^* \) and \( (A \cup B)^* \cong (A^*; B^*)^* \). \( A^* \) is the least fixed point of the function \( f(A) = 1 \cup A; f(A) \). We can similarly define \( A^\dagger \) as the fixed point of \( f(A) = 1 \cup A; f(A) \), mentioned earlier in [Pra86].

Interaction, \( A \otimes B \), earlier known as orthocurrence or flow [Pra86, CCMP91], represents one
process flowing through another. For example, if process \( A \) is three trains running sequentially, and process \( B \) represents four stations on the track, then there are \( 3 \times 4 = 12 \) events, corresponding to each train arriving at each station. For each train, the stations must arrive in the same order, and for each station the trains must arrive in the same order too. This leads us to define \( E(A \otimes B) = E(A) \times E(B) \), and \( Q(A \otimes B) = \{ q \subseteq E(A \otimes B) \mid \forall b \in E(B), if \ q_b = \{ a \mid (a, b) \in q \}, then q_b \in Q(A) \}, and similarly for events in \( E(A) \). \( F(A \otimes B) \) is defined similarly, by replacing \( Q \) by \( F \) in this definition.

**Proposition 4** The following properties hold

\[
\begin{align*}
(1) & \quad A \otimes B \cong B \otimes A \\
(2) & \quad A \otimes (B \otimes C) \cong (A \otimes B) \otimes C \\
(3) & \quad (A|B) \otimes C = (A \odot C)|(B \otimes C)
\end{align*}
\]

**Proof:**

We omit the proofs of the first two identities, as they follow from the definition.

\[
E((A|B) \otimes C) = E((A \otimes C)|(B \otimes C)), \text{ as } \times \text{ distributes over set union.}
\]

\[
Q((A|B) \otimes C) = \{ q \mid \forall c \in E(C), q_c \in Q(A|B), \forall d \in E(A|B), q_d \in Q(C) \} = \{ q \mid \forall c \in E(C)[\forall \alpha \in E(A)[q_{(q_1)_\alpha} \in Q(C)]], \forall a \in E(B), q_a \in Q(C) \}
\]

In the third step, we separated out the \( q \) into two states, one whose first components came from \( A \), and the other, whose first components came from \( B \). Then we used the fact that \( q_\alpha \) is the same as \( (q_1)_\alpha \), and same for \( q_b \). The proof for the final states is similar. Interaction does not distribute over sequential composition or choice. \( \blacksquare \)

## 5 Relationship with Event Structures

We show that eks’s subsume the most general case of event structures as defined by Winskel [Win88b]. As we have shown above, conflict and enabling of events can be expressed in an eks. Since these are the only two concepts required in the definition of an event structure, we can show that event structures can be modeled by eks’s, for what amount to trivial reasons. However event structures cannot distinguish causality and enabling, showing that eks’s are strictly more general than event structures.

An event structure is a set of events \( E \), with a conflict relation \( \# \) and an enabling relation \( \vdash \), defined in [Win88a]. A configuration is any subset of events which could have occurred in the event structure, so it must be conflict free, and every event must be caused by some previous events in the set according to the enabling relation. Given an event structure \( E = (E, \#, \vdash) \), let \( \mathcal{F}(E) \) be its set of configurations. We form the eks \( G = (E, \mathcal{F}(E)) \), which is clearly equivalent to the event structure, and this leads us to the following theorem.
Proposition 5 For every event structure that is formed from its family of configurations, there is an eks which has the same properties:

1. If $e \neq e'$ in the event structure, then $\Gamma(G) \models \neg(a \land b)$.
2. If $X_1 \models e, \ldots, X_n \models e$ in the event structure, then $\Gamma(G) \models e \rightarrow \bigvee X_i$, where $X_i$ is taken to be the conjunction of the events in $X_i$.

The labelling of eks’s enables this embedding to respect the labelling function of labelled event structures too. The definition of sum in Winskel corresponds to the definition of (disjoint) sum above:

Proposition 6 Let $E_0$ and $E_1$ be two event structures, and $G_0$ and $G_1$ be their corresponding eks’s. Then $G = G_0 \cup G_1$ is the eks corresponding to $E = E_0 + E_1$ as defined by Winskel.

Proof: The sum of $E_0$ and $E_1$ is the disjoint union of their events along with their individual conflict and enabling relations, such that every event of $E_0$ is in conflict with $E_1$. Then $E(G)$ corresponds to the events of $E$. Also, any configuration of $E$ is a configuration of $E_0$ or of $E_1$, as no mixed configuration is possible. This is exactly the state set of $G$.

There is no universally accepted definition of sequential composition for event structures. Baeten and Vaandrager[BV92] define sequential composition, and our definition does agree with theirs operationally. They define a special event, a $\bigvee$ which is interpreted as the last event of the process executing the event structure. Sequential composition $E_0; E_1$ is then defined by refining each $\bigvee$ of $E_0$ with $E_1$. We can mimic this by letting each configuration in which a $\bigvee$ can be performed next as a final state of $G_0$, the eks corresponding to $E_0$, and omitting $\bigvee$ from the set of events. The correspondence between the states of $G_0; G_1$ and the configurations of $E_0; E_1$ is then clear.

In [Win88b], Winskel defined the partial synchronous composition of two event structures. It is possible to define this for eks’s also, in a way that respects the above embedding.

We let $A || B$ stand for the partial synchronous product of $A$ and $B$. Then $E(A || B) = E(A) \cup E(B) \cup E(A) \times E(B)$. $Q(A || B)$ is a set of subsets of $E(A || B)$, each element $q$ of which satisfies the following properties:

1. The sets $q \cap E(A)$, and for each $b \in E(B) \{a \mid (a, b) \in q\}$ are pairwise disjoint. Similarly, the sets $q \cap E(B)$, and for each $a \in E(A) \{b \mid (a, b) \in q\}$ are pairwise disjoint.
2. Define $q_A = (q \cap E(A)) \cup Pr_1(q \cap E(A) \times E(B))$ and $q_B = (q \cap E(B)) \cup Pr_2(q \cap E(A) \times E(B))$. Then $q_A \in Q(A)$ and $q_B \in Q(B)$.
3. If $q' \subseteq q_A$ and $q' \in Q(A)$ then $\exists q'' \in Q(A || B)[q'' \subseteq q$ and $q' = q''_A]$ and the same for $B$.

The first property says that an event of $A$ may either occur alone or may synchronize with exactly one event of $B$. The second condition says that if we extract all the events of $A$ which have happened in a state $q$, then these must form a state of $A$, called $q_A$. The third condition says that $Pr_1(X)$ is the set of first components of elements of $X$. 

11
if can reach the state \( q_A \) in \( A \) by doing some events, then we should be able to do the same events (some synchronized with events in \( B \)) to reach the state \( q \). This prevents some undesirable states, e.g., the state \( \{(a, d), (b, c)\} \) is not allowed when forming \( (a; b)|||(c; d) \). It is eliminated as there is no way to reach it by first doing \( a \) and then \( b \), as that would lead to doing \( d \) before \( c \). The only states allowed in the above example are\(^7\) \( \{\phi, a, a+b, c, c+d, a+c, a+c+d, a+c+b, a+b+c+d, ac, ac+b, ac+d, ac+b+d, ad+c, ad+c+b, cb+a, cb+a+d, a+c+bd, ac+bd\} \). All the other states are ruled out as they do not satisfy one or more of the conditions.

This definition agrees with that of Winskel[Win88b].

**Proposition 7** Let \( E_1 \) and \( E_2 \) be two event structures, and \( A_1 \) and \( A_2 \) the eks’s corresponding to them. Then the eks \( A_1||A_2 \) corresponds to the event structure \( E \) formed by the partial synchronous composition of \( E_1 \) and \( E_2 \).

**Proof:** Outline: The events of \( E \) match the events of \( A_1||A_2 \), by definition. Each state in \( A_1||A_2 \) is conflict free, by the first condition. Also, if any two events were in conflict in \( E_1 \), then they would not occur in any state of \( A_1 \), so they would remain in conflict in states of \( A_1||A_2 \), by condition 2. Finally, we can use the third condition to prove by induction that all states in \( A_1||A_2 \) are secured. Conversely, since the only states that were dropped were the ones that did not satisfy these conditions, the remaining states are exactly the configurations of \( E \).

Another model which is superficially similar to eks’s is Gunawardena’s causal automata[Gun91, Gun92]. However the semantics for causal automata are quite different from that of eks’s, in that if \( a \rightarrow b \) is true in a causal automaton, then \( b \) must have occurred before \( a \), whereas in eks’s, this condition means only that \( a \) cannot occur before \( b \). As a result, causal automata can express deadlock, which we have not been able to express using eks’s. The logical constraints that causal automata can express on events are a strict subset of the constraints expressible in eks’s, for example causal automata cannot distinguish between choice and non-determinism, or between causality and enabling.

### 6 Summary

We have defined a process algebra of eks’s, and given various properties for the connectives. We have also shown a duality for eks’s which enables us to look at each eks as a declarative program, a schedule or as an imperative program, an automaton. This would enable a programmer to choose the point of view in which it is easier to write any program, and also allows for an easy conversion between the two views. This duality is a form of Stone duality[Joh82], and we will discuss this connection elsewhere.

The natural semantics for eks’s is a true concurrency semantics, as in event structures and Petri nets. However eks’s also enable us to specify nondeterminism and choice, and causality, which the other two models do not.

The major limitation of our approach is that choosing conflicting alternatives ensures that we are in different branches of the automaton, and can never come together again. This seems to

\(^7\)We write \( ac+b \) for \( \{(a, c), b\} \).
be a feature of all existing schedule automaton dualizations [NPW81, Win86, Pra92]. It makes it necessary to unfold any loops in an automaton, and duplicating the actions, as in our definition of $A^*$. This may have undesirable consequences for algorithms using eks's, and complicate decision problems. It also does not take into account the fact that some conflicts, like in mutual exclusion, are temporary. This permanence of conflict is the reason why the information-time graphs for eks's are monotonically increasing. In other words, eks's never forget any information about any choices made in the past, even if some of this information is useless.

We hope to overcome this problem by extending schedule automaton duality to structures in which conflict is not so permanent. One approach for doing this seems to be higher dimensional automata [Shi85, Pra91, GJ92], which distinguish a mutex $b$ from $ab + ba$ by representing the former as a square with no interior, and the latter as in eks's with labelling. Eks's cannot make this distinction now, but higher dimensional automata do not yet have the schedule automata duality, and we hope to reconcile the two.

We have not shown completeness of the properties of the eks connectives, and plan to address this in the future. The set of connectives can be embellished with the other operators from linear logic, and eks's then form a model for it, but this model is also not complete. However it does give us a logic to reason with eks's, and is useful in constructing a verification language for concurrency. This connection is explored in [Pra93].

References


Specification of Instruction-Level Parallelism

Ed Harcourt*       Jon Mauney*       Todd Cook†

*Department of Computer Science
†Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695
July 15, 1993

Abstract

We present a technique for formally describing, at a high-level, the timing properties of Superscalar/RISC instruction set processors. We illustrate the technique by specifying a hypothetical processor that shares many properties of commercial processors including delayed loads and branches, interlocked floating-point instructions, and multiple instruction issue (Superscalar). As our mathematical formalism we use SCCS, a synchronous process algebra used for specifying timed concurrent systems. Timing properties are specified at an abstract level without resorting to implementation detail. Such high-level specifications are useful for timing-level simulator generation, synthesis and verification of hardware and software (e.g. compilers, schedulers), and precise documentation. We have implemented our specification within the framework of the Concurrency Workbench, a tool for simulating and analyzing SCCS specifications.

1 Introduction

In modern computer architecture the temporal and concurrent properties of the instructions are often visible to the user of the processor. Consequently, such properties should be included in any behavioral architecture specification. We present a technique for formally describing, at a high-level, the timing properties of Superscalar/RISC instruction set processors [PH90, Joh91]. We illustrate the technique by specifying a hypothetical RISC that shares many properties of
commercial RISCs including delayed loads and branches, interlocked floating-point instructions, and multiple instruction issue (Superscalar).

As our mathematical formalism we use SCCS, a synchronous process algebra designed for specifying timed concurrent systems [Mil89, Mil83]. SCCS allows us to explicitly specify the temporal and concurrent properties of a processor (and instructions). In contrast, a functional approach only allows us to specify final computations [Pai90]. We have implemented our specification on the Concurrency Workbench [CPS93, Mol92] allowing us to interactively experiment with, analyze, and simulate our processor description. This research is in conjunction with research to design a specification language for instruction set architecture [CFHM93].

1.1 Levels of Abstraction

There are many views of an instruction set processor — a common hierarchy is:

- The architecture level is a functional view that represents the processor as seen by the assembly language programmer.
- The organization level includes the general structure of the processor in terms of functional units (e.g., integer and floating-point pipelines, caches, and busses).
- The logic level contains the low level implementation detail of the functional units.

The user of a processor is concerned with the architectural level, as they must have information to write correct programs. However the user would also like to use the processor most efficiently. For example, in some RISC architectures the following instruction sequence may possibly be programmed more efficiently.

\[
\begin{align*}
(1) & \quad \text{Load R1, (R2)} \quad ; R1 \leftarrow \text{Mem}[R2] \\
(2) & \quad \text{Add R2, R2, R1} \quad ; R2 \leftarrow R2 + R1 \\
(3) & \quad \text{Add R3, R3, #1} \quad ; R3 \leftarrow R3 + 1
\end{align*}
\]

Instruction (2) will usually cause an interlock (on the MIPS this is an incorrect program) which wastes cycles. However, instructions (2) and (3) may be switched without altering the meaning of the program. This switch would most likely eliminate the interlock caused by (2).
There is no hard line that determines where one processor view ends and another begins. Usually the architecture level does not contain timing information and the organization level does. But the organization level also contains a considerable amount of other detail that is of no concern to the user.

A motivating example comes from the MIPS manual which states that for the load-word instruction, \texttt{LW rt, offset(base)}, "\textit{... the contents of general register \texttt{rt} are undefined for time $T$ of the instruction immediately following this load instruction}" [KH92]. Such informal descriptions are vague and imprecise and demonstrate the kind of timing constraint we wish to formalize at an abstract level — that is, hides organization detail. There may be any number of reasons for the delay in the load instruction and we only wish to specify the delay and not the underlying cause. The user should not be expected to infer the delay by studying low-level organization.

The goal of this research, then, is to develop a mathematical model of instruction timing at an abstract level that hides irrelevant detail of organization.

1.2 Extensions to SCCS

It is assumed that the reader is familiar with SCCS as presented in [Mil83, Mil89]. We use two extensions to SCCS that will aid us in writing processor specifications.

Frequently we wish to execute two agents $A$ and $B$ in parallel where $B$ begins executing one clock cycle after $A$ (\textit{e.g.}, issuing instructions on consecutive cycles). This is modeled by the agent $A \times 1 : B$ and the expression $A \ \text{Nezt} \ B$ denotes this agent.

Another useful operator is the \textit{priority sum} operator, $\triangleright$ [CW91]. If in $A \triangleright B$ both $A$ and $B$ can execute then $A$ is preferred.

2 Specifying a Processor

A processor is a system of interacting processes where registers and memory interact with one or more functional units. Equation 1 represents such a system at the highest level.

\[
\text{Processor} \overset{\text{def}}{=} (\text{Instruction Unit} \times \text{Memory} \times \text{Registers}) \uparrow I
\]

where $I = \{\text{Instructions from section 2.A.}\}$
Before we proceed in specifying instructions and their interaction it is necessary to develop an appropriate model of registers and memory.

2.1 Defining the Registers

In this section we develop an abstract model of storage in which storage cells are modeled as agents. The agent \( RegI(y) \) defines one register holding a value \( y \), such that an action \( \text{putr}(z) \) executed at time \( t \) stores \( z \) in the register which is available for use at time \( t + 1 \). The action \( \text{getr}(y) \) retrieves the value stored in the register and assigns this to \( y \). Another action, a product of two particulate actions, \( \text{putr}(z)\text{getr}(y) \), allows \( RegI \) to be read and written simultaneously. The value read is the old value in \( RegI \) not the new one being written. Since reading a register does not alter its value, multiple \( \text{getr} \) actions are allowed on the same register. The action \( \text{getr}(y)\text{getr}(y) \) represents reading the register twice, which we abbreviate to \( \text{getr}(y)^2 \).

\[
\text{RegI}(y) \stackrel{\text{def}}{=} \sum_{j \in \{1,2\}} \text{getr}(y)^j : \text{Reg}(y) + \sum_{j \in \{0,1,2\}} \text{getr}(y)^j \text{putr}(z) : \text{Reg}(z) + 1 : \text{Reg}(y) \quad (2)
\]

2.1.1 Register Locking

The actions \( \text{getr} \) and \( \text{putr} \) are atomic. It may be that a register is going to be updated some time in the future (e.g., delayed loads) and any attempt to read or write the register by another agent should result in an error. We augment equation 2 by allowing an agent to reserve a register for future writing using the action \( \text{lockreg} \) and then, at some point in the future, by writing the register (with \( \text{putr} \)) and releasing it with the action \( \text{releasereg} \). When an agent locks a register the register goes into a state \( \text{Locked\_Reg} \) where the only allowable action is \( \text{putr}(z)\text{releasereg} \). All other combinations of \( \text{getr} \) and \( \text{putr} \) in the locked state lead to the inactive agent 0. This need to trap all of the other illegal action sequences complicates matters so we have factored them into equation 4.

\[
\text{Reg}(y) \stackrel{\text{def}}{=} \text{RegI}(y) + \text{lockreg} : \text{Locked\_Reg}(y)
\]

\[
\text{Locked\_Reg}(y) \stackrel{\text{def}}{=} \text{Illegal\_Access}(y) + \text{putr}(z)\text{releasereg} : \text{Reg}(z)
+ 1 : \text{Locked\_Reg}(y)
\quad (3)
\]
\[ \text{Illegal \ Access}(y) \overset{\text{def}}{=} \sum_{j \in \{1,2\}} \text{getr}(y)^j : 0 + \sum_{j \in \{0,1,2\}} \text{getr}(y)^j \text{putr}(x) : 0 + \sum_{j \in \{1,2\}} \text{getr}(y)^j \text{putr}(x) \text{releasereg} : 0 \] (4)

Given the definition of one register a family of registers \((\text{Reg}_1, \text{Reg}_2, \text{etc.})\) is now defined by subscripting each of the actions by a register number. For example, the action putr \(_i\) represents writing register \(i\). Thirty-two registers are constructed by

\[ \text{Registers} \overset{\text{def}}{=} \prod_{0}^{31} \text{Reg}_i(y) \] (5)

Notice that when no putr or getr action is requested the registers are idling.

2.2 Defining Memory

Given the definition of the register \(\text{Reg}_1\) (non-locking version), a similar definition of an agent \(\text{Memory}\) is straightforward. Analogously, actions getm and putm read and write memory cells and the agent \(\text{Memory}\) is defined to be a product of individual memory cells.

2.3 Instruction Pipeline

Instruction pipelines are usually described in terms of its stages of execution, for example: fetch, decode, execute, memory access, write back (abbreviated IF, ID, EX, MEM, WB). \(\text{IPL}\) (for instruction pipeline) defines a model of an instruction pipeline.

\[ \text{IPL} \overset{\text{def}}{=} \text{IF} \times \text{ID} \times \text{EX} \times \text{MEM} \times \text{WB} \]

This is a reasonable and obvious representation, but if we are interested only in \(\text{external behavior}\) it is over specified. We should resist attempting to specify an architecture’s timing behavior in terms of individual stages as this commits us to describe the functionality of each individual stage which would have to include, for example, forwarding hardware and latches. We should strive for a more abstract specification.
2.4 ToyP, a Toy Processor

To construct a specification of a processor we present the instructions of a hypothetical RISC, ToyP, that shares many features of commercial RISCs. ToyP is loosely based on the MIPS architecture [KH92]. ToyP instructions, memory word size, registers, and addresses are thirty two bits. ToyP is a Load/Store architecture with three-operand arithmetic instructions.

Here is an informal description of the semantics and timing behavior of some ToyP instructions.

- **Add R\textsubscript{i}, R\textsubscript{j}, R\textsubscript{k}** adds registers \( j \) and \( k \) and puts the result in register \( i \). The instruction executing immediately after an Add may use register \( i \).

- **Load R\textsubscript{i}, R\textsubscript{j}, #Const** is a delayed load instruction. Register \( i \) is being loaded from memory at the base address in register \( j \) with offset #Const. The instruction executing immediately after Load cannot use register \( i \).

- **BZ R\textsubscript{i}, #Locn** is a delayed branch instruction. The instruction immediately after the branch is always executed before the branch is taken (if \( R\textsubscript{i} = 0 \)). If the branch is not taken then instruction after the branch is not executed. Another BZ instruction may not appear in the branch delay slot.

- **Fadd FR\textsubscript{i}, FR\textsubscript{j}, FR\textsubscript{k}** is an interlocked floating-point add that takes six cycles before the result can be used. If another Fadd instruction tries to use the result before the current Fadd is finished then instruction execution stalls until the result is ready.

2.5 Instruction Issue

Given our definitions of *Registers* and *Memory* we now describe an agent \( \text{Instr}(PC) \) (equation 6) that specifies the behavior of ToyP instructions off of program counter \( PC \). Instructions are divided into four classes: arithmetic, load and store, branch, and floating-point and are described by agents *Alu, Load_Store, Branch*, and *Float*.

\[
\text{Instr}(PC) \overset{\text{def}}{=} (\text{Non\_Branch}(PC) \text{ Next } \text{Instr}(PC + 4)) + \text{Branch}(PC) \triangleright \text{Stall}(PC) \tag{6}
\]

\[
\text{Non\_Branch}(PC) \overset{\text{def}}{=} \text{Alu}(PC) + \text{Load\_Store}(PC) + \text{Float}(PC) \tag{7}
\]

\[
\text{Stall}(PC) \overset{\text{def}}{=} 1 : \text{Instr}(PC) \tag{8}
\]
There are three possible execution paths of $Instr(\text{PC})$.

- A non-branch instruction executes and the next instruction to execute is at $\text{PC} + 4$.

- A branch instruction may execute. Here, the decision on what instruction to execute next is deferred.

- If no instruction can execute then the processor must stall. The $\triangleright$ operator (section 1.2) is used here because the processor should stall only when no other alternative is available.

2.5.1 Arithmetic Instructions

All Von Neumann architectures are based on the "stored program model" and fetch instructions from memory using a program counter which we call, $\text{PC}$. The action

$$\text{get}_{\text{PC}}(\text{Add } R_i, R_j, R_k)$$

represents fetching an Add instruction from memory. And in fact, from a user's view, an instruction $\text{Add } R_i, R_j, R_k$ appears to take one cycle to execute. In the following instruction sequence,

$$\begin{align*}
\text{Add } R_1, R_2, R_3 \\
\text{Mov } R_2, R_1
\end{align*}$$

the Add instruction executes at time $t$ and the Mov executes at time $t + 1$. From a behavioral view there is no problem with writing $R_1$ and reading $R_1$ in consecutive instructions. The user does not and should not need to understand bypass hardware in order to discover that the above instruction sequence is legal.

The agent

$$Alu(PC) \overset{def}{=} \text{get}_{\text{PC}}(\text{Add } R_i, R_j, R_k)\text{get}_{R_j}(x)\text{get}_{R_k}(y)\text{put}_{R_i}(x + y) : DONE$$

represents the execution of the Add instruction specifying that registers are accessed and the result is written atomically. The agent DONE is the idle agent and has the effect of representing termination of the instruction. (In [Mil83, Mil89] the idle agent is called 1. We use DONE to avoid confusion with the idle action 1.)
2.5.2 Load and Store Instructions

The following instruction sequence,

\[
\text{Load } R1, R2, \#8 \\
\text{Mov } R3, R1
\]

is illegal in ToyP because of the use of \( R1 \) immediately after the Load. The Load instruction accesses memory at time \( t \) and the result of the load is available at time \( t + 2 \). This is represented by,

\[
\text{Load}_\text{Store}(PC) \overset{\text{def}}{=} \text{get}_{\text{PC}}(\text{Load}_i, R_j, \Delta)\text{get}_{r}(B)\text{get}_{m_i}\text{lockreg}_i: \text{put}_{r}(V)\text{releasereg}_i: \text{DONE}
\]

The Store instruction is similarly defined except that the result is ready immediately (presumably because of forwarding hardware).

2.5.3 The Branch Instruction

Equation 9 specifies the behavior of the delayed branch instruction, BZ.

\[
\text{Branch}(PC) \overset{\text{def}}{=} \text{get}_{\text{PC}}(\text{BZ}_i, \text{Locn})\text{get}_{r}(V):
\]

\[
\text{if } V = 0 \text{ then}
\]

\[
\text{Non}_\text{Branch}(PC + 4) \text{ Next } \text{Instr}(\text{Locn})
\]

\[
+ \text{get}_{m_{PC+4}}(\text{BZ}_i, \text{Locn}) : 0
\]

\[
\text{else}
\]

\[
\text{Instr}(PC + 8)
\]

(9)

The BZ instruction has the effect that

- at time \( t \), a BZ instruction is fetched and register \( R_i \) is accessed.

- at time \( t + 1 \), if the value of \( R_i \) is not zero then execution continues with the instruction after the branch delay slot.

- at time \( t + 1 \), if the value of \( R_i \) is zero then a non-branch instruction is executed in the branch delay slot and execution continues with the instruction at \( \text{Locn} \) at time \( t + 2 \).
• If another BZ instruction is in the delay slot then we reach the inactive agent 0, which represents an error state.

2.6 Interlocked Floating-Point Instructions

The floating-point add instruction Fadd takes six cycles to compute its result. For instructions that have a large latency it is generally unreasonable to expect the programmer (or scheduler) to find enough independent instructions to execute until the Fadd is complete. Inserting Nop instructions would significantly increase code size, therefore, floating-point instructions are typically interlocked.

2.6.1 Floating-Point Registers

We associate a lock with each floating-point register as in the integer registers. The difference though is that an attempt to read or write an integer register while it is locked is illegal while reading or writing a floating-point register while it is locked causes the processor to stall. ToyP has a separate set of thirty two floating-point registers that are defined similarly to the integer registers except that we add two new actions, lockfreg and releasefreg. Actions putfr and getfr are the two actions that write and read a floating-point register.

\[
F_{reg}(y) \xrightarrow{\text{def}} \text{lockfreg}_i : \text{Write}_i + \sum_{j \in \{1,2\}} \text{getfr}_i(y) \xrightarrow{\text{def}} F_{reg}(y) + 1 : F_{reg}(y)
\]

\[
\text{Write}_i \xrightarrow{\text{def}} \text{putfr}_i(z) \text{releasefreg}_i : F_{reg}(z) + 1 : \text{Write}_i
\]

2.6.2 The Fadd instruction

Having defined interlocked fp-registers we can specify the behavior of the Fadd instruction. The Fadd instruction must (1) access its source registers and lock its destination register using the action lockreg; (2) compute the addition; (3) write the result in the destination register and release the destination register using the action releasereg. Equation 10 specifies ToyP’s Fadd instruction.

\[
\text{Float}(PC) \xrightarrow{\text{def}} \text{getmp}_{PC}(\text{Fadd, FR}_i, \text{FR}_j, \text{FR}_k) \text{lockfreg}_i \text{getfr}_j(z) \text{getfr}_k(y) : \text{(1:)putfr}_i(z + y) \text{releasefreg}_i : \text{DONE}
\]

The processor stalls when an instruction wishes to access a locked fp-register. Since the action will not be available the execution of the instruction is suppressed and the only available action
then is to execute the idle action of agent \textit{Stall}.

3 A Two-Issue Superscalar ToyP

This section describes variations of ToyP that can issue and execute multiple instructions per cycle. Such \textit{multiple issue} processors are commonly referred to as \textit{superscalar} processors. Two or more instructions can be executed in parallel if they are data independent and can also be issued to separate functional units.

 Usually, one floating-point and one integer instruction can be issued in parallel as they use separate functional units. Also, if they use disjoint register files, they are guaranteed to be data independent. Two integer instructions can be issued in parallel only if there are two or more integer functional units. In this case, the problem is complicated somewhat because data dependencies between two integer instructions can arise inhibiting their parallel execution.

3.1 A Float \times Integer Superscalar

In this variant of ToyP one integer and one floating point instruction can be issued in parallel. Branch and Load/Store instructions must be issued sequentially.

If two instructions can be issued in parallel then it must be an integer instruction followed by a floating point instruction or vice-versa.

\[
(Float(PC) \times Alu(PC + 4)) + (Alu(PC) \times Float(PC + 4))
\]

We can rewrite this using summation and also to include continuing execution at \(PC + 8\).

\[
Do\_Two(PC) \overset{\text{def}}{=} \left( \sum_{i,j \in \{0,4\}} (Alu(PC + i) \times Float(PC + j)) \right) \text{Next } Instr(PC + 8)
\]

\(Do\_Two\) assumes, justifiably, that an instruction cannot be both an integer and a floating-point instruction. This implies that when \(i = j\) the agent \(Alu(PC + i) \times Float(PC + j)\) reduces to zero. There are no data dependencies to worry about because each instruction accesses separate register files.
3.1.1 Instruction Issue

Our top-level instruction issue equation 6 must be modified to reflect this new two-issue capability. Renaming equation 6 from Instr to Do_One our processor can now execute two, one, or zero (i.e., stall) instruction(s) per cycle which we capture in equation 11.

\[ \text{Instr}(PC) \overset{\text{def}}{=} \text{Do_Two}(PC) \triangleright \text{Do_One}(PC) \triangleright \text{Stall}(PC) \]  

(11)

Notice the use of \( \triangleright \) (section 1.2) instead of +. Whenever it is possible to do Do_Two it is also possible to do Do_One and issuing two instructions should take priority over issuing one, when possible.

3.2 An Integer \( \times \) Integer Superscalar

In this section we specify a version of ToyP that can execute two integer ALU instructions in parallel. At first glance, it would seem that

\[ \text{Alu}(PC) \times \text{Alu}(PC + 4) \]  

(12)

specifies the ability to execute two integer instructions in parallel. However, because both instructions use the same register file we now have the possibility of data hazards existing between the two integer instructions. Hence, sometimes parallel execution is thwarted.

Using particle restriction (\( A \setminus S \) where \( S \) is a set of particles) we can force equation 12 to apply only to legal integer instruction sequences of length two. In a legal instruction sequence the first integer instruction can write register \( i \) and the second integer instruction cannot write or read register \( i \).

\[ \sum_{i=0}^{31} (\text{Alu}(PC) \setminus A \times \text{Alu}(PC + 4) \setminus B) \]

where \( A = \{\text{putr}_i, \text{getr}_0 \ldots \text{getr}_{31}\} \)

\( B = \{\text{getr}_0 \ldots \text{getr}_{31}, \text{putr}_0 \ldots \text{putr}_{31}\} - \{\text{putr}_i, \text{getr}_i\} \)  

(13)

Equation 13 represents all of the allowable integer instruction sequences of length two that may execute in parallel.
4 Simulation

A simulation of our ToyP specification amounts to running an agent that represents a ToyP program with our agent that represents ToyP. That is, ToyP × Program. We observe the behavior of the program by calculating the transition graph of an agent. We do not have room to reproduce a transition graph here. We only note that our simulation takes place within the framework of the Concurrency Workbench which allows us to experiment with our processor specification.

5 Conclusions and Comments

In this paper we have presented a technique for specifying the timing properties of instruction-level parallel processors using SCCS, a synchronous process calculus. The timing properties specified are delayed loads and branches, interlocked floating-point operations, and multiple instruction issue.

With the plethora of formal specification languages (especially for hardware), the question arises why we should prefer our approach to another. The answer, as is often the case, depends on what one is interested in, and is, in our case, explicit specification of both timing and concurrency. The reason that this is important, is that, if the programmer wants to use the processor the most efficiently, timing and concurrency must be specified. Our SCCS processor description specifies a processor for what it really is; a communicating system of functional units.

In future research, we plan to derive instruction scheduling parameters (e.g., latencies) from processor specifications allowing us to automatically synthesize instruction schedulers. The transition graphs of instructions clearly indicate when instructions begin executing. Latency information is derived by appropriately testing the specification and observing when instructions begin execution.

References


Specification of Transition Systems with Negation
(Extended Abstract)

Rémi Lissajoux
Thomson-CSF/LCR
91404 Orsay Cedex
France

July 2, 1993

Abstract

We define new conditions to specify transitions systems with negation on process algebras, with a SOS a la Plotkin. These conditions lie strictly between those given in [Bloom et al.88] and [Groote91]. They ensure both existence and unicity of a model and congruence of the bisimulation. They are based on a generalization of guardedness of terms (and of recursion) which is defined modulo a set of Structural Transition Rules, and not only with the prefix operator [Vaandrager93]. Therefore, they are applicable to any (free) algebra.

1 Introduction

In the field of concurrent systems theory, some recent work has been devoted to the use of negation in transition systems specifications. Negation is used to specify priority relations between process transitions (see [Groote91, Cleaveland et al.88, Okulicka90] for some examples). This allows the specification of testing processes (contexts) which cannot be expressed in classical process algebras [Bloom et al.88, Bloom et al.92].

Transition systems specifications (TSS) with negation do not “specify” a transition system (TS) in a straightforward way as SOS a la Plotkin does [Plotkin81]. [Groote91] uses syntactical conditions on TSS (stratification) to ensure existence of models without ambiguity. [Bloom et al.88] give similar conditions to their GSOS format.
The format given in [Groote91] is quite general. It applies to any TSS which is a set of (ground) rules of the form:

$$\{t_k \xrightarrow{k} t'_{k} \mid k \in K\} \cup \{t_l \xrightarrow{l} t' \mid l \in L\}$$

$$\xrightarrow{t} t'$$

where $t$ are ground terms of a free algebra on a given signature.

Moreover, stratification is a syntactic condition. But it is very general, and is not simple and direct to apply to most TSS we may encounter.

GSOS format of [Bloom et al.88] applies to algebras of the form:

$$T ::= \xi \mid a.T \mid op(T_1, \ldots, T_n) \mid \text{fix}\xi.T$$

To ensure the existence and unicity of a model of a GSOS, only classical guarded recursion is required. The problem here is that we are restricted to algebras with prefix operator, and that guarded terms are defined with this operator.

2 Existence and unicity of a model

We propose new conditions on TSS which lie strictly between GSOS and stratified TSS. We keep from GSOS the simplicity of the format based on Structural Transition Rules (STR) and the generality of stratified TSS applicable on any free algebra. For this, we generalize the notion of guarded term following [Vaandrager93].

In our formalism, a term is guarded for a set of STR. STR are of the form

$$\{x_k \xrightarrow{k} y_{ki} \mid k \in [1..K], i \in I_k\} \cup \{x_k \xrightarrow{k} l \mid k \in [1..K] l \in L_k\}$$

$$t(x_1, \ldots, x_K) \xrightarrow{t} t'$$

where $x, y_{ki}$ are distinct variables which include variables in $t'$.

For each constructor $f$ in the signature, we call $\text{def}(f)$ the subset of a set $S$ of STR which "defines" $f$:

$$\text{def}(f) = \{r \in S \mid \text{concl}(r) = f(x_1, \ldots, x_{\text{ar}(f)})\}$$

**DEFINITION 1** We say $f$ is guarded in $k$ relatively to a set of STR $\Gamma$ iff for every rules in $\text{def}(f)$ we have (with the notations of (2)) $L_k = I_k = \emptyset$.

Intuitively, if $f$ is guarded in $k$, then transitions of terms $f(t_1, \ldots, t_K)$ do not depend on transitions of $t_k$.

For any free algebra, without recursion, any TSS composed of STR has a trivial stratification based on the size of terms. Therefore, there exists a
unique model. With recursion, size of terms is no longer stratified any more, and we need guarded terms.

To introduce recursion in our framework, we use special constants in the signature (supposed not "defined" by any STR) that we call procedures and a function \( \mathcal{D} \) (a declaration) which maps procedures to terms. To each procedure we associate an operational rule

\[
\frac{D(v) \xrightarrow{\Delta} X}{v \xrightarrow{\Delta} X}
\]

So, a set \( S \) of STR and a declaration \( \mathcal{D} \) define a TSS on a free algebra with recursion.

For a declaration \( \mathcal{D} \) and a set \( S \) of STR, we define \( \leftrightarrow \) the least relation satisfying

1. for any procedure \( v \), \( v \leftrightarrow D(v) \).
2. for any procedure \( v \), for any \( f(t_1, \cdots, t_n) \) such that \( v \leftrightarrow f(t_1, \cdots, t_n) \), if \( f \) is not guarded in \( i \) relatively to \( S \), then \( v \leftrightarrow t_i \).

\( \leftrightarrow \) is the restriction of \( \leftrightarrow \) to procedures.

**Definition 2** A declaration \( \mathcal{D} \) is guarded relatively to a set \( S \) of STR iff \( \leftrightarrow \) is well-founded.

If the sets of STR and procedures are finite, guardedness is decidable. Moreover, we can prove that a set of STR and a declaration guarded relatively to this set define a (strictly) stratified TSS. So, we get the following result (from [Groote91]):

**Theorem 1** Given a set \( S \) of STR and a declaration \( \mathcal{D} \), if \( \mathcal{D} \) is guarded relatively to \( S \), then a model exists and is unique.

### 3 Bisimulation as a congruence

Given a model for TSS, we are confronted by the problem of defining to process equivalence. Moreover, we look for a congruence to ensure that equivalent processes may not be distinguished when plugged in an appropriate context. Bisimulation [Park80] is one of the finest process equivalence, and has nice mathematical properties [Milner89]. Following [Groote91], we show that with the conditions of the theorem 1, (strong) bisimulation is a congruence. We can use [Groote91] results because a set of STR rules and a guarded declaration define clearly a stratified, well-founded, ntyft format TSS.

**Theorem 2** Given a set \( S \) of STR and a declaration \( \mathcal{D} \), if \( \mathcal{D} \) is guarded relatively to \( S \), then, for the unique model, bisimulation is a congruence.
4 Related works

Our formalism is more general than GSOS. For an algebra with prefix operation, we can define a pseudo constructor $\delta_\lambda$ for each label $\lambda$. Informally, for each term $t$, $\delta_\lambda(t)$ is $\lambda.t$. Rule for $\delta$ is

$$\frac{}{\delta_\lambda(X) \xrightarrow{\lambda} X}$$

$\delta_\lambda$ is guarded in 1 to any TSS extended with these rules. It is straightforward to see that guarded recursion in the sense of GSOS implies guarded recursion in our sense.

We are less general than format of [Groote91]. Because of the (necessary) restriction that variables in STR must be distinct, we don’t have copying and lookahead facilities of ntyft/ntyxt TSS. So, we loose the main result that with operators defined in the ntyft/ntyxt, completed trace congruence is bisimulation.

Acknowledgments: I would like to thank referees for pointing me out the work by F.W. Vaandrager.

References


A Comparison of Simulation Techniques and Algebraic Techniques for Verifying Concurrent Systems*

Nancy Lynch and Roberto Segala
MIT- Laboratory for Computer Science

July 15, 1993

Abstract

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

1 Introduction

Simulation-based assertional techniques and process algebraic techniques are two of the major methods that have been proposed for the verification of concurrent and distributed systems. Although the two methods are used for the same task, the proofs that are carried out in the two styles seem to be quite different. Indeed, the two methods have been developed by largely disjoint research communities, using different semantic models. The literature contains many examples of proofs using the two methods: some typical examples of simulation proofs appear in [LT87, LLS93], while examples of algebraic proofs appear in [Bae90, Jos92].

In this paper, we unify, evaluate and compare the simulation-based and process algebraic verification techniques in terms of the input/output automaton (I/O automaton) model of Lynch and Tuttle [LT87]. This framework has been used extensively for the verification of complex algorithms and pieces of distributed systems [WLL88, LS92, LP92, LLS93], and has already been given a process algebraic characterization [Vaa91, Seg92, DS92]. We show how

each of these techniques can be applied to the common task of verifying both safety and liveness properties of systems described as I/O automata. We then use each technique to verify a small but typical delay insensitive circuit taken from [Jos92]: a Muller C element [MB59] implemented in terms of a majority element and a wire. Both the implementation and the specification are described as I/O automata, and the verification consists of showing that the fair preorder relation (i.e., fair trace inclusion) holds between the implementation and the specification automata.

The two proofs proceed very differently. First, the simulation proof uses a forward simulation [LV91] from the implementation to the specification, then invokes an execution correspondence lemma [LLS93] to obtain a correspondence between executions of the implementation and the specification. Then a simple argument about fairness is made, based on the correspondence between executions; this fairness argument uses the convenient notion of a forcing condition for an I/O automaton fairness class. The fairness argument could easily be formalized using a temporal logic of states and actions [Sta84, LLS93], although we do not do this in this paper.

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

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

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

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

2 The Input/Output Automaton Model

We begin with a brief review of the I/O automaton model, which will be used as the basis of the rest of the work in this paper. For a complete account, we refer the reader to [LT87].

Definition 2.1 (Notation for sequences) Given an alphabet \( A \), let \( A^* \) be the set of finite length sequences made of elements of \( A \) and let \( A^\omega \) be the set of infinite length sequences made of elements of \( A \). Finally, let \( A^* \cup A^\omega \) be denoted by \( A^{\infty} \).

Definition 2.2 (I/O automata) An I/O automaton \( A \) consists of five components:

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

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

\[
q \xrightarrow{a_1 \ldots a_n} q' \iff \exists q_0, \ldots, q_n \text{ with } q_0 = q \text{ and } q_n = q' \text{ such that } q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n.
\]

Similarly, for infinite sequences, we write

\[
q \xrightarrow{a_1 a_2 \ldots} \text{ if } \exists (q_i)_{i \in \mathbb{N}} \text{ such that } q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \ldots
\]

Two derived transition relations, abstracting from internal computations, are

\[
q \xrightarrow{a} q' \iff \exists s_1, s_2 \in \text{int}(A) q \xrightarrow{a_1 a_2} q',
\]

\[
q \xrightarrow{a} q' \iff \exists s_1 \in \text{int}(A) q \xrightarrow{a_1 a} q'.
\]

The last two transition relations can be extended to finite and infinite sequences of actions in the same way as for \( \text{steps}(A) \).
Definition 2.3 (Executions and traces) An execution fragment of an I/O automaton $A$ is a (finite or infinite) sequence of alternate states and actions starting with a state and, if the execution fragment is finite, ending in a state

$$\alpha = q_0a_1q_1a_2q_2 \cdots$$

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

The trace of an execution fragment $\alpha$ of an I/O automaton $A$, written $\text{trace}_A(\alpha)$, or just $\text{trace}(\alpha)$ when $A$ is clear, is the list obtained by projecting $\alpha$ onto the set of external actions of $A$, i.e., $\text{trace}(\alpha) = \alpha[\text{ext}(A)]$.\footnote{Our definition of trace coincides with the usual definition of behavior for I/O automata. We have changed the terminology in the interests of consistency with the usual notation of process algebra.} We say that $\beta$ is a trace of an I/O automaton $A$ if there exists an execution $\alpha$ of $A$ with $\text{trace}(\alpha) = \beta$. We denote by $\text{traces}^*(A), \text{traces}^w(A)$ and $\text{traces}(A)$ the sets of finite, infinite and all traces of $A$, respectively.

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

Definition 2.4 (Fair executions) A fair execution fragment of an I/O automaton $A$ is an execution fragment $\alpha \in \text{execs}(A)$ such that for all $X \in \text{part}(A)$

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

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

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

Definition 2.5 (Fair preorder) Given two I/O automata $A$ and $B$ with the same external action signature, the fair preorder is defined as

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

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

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

- $out(\text{Hide}_I(A)) = out(A) \setminus I$, and
- $int(\text{Hide}_I(A)) = int(A) \cup (acts(A) \cap I)$.

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

**Definition 2.7 (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. For the parallel composition we need a notion of compatibility for action signatures.

**Definition 2.8 (Strong compatibility of I/O automata)**

1. A set of action signatures $\{S_i : i \in I\}$ are strongly 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 strongly compatible iff their action signatures are strongly compatible.

**Definition 2.9 (Composition of I/O automata)** The composition \( A = \prod_{i \in I} A_i \) of strongly compatible I/O automata \( \{ A_i : i \in I \} \) is defined to be the I/O automaton with

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

3 The Problem

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

3.1 Informal Description

The example consists of a simple delay insensitive circuit, taken from [Jos92], called the Muller C element [MB59]. Its interface is shown in Figure 1. A Muller C element has two input ports \( a, b \) and one output port \( c \). Once it is in its initial state with all input and output voltage levels low, a Muller C element waits for both its inputs to reach the high voltage level for then raising its output voltage level. It then waits for both its inputs to reach the low voltage level for then reaching again its initial state. In our specification no changes on the input ports are allowed whenever the voltage level of an output port has to change. Real implementations may exhibit unexpected behaviors (such as the glitch phenomenon) in such cases. For the above
reason we do not specify the behavior of any element whenever an output voltage level has to change and an input occurs.

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

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

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

3.2.1 Actions as Voltage Level Transitions

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

3.2.2 Specifications of the Elements

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

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

**Specification 3.1 (Muller C element)** A Muller C element \( C_N \) is defined as follows.

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

The transition relation is defined by the following table:

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

**Specification 3.2 (Wire)** A wire $W_N$ is defined as follows.

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

The transition relation is defined by the following table:

<table>
<thead>
<tr>
<th></th>
<th>$m$</th>
<th>$c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>$m$</td>
<td>NE</td>
</tr>
<tr>
<td>$m$</td>
<td>NS</td>
<td>$\lambda$</td>
</tr>
</tbody>
</table>

**Specification 3.3 (Majority element)** A majority element $M_N$ is defined as follows.

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

The transition relation is defined by the following table:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>${a}$</td>
<td>${b}$</td>
<td>${c}$</td>
<td>NE</td>
</tr>
<tr>
<td>${a}$</td>
<td>$\emptyset$</td>
<td>${a, b}$</td>
<td>${a, c}$</td>
<td>NE</td>
</tr>
<tr>
<td>${b}$</td>
<td>${a, b}$</td>
<td>$\emptyset$</td>
<td>${b, c}$</td>
<td>NE</td>
</tr>
<tr>
<td>${c}$</td>
<td>${a, c}$</td>
<td>${b, c}$</td>
<td>$\emptyset$</td>
<td>NE</td>
</tr>
<tr>
<td>${a, b}$</td>
<td>NS</td>
<td>NS</td>
<td>${a, b, c}$</td>
<td>${c}$</td>
</tr>
<tr>
<td>${a, c}$</td>
<td>NS</td>
<td>${a, b, c}$</td>
<td>NS</td>
<td>${b}$</td>
</tr>
<tr>
<td>${b, c}$</td>
<td>${a, b, c}$</td>
<td>NS</td>
<td>NS</td>
<td>${a}$</td>
</tr>
<tr>
<td>${a, b, c}$</td>
<td>${b, c}$</td>
<td>${a, c}$</td>
<td>${a, b}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
3.2.3 From Specifications to I/O Automata

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

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

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

The following proposition shows that everything is possible whenever \( \Omega \) is reached, i.e., any choice of implementation is correct whenever the specification reaches state \( \Omega \).

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

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

Now we can state the problem formally: verify that

\[
\text{Hide}_{\{m\}}(\mathcal{A}(M_N)\|\mathcal{A}(W_N)) \sqsubseteq_F \mathcal{A}(C_N).
\]
4 A Verification using Simulation

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

4.1 The Theory

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

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

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

4.1.1 Forward Simulations and the Execution Correspondence Lemma

The notion of forward simulation that we use is taken from the comprehensive paper by Lynch and Vaandrager [LV91].

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

1. If $q \in \text{start}(A)$ then $f[q] \cap \text{start}(B) \neq \emptyset$.

2. If $q \xrightarrow{a} q'$ and $p \in f[q]$, then there exists a state $p' \in f[q']$ such that $p \xrightarrow{a_{\text{ext}}(B)} p'$.

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

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

However, since we would like to base our proof of fair trace inclusion on our proof of trace inclusion, it is useful to have a stronger consequence of the existence of a forward simulation. This lemma is proved in [LLS93].

\[\text{[LLS93]}\]

In [LLS93], it is also shown that a similar lemma holds for other types of simulation relations such as backward simulations.
Lemma 4.3 (Execution correspondence) Let $f$ be a forward simulation from an I/O automaton $A$ to an I/O automaton $B$. Then, for each execution $\alpha = q_0a_1q_1a_2q_2\cdots$ of $A$ there is an execution $\alpha' = q'_0b_1q'_1b_2q'_2\cdots$ of $B$ and a total monotone nondecreasing mapping $c : \{0, \ldots, |\alpha|\} \to \{0, \ldots, |\alpha'|\}$ such that

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

If the forward simulation is well chosen, Proposition 4.3 can be used as the basis of a proof of fair trace inclusion, as follows. For each fair execution $\alpha$ of $A$, first produce a corresponding execution $\alpha'$ of $B$. Then show that the fairness of $\alpha$ implies the fairness of any corresponding execution of $B$. This is the general strategy we will follow in our proof.

4.1.2 Forcing I/O Automata

In carrying out the proof of fairness, it turns out to be notationally convenient to use a slight generalization of I/O automata that we call forcing I/O automata [L.L.S93]. The generalization consists of associating a set of states called a forcing set with each class of $\text{part}(A)$. Forcing I/O automata are no more expressive than ordinary I/O automata, in terms of the sets of fair traces they can represent; they are useful, however, because they sometimes admit more concise representations.

Definition 4.4 (Forcing I/O automata) A forcing I/O automaton $A$ is an I/O automaton with the following additional structure:

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

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

Definition 4.5 (Fair executions) A fair execution fragment of a forcing I/O automaton $A$ is an execution fragment $\alpha \in \text{execs}(A)$ such that for all $X \in \text{part}(A)$
• If $\alpha$ is finite then the final state of $\alpha$ is not in the forcing set of $X$.

• If $\alpha$ is infinite then either actions from $X$ appear infinitely often in $\alpha$ or states not in the forcing set of $X$ appear infinitely often in $\alpha$.

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

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

**Proposition 4.6** *Given a forcing I/O automaton $A$, consider an I/O automaton $F(A)$ where*

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

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

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

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

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

1. $F(\text{Hide}_1(A))$ and $\text{Hide}_1(F(A))$ are the same I/O automaton;

2. $F(A\parallel B)$ and $F(A)\parallel F(B)$ are the same I/O automaton.***
4.2 Specification of the Components

In Section 3.2 we described the system components using a "neutral" formalism that is not biased toward either verification method. Each of the two methods, however, has its own characteristic language for describing system components. In this section, we represent each element of Section 3.2 using a variant of the precondition-effect language of [LT87] that is suitable for describing forcing I/O automata. We also relate the new specifications to the neutral ones.

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

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

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

**Specification 4.9 (Muller C element)** A Muller C element $C_F$ is defined as follows.

$S = \{\{a, b\}, \{c\}, \emptyset\}$

$Q = \{\emptyset, \{a\}, \{b\}, \{a, b\}, \Omega\}$

$Q_0 = \{\emptyset\}$

$P = \{\{c\}\}$ where $\{c\}$ has forcing set $\{\{a, b\}\}$

**Transitions:**

**Action $a$**

**Precondition:** $q \neq \{a, b\}$

**Effect:** $q' := q \oplus \{a\}$

**Action $b$**

**Precondition:** $q \neq \{a, b\}$
Effect: \[ q' := q \oplus \{ b \} \]

Action \( c \)
- **Precondition:** \( q = \{ a, b \} \)
- **Effect:** \[ q' := \emptyset \]

**Specification 4.10 (Majority element)** A majority element \( M_F \) is defined as follows.

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

**Transitions:**

Action \( a \)
- **Precondition:** \( q \not\in \{ a, b \}, \{ a, c \} \)
- **Effect:** \[ q' := q \oplus \{ a \} \]

Action \( b \)
- **Precondition:** \( q \not\in \{ a, b \}, \{ b, c \} \)
- **Effect:** \[ q' := q \oplus \{ b \} \]

Action \( c \)
- **Precondition:** \( q \not\in \{ a, c \}, \{ b, c \} \)
- **Effect:** \[ q' := q \oplus \{ c \} \]

Action \( m \)
- **Precondition:** \( |q| \geq 2 \)
- **Effect:** \[ q' := \{ a, b, c \} \setminus q \]

**Specification 4.11 (Wire)** A wire \( W_F \) is defined as follows.

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

**Transitions:**

Action \( m \)
- **Precondition:** \( q = \lambda \)
Effect: \[ q' := m \]

Action \( c \)

**Precondition:** \( q = m \)

**Effect:** \[ q' := \lambda \]

Proposition 4.12

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

4.3 The Verification

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

**Proposition 4.13** \( \text{Hide}_{\{m\}}(M_F||W_F) \subseteq_F C_F \), i.e., a Muller C element can be implemented by a majority element and a wire.

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

More precisely, the mapping \( f \) to use is the following:

\[
\begin{align*}
(\emptyset, \lambda) &\mapsto \{\emptyset\} \\
\{(a), \lambda\} &\mapsto \{(a), \Omega\} \\
\{(b), \lambda\} &\mapsto \{(b), \Omega\} \\
\{(a, b), \lambda\} &\mapsto \{(a, b), \Omega\} \\
\{(c), m\} &\mapsto \{(a, b), \Omega\} \\
\text{all other pairs} &\mapsto \{\omega\}
\end{align*}
\]

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

**Action a:** We distinguish the following cases:
• If $a$ occurs from $(x, \lambda)$ where $x \in \emptyset, \{a\}, \{b\}$ then $(x, \lambda) \xrightarrow{a} (x \oplus \{a\}, \lambda)$ and $x \xrightarrow{a} x \oplus \{a\}$.

• If $a$ occurs from $(\{a, b\}, \lambda)$ then $(\{a, b\}, \lambda) \xrightarrow{a} (\Omega, \lambda)$. Moreover $\{a, b\} \xrightarrow{a} \Omega$ and $\Omega \xrightarrow{a} \Omega$.

• If $a$ occurs from $(\{c\}, m)$ then $(\{c\}, m) \xrightarrow{a} (\Omega, m)$. Moreover $\{a, b\} \xrightarrow{a} \Omega$ and $\Omega \xrightarrow{a} \Omega$.

• If $a$ occurs from any state $(x, \lambda)$ where $x \not\in \emptyset, \{a\}, \{b\}, \{a, b\}$ then $(x, \lambda) \xrightarrow{a} (x', \lambda)$ and $x' \not\in \emptyset, \{a\}, \{b\}, \{a, b\}$. Moreover $\Omega \xrightarrow{a} \Omega$.

• If $a$ occurs from any state $(x, m)$ where $x \neq \{c\}$ then $(x, m) \xrightarrow{a} (x', m)$ and $\Omega \xrightarrow{a} \Omega$. Note that, since for $x = \{a, c\}$ we have $x' = \{c\}$, we need $\Omega$ in the mapping for $(\{c\}, m)$.

**Action $b$:** This case is the same as the case for action $a$.

**Action $c$:** This action is enabled only from states of the form $(x, m)$ and yields a new state $(x', \lambda)$. If $x = \{c\}$ then $x' = \emptyset$ and $\{a, b\} \xrightarrow{a} \emptyset$. In all other cases $x'$ can be anything but $\emptyset$. This is the case for which we need to map $(\{a\}, \lambda), (\{b\}, \lambda)$ and $(\{a, b\}, \lambda)$ to $\Omega$.

**Action $m$:** This action is enabled from each state $(x, \lambda)$ and $(x, m)$ with $|x| \leq 2$. If the starting state is $(x, m)$ then the final state is $(x', \Omega)$. Moreover both starting and final states are mapped to $\Omega$. If the starting state is $(x, \lambda)$ with $x \neq \{a, b\}$ then the final state is $(x', m)$ and both starting and final states are mapped to $\Omega$. If the starting state is $(\{a, b\}, \lambda)$ then $(\{a, b\}, \lambda) \xrightarrow{m} (\{c\}, m)$ and both starting and final states are mapped to $\{a, b\}$ and $\{a\}$.

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

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

Note that the fairness part of the proof above is done somewhat less formally than the simulation part; the fairness part can be formalized using a temporal logic of states and actions [Sta84, LLS93].

Now we can give the main result:
Theorem 4.14 \( \text{Hide}_{(m)}(A(M_N)\|A(W_N)) \subseteq_F A(C_N) \).

**Proof.** From Propositions 4.13 and 4.6 we derive \( F(\text{Hide}_{(m)}(M_F\|W_F)) \subseteq_F F(C_F) \). From Proposition 4.8 we derive \( \text{Hide}_{(m)}(F(M_F)\|F(W_F)) \subseteq_F F(C_F) \). From Proposition 4.12 we obtain \( \text{Hide}_{(m)}(A(M_N)\|A(W_N)) \subseteq_F A(C_N) \).

5 A Verification using Process Algebras

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

5.1 The Theory

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

Below, we define quiescence and relate it to fairness. We then define DIOA ("Demonic I/O Automata"), a process algebra for proving quiescent trace inclusion relationships between I/O automata. ³

5.1.1 From the Quiescent Preorder to the Fair Preorder

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

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

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

³The adjective "demonic" is pictorially used in [Seg92] to emphasize the fact that demonic I/O automata behave catastrophically in the presence of unexpected inputs. It is in contraposition to the approach of [Vaa91] which is called "angelic" in [Seg92].
The quiescent preorder was first introduced in [Vaa91] and is an attempt at approximating the fair preorder by only looking at the finite executions of an I/O automaton. As pointed out through some examples in [Seg92], the quiescent preorder is not an intuitively reasonable notion of implementation in general, however [Seg93] gives some sufficient conditions for the quiescent preorder to coincide with the fair preorder. Below we present some of the results of [Seg93]. We start with some definitions.

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

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

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

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

**Definition 5.5 (Finite internal nondeterminism)** An I/O automaton $A$ has finite internal nondeterminism (FIN) if $\forall h \in \text{acts}^*(A) \{ q | \exists q_n \in \text{start}(A) q_0 \xrightarrow{h} q \}$ is finite.

The above definition of FIN is weaker than the definition given in [LV91]. The definition of [LV91] requires, for every trace $h$, the set of reachable states with $h$ to be finite. In our definition we only require a smaller set to be finite, i.e., the set of states reachable through $h$ with its last external transition.

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

An infinite fair trace made of input actions only can be obtained from an execution containing infinitely many internal transitions. The quiescent preorder, on the other hand, can detect only quiescent states.
Theorem 5.7 (Relationship between the quiescent and fair preorder) Given two I/O automata $A_1, A_2$ with the same external action signature such that $\text{part}(A_1) = \{\text{local}(A_1)\}$ and $\text{part}(A_2) = \{\text{local}(A_2)\}$, if $A_1$ is quiescent detectable and input quiescent detectable, and $A_2$ is fair continuous and has FIN, then

$$A_1 \subseteq_Q A_2 \implies A_1 \subseteq_F A_2.$$ 

If $A_2$ is quiescent detectable then

$$A_1 \subseteq_F A_2 \implies A_1 \subseteq_Q T_2.$$ 

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

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

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

If an I/O automaton $A$ with multiple classes implements an I/O automaton $B$ with a single class, and if the involved automata satisfy the conditions of Theorem 5.7, then the proposition above gives a sufficient condition for deriving the full fair preorder from the quiescent preorder. In fact, from $A' \subseteq_Q B$, where $A'$ is the I/O automaton $A$ with a single class, we derive $A' \subseteq_F B$, and, from Proposition 5.8, we derive $A \subseteq_F B$. Examples of systems satisfying the condition of Proposition 5.8 are the monotone I/O automata of [Sta90], which can model a large class of dataflow networks, and the semi-modular, speed-independent circuits of [MB59]. Our problem is based on delay insensitive circuits.
<table>
<thead>
<tr>
<th>Name</th>
<th>Op.</th>
<th>Domain</th>
<th>Range</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>quiescent</td>
<td>nil$\mathcal{S}$</td>
<td>$\lambda$</td>
<td>$\mathcal{S}$</td>
<td></td>
</tr>
<tr>
<td>omega</td>
<td>$\Omega_{\mathcal{S}}$</td>
<td>$\lambda$</td>
<td>$\mathcal{S}$</td>
<td></td>
</tr>
<tr>
<td>prefixing</td>
<td>$a_{\mathcal{S}}$</td>
<td>$\mathcal{S}$</td>
<td>$\mathcal{S}$</td>
<td>$a \in ext(\mathcal{S})$</td>
</tr>
<tr>
<td>choice</td>
<td>$\oplus_{\mathcal{S}}$</td>
<td>$\mathcal{S}, \mathcal{S}$</td>
<td>$\mathcal{S}$</td>
<td></td>
</tr>
<tr>
<td>choice</td>
<td>$I,J +$</td>
<td>$\mathcal{S}, \mathcal{S}$</td>
<td>$\mathcal{S}$</td>
<td>$I, J \subseteq in(\mathcal{S})$</td>
</tr>
<tr>
<td>parallel</td>
<td>$s_1</td>
<td></td>
<td>s_2$</td>
<td>$s_1, s_2$</td>
</tr>
<tr>
<td>hiding</td>
<td>$\tau_f^S$</td>
<td>$\mathcal{S}$</td>
<td>$\mathcal{S'}$</td>
<td>$I \subseteq \text{out}(\mathcal{S}), \mathcal{S'} = (\text{in}(\mathcal{S}), \text{out}(\mathcal{S}) \setminus I, \text{int}(\mathcal{S}) \cup I)$</td>
</tr>
<tr>
<td>renaming</td>
<td>$\rho_{\mathcal{S}}$</td>
<td>$\mathcal{S}$</td>
<td>$\mathcal{S'}$</td>
<td>for each injective $\rho: \text{acts}(\mathcal{S}) \rightarrow \text{acts}(\mathcal{S'})$ \n $\mathcal{S'} = (\rho(\text{in}(\mathcal{S})), \rho(\text{out}(\mathcal{S})), \rho(\text{int}(\mathcal{S})))$</td>
</tr>
<tr>
<td>process</td>
<td>$X_{\mathcal{S}}$</td>
<td>$\lambda$</td>
<td>$\mathcal{S}$</td>
<td>$X_{\mathcal{S}} \in \mathcal{X}_{\mathcal{S}}$</td>
</tr>
</tbody>
</table>

Table 1: The signature of DIOA

5.1.2 The Calculus of Demonic I/O Automata

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

Table 1 contains all the operators of DIOA and Table 2 contains their operational semantics in terms of transition systems. The operators of DIOA recall the standard operators of CCS
\[
\begin{array}{lll}
nil & \quad \text{nil}_s \xrightarrow{a} \Omega_s & \forall a \in \text{in}(S) \\
\text{ome}_1 & \quad \Omega_s \xrightarrow{a} \Omega_s & \quad a \in \text{ext}(S) \\
\text{ome}_2 & \quad \Omega_s \xrightarrow{\tau} \text{nil}_s \\
\text{pre}_1 & \quad a \cdot s e \xrightarrow{a} e \\
\text{pre}_2 & \quad a \cdot s e \xrightarrow{b} \Omega_s & \forall b \in \text{in}(S) \setminus \{a\} \\
\text{ich}_1 & \quad e_1 \oplus_s e_2 \xrightarrow{\tau} e_1 \\
\text{ich}_2 & \quad e_1 \oplus_s e_2 \xrightarrow{\tau} e_2 \\
\text{ich}_3 & \quad \frac{e_1 \xrightarrow{a} e'_1}{e_1 \oplus_s e_2 \xrightarrow{a} e'_2} & \quad \forall a \in \text{in}(S) \\
\text{ich}_4 & \quad \frac{e_2 \xrightarrow{a} e'_2}{e_1 \oplus_s e_2 \xrightarrow{a} e'_1} & \quad \forall a \in \text{in}(S) \\
\text{ech}_1 & \quad \frac{e_1 \xrightarrow{a} e'_1}{e_1 I + s e_2 \xrightarrow{a} e'_2} & \forall a \in I \cup \text{out}(S) \\
\text{ech}_2 & \quad \frac{e_2 \xrightarrow{a} e'_2}{e_1 I + s e_2 \xrightarrow{a} e'_1} & \forall a \in J \cup \text{out}(S) \\
\text{ech}_3 & \quad e_1 I + s e_2 \xrightarrow{a} \Omega_s & \forall a \in \text{in}(S) \setminus (I \cup J) \\
\text{ech}_4 & \quad \frac{e_1 \xrightarrow{\tau} e'_1}{e_1 I + s e_2 \xrightarrow{\tau} e'_1 I + s e_2} \\
\text{ech}_5 & \quad \frac{e_2 \xrightarrow{\tau} e'_2}{e_1 I + s e_2 \xrightarrow{\tau} e'_1 I + s e'_2} \\
\text{tau}_1 & \quad \frac{e \xrightarrow{a} e'}{\tau^s(e) \xrightarrow{a} \tau^s(e')} \\
\rho & \quad \frac{e \xrightarrow{a} e'}{e \xrightarrow{\rho(a)} e'} \\
\text{par}_1 & \quad \frac{e_1 \xrightarrow{a} e'_1}{e_1 s_1 \parallel s_2 e_2 \xrightarrow{a} e'_1 s_1 \parallel s_2 e'_2} \\
\text{par}_2 & \quad \frac{e_1 \xrightarrow{a} e'_1}{e_1 s_1 \parallel s_2 e_2 \xrightarrow{a} e'_1 s_1 \parallel s_2 e'_2} & \quad a \in \text{acts}(S_1) \setminus \text{ext}(S_2) \\
\text{par}_3 & \quad \frac{e_2 \xrightarrow{a} e'_2}{e_1 s_1 \parallel s_2 e_2 \xrightarrow{a} e_1 s_1 \parallel s_2 e'_2} & \quad a \in \text{acts}(S_2) \setminus \text{ext}(S_1) \\
\text{rec} & \quad \frac{e \xrightarrow{a} e'}{X \xrightarrow{a} e'} & \text{if } X \equiv e \\
\end{array}
\]

Table 2: The transition rules for DIOA. \(\tau\) is any internal action.
\[ si(nil) = \emptyset \quad so(nil) = \emptyset \]
\[ si(\Omega) = \emptyset \quad so(\Omega) = out(\Omega) \]
\[ si(a . e) = \{ a \} \cap in(e) \quad so(a . e) = \{ a \} \cap out(e) \]
\[ si(e_1 \oplus e_2) = si(e_1) \cap si(e_2) \quad so(e_1 \oplus e_2) = so(e_1) \cup so(e_2) \]
\[ si(e_1 \circ f e_2) = (I \cap si(e_1)) \cup (J \cap si(e_2)) \quad so(e_1 \circ f e_2) = so(e_1) \cup so(e_2) \]
\[ si(X) = si(E(X)) \quad so(X) = so(E(X)) \]

Table 3: Definition of \( si \) and \( so \) for DIOA.

[Mil89]; however they are different in the sense that they also guarantee input enabling by moving an automaton to the state \( \Omega \) whenever some unexpected input is provided. The expression \( nil \) models a quiescent automaton that moves to \( \Omega \) for any input. The prefixing operator allows the specification of an automaton which first perform a specific action \( a \). The internal choice operator models nondeterministic choice independently of the external environment. Particularly unfamiliar to the process algebraic community is the external choice operator, which is parameterized by two sets of input actions. The two parameters describe which arguments of the operator deal with different input actions. Consider the expression \( exp = a \cdot e_{[a] + [b]} \circ f b \cdot e f \). The subexpression \( a \cdot e \) is describing the behavior of \( exp \) in the presence of input action \( a \) while the subexpression \( b \cdot e \) is describing the behavior of \( exp \) in the presence of input action \( b \). The parameters are necessary since \( a \cdot e \) also reacts to input \( b \) although that reaction is not desired. The meaning of an expression like \( a \cdot e + b \cdot e f \), however, is intuitively clear. Although this intuition is not expressible for general DIOA expressions, Table 3 defines a function \( si(e) \) (Specified Inputs) which is capturing our intuitive idea for DIOA expressions of the kind \( a_1, e_1 + \cdots a_n, e_n \). Function \( si \) allows us to define an unparameterized choice operator by writing \( e + f \) for \( e_{si(e)} + sf f \), where function \( si \) is defined in Table 3. The interested reader is referred to [Seg92] for a more detailed description of \( si \) and its generalization to all DIOA expressions.

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

**Definition 5.9** Given a DIOA expression \( e \), the associated I/O automaton \( D(e) \) is defined as

- \( states(D(e)) = \{ e' \mid \exists t \in acts(e)^*, e \xrightarrow{t} e' \} \)
- \( start(D(e)) = \{ e \} \)
- \( sig(D(e)) = (in(e), out(e), int(e)) \)
Table 4: Some axioms for the quiescent preorder of DIOA.

- steps(D(e)) = \{(e', a, e'')|e' ∈ states(D(e)), e' \xrightarrow{a} e''\}
- part(D(e)) = \{local(e)\}

Proposition 5.10 Given two DIOA expressions e, f,

1. D(τ₁(e)) and Hide₁(D(e)) are the same I/O automaton;
2. D(e||f) and D(e)||D(f) only differ in that part(D(e||f)) = \{local(e) \cup local(f)\} and part(D(e)||D(f)) = \{local(e), local(f)\}.

The implementation relation for DIOA is the quiescent preorder, which is a weak congruence for all the operators but the unparameterized +. A weak congruence is a relation that is preserved under legal contexts, i.e., x R y implies C[x] R C[y] if C[.] is a legal context for both x and y. Table 4 contains some axioms for the quiescent preorder over DIOA. The axioms we present are just a some of those of [Seg92], however they are sufficient for our examples. They are sound in the sense that they state true properties of the I/O automata associated with the expressions. Axiom Ec₇ uses a function Quiet(f) which is true only if f is a quiescent expression, i.e., D(f) enables only input actions in its start state. Ec₇ models the idea that, whenever a specification e does not say anything about some input actions, any choice of implementation f in the presence of those actions is correct. Axiom I₃ allows us to move external actions out of the hiding operator. Axiom I₄ uses a function so in its side condition. Function so (Specified Outputs) is defined in Table 3 and gives those output actions of its argument that can be performed up to internal transitions. The side condition for Axiom I₄ is necessary since an external choice context is not resolved with internal actions (see transition rules ech₄₅). Axiom I₁₁ allows us to eliminate initial internal computation from I/O automata whenever no input is expected (si(e) = ∅). Two other important axioms deal with the parallel operator and with recursion. The expansion axiom allows to unfold a parallel expression into a nondeterministic sequential one; the recursive substitutivity rule states conditions for which a set of equations have unique fixpoint, and gives a method for proving that a process is implementing the fixpoint of a set of equations. In Section 5 the recursive substitutivity rule plays a fundamental role.
Proposition 5.11 (Expansion axiom) The following axiom is sound for the quiescent pre-order.

Let \( e \equiv e_1 \|^e_2 \cdots \|^e_n \) where each \( e_i \) is of the form \( \sum_j a_{ij} \cdot e_{ij} \). For each action \( a \in \text{ext}(e) \) let

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

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

\[
E_a = \begin{cases} 
\emptyset & \text{if } \text{out}(a) \neq 0 \text{ and } E_a^{\text{out}(a)} = \emptyset \\
\{ f_1 \cdots f_n : f_i \in E_a^i \lor (E_a = \emptyset \land f_i \equiv \Omega) \} & \text{otherwise}
\end{cases}
\]

Then \( e \equiv \bigcap_{a \in \text{ext}(e)} (\sum_{f \in E_a} a, f) \).

Theorem 5.12 (Recursive substitutivity) Let \( \tilde{X} \overset{\text{def}}{=} \tilde{E}(\tilde{X}) \) be a set of equations \( \{ E_i \overset{\text{def}}{=} \sum_j (a_j \cdot X_i_j) \} \), and let \( \tilde{P} \) be a set of DIOA expressions. If \( \tilde{P} \subseteq_{Q} \tilde{E}[\tilde{P}/\tilde{X}] \) then \( \tilde{P} \subseteq_{Q} \tilde{X} \).

5.2 Specification of the Components

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

Specification 5.13 (Muller C element) A Muller C element is specified as follows:

\[
\begin{align*}
C & \overset{\text{def}}{=} a \cdot C_a + b \cdot C_b \\
C_a & \overset{\text{def}}{=} a \cdot C + b \cdot C_{ab} \\
C_b & \overset{\text{def}}{=} a \cdot C_{ab} + b \cdot C \\
C_{ab} & \overset{\text{def}}{=} c \cdot C
\end{align*}
\]

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

**Specification 5.14 (Majority element)** A majority element is specified by the following equations

\[
M \overset{\text{def}}{=} a \cdot M_a + b \cdot M_b + c \cdot M_c \\
M_a \overset{\text{def}}{=} a \cdot M + b \cdot M_{ab} + c \cdot M_{ac} \\
M_{ab} \overset{\text{def}}{=} m \cdot M_c + c \cdot M_{abc} \\
M_{abc} \overset{\text{def}}{=} m \cdot M + a \cdot M_{bc} + b \cdot M_{ac} + c \cdot M_{ab}
\]

where $a, b, c$ are input actions and $m$ is an output action. The equations for $M_b, M_c, M_{ac}$ and $M_{bc}$ are similar to the equations above and can be easily derived.

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

**Specification 5.15 (Wire)** A wire is specified by the following equation:

\[
W \overset{\text{def}}{=} m \cdot c \cdot W
\]

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

**Proposition 5.16** $A(C_N) \equiv_F D(C), A(M_N) \equiv_F D(M), A(W_N) \equiv_F D(W)$.

**Remark 5.17** A stronger equivalence statement than the one of Proposition 5.16 hold, namely that the involved I/O automata are isomorphic if we do not consider states $\Omega$ and $\textit{nil}$. We do not need this strong statement for this paper, therefore we are not formal here.
5.3 The Verification

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

Proposition 5.18 $\tau_{(m)}(M||W) \sqsubseteq_Q C$, i.e., a Muller C element $C$ can be implemented using a majority element and a wire.

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

By applying the expansion axiom and the hiding axioms we obtain

$I \equiv_Q \tau_{(m)}(M||W)$

$\equiv_Q \tau_{(m)}((a \cdot M_a + b \cdot M_b + c \cdot M_c)||((m \cdot c \cdot W))$ by expanding $M||W$

$\equiv_Q \tau_{(m)}((a \cdot (M_a||((m \cdot c \cdot W)) + b \cdot (M_b||((m \cdot c \cdot W))))$ by Axiom $E_2$

$\equiv_Q \tau_{(m)}((a \cdot (M_a||W)) + b \cdot (M_b||W))$ by substituting $W$ for $E(W)$

$\equiv_Q \tau_{(m)}((a \cdot (M_a||W)) + \tau_{(m)}(b \cdot (M_b||W))$ by Axiom $I_4$

$\equiv_Q a \cdot \tau_{(m)}(M_a||W) + b \cdot \tau_{(m)}(M_b||W)$ by Axiom $I_3$

$\equiv_Q a \cdot I_a + b \cdot I_b$ by definition of $I_a$ and $I_b$

where we define

$I_a \overset{def}{=} \tau_{(m)}(M_a||W)$

$I_b \overset{def}{=} \tau_{(m)}(M_b||W)$

With the same method we have

$I_a \equiv_Q \tau_{(m)}(M_a||W) \equiv_Q a \cdot \tau_{(m)}(M||W) + b \cdot \tau_{(m)}(M_{ab}||W) \equiv_Q a \cdot I + b \cdot I_{ab}$

and

$I_b \equiv_Q \tau_{(m)}(M_b||W) \equiv_Q a \cdot \tau_{(m)}(M_{ab}||W) + b \cdot \tau_{(m)}(M||W) \equiv_Q a \cdot I_{ab} + b \cdot I$

where we define

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

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

$I_{ab} \equiv_Q \tau_{(m)}(M_{ab}||W)$

$\equiv_Q \tau_{(m)}(a \cdot (M||W) + b \cdot (M||W) + m \cdot (M_c||c \cdot W))$

$\equiv_Q \tau_{(m)}(m \cdot (M_c||c \cdot W))$
\[ \equiv \tau_{(m)}(m \cdot (a \cdot (M_{ac} || c \cdot W) + b \cdot (M_{bc} || c \cdot W) + c \cdot (M || W))) \]
\[ \subseteq \tau_{(m)}(m \cdot c \cdot (M || W)) \]
\[ \equiv \tau_{(m)}(M || W) \]
\[ \equiv c \cdot I \]

The first step follows the lines of the previous derivations by expanding process variables, applying the expansion theorem, and reconverting untouched expanded expressions to their corresponding process variable; the second step is an application of Axiom Ec7 where inputs a and b are eliminated. According to the specification of \( C_{a,b} \), in fact, no input should occur before output c occurs. The expression on the second line specifies an implementation choice in the presence of inputs a and b while the expression on the third line does not specify any implementation choice. The third step is similar to the first one while the fourth step consists of successive applications of the hiding axioms. Action m is eliminated through Axiom I_{11} and action c is brought outside the scope of the hiding operator through Axiom I_{9}. The last step is a direct consequence of the definition of I.

We can now apply the recursive substitutivity axiom and conclude \( \tau_{(m)}(M || W) \subseteq \tau_{c} C \). The fair trace inclusion follows from Theorem 5.7 and Proposition 5.8. All the involved I/O automata, in fact, are quiescent detectable, quiescent continuous, input quiescent detectable and have FIN. Moreover no input action disables any output action.

**Theorem 5.19** \( Hide_{(m)}(A(M_{N}) || A(W_{N})) \subseteq F A(C_{N}) \).

**Proof.** From Proposition 5.18, the soundness of the DIOA proof system, and Theorem 5.7, we derive \( D(\tau_{(m)}(M || W)) \subseteq F D(C) \). From Proposition 5.10 we derive \( Hide_{(m)}(D(M || W)) \subseteq F D(C) \). From Proposition 5.10 and Proposition 5.8 we have \( D(M || W) \subseteq F D(M || W) \), therefore we derive \( Hide_{(m)}(D(M || W)) \subseteq F D(C) \). Finally, from Proposition 5.16 we derive \( Hide_{(m)}(A(M_{N}) || A(W_{N})) \subseteq F A(C_{N}) \).

6 Comparison of the Algebraic and the Simulation Techniques

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

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

This leads to the question of whether the simulation and algebraic methods we have used might not be equivalent in general; however, it turns out that they are incomparable.

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

$$X \overset{\text{def}}{=} a \cdot b \cdot X + a \cdot c \cdot X$$

$$Y \overset{\text{def}}{=} a \cdot (b \cdot Y + c \cdot Y).$$

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

The difference between the systems $X$ and $Y$ arises when the decision about whether to perform $b$ or $c$ is made: $X$ decides before $Y$. A forward simulation between two processes $A$ and $B$ exists only if $B$ does not decide before $A$. $Y$ can be proved to implement $X$ by using a different simulation technique based on a notion of backward simulation [LV91]. However, there are also examples that can be proved using DIOA deductions but not by backward simulations. One example is

$$X \overset{\text{def}}{=} a \cdot c \cdot X + b \cdot Z \quad Z \overset{\text{def}}{=} c \cdot X$$

$$Y \overset{\text{def}}{=} a \cdot Z' + b \cdot Z' \quad Z' \overset{\text{def}}{=} c \cdot Y$$

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

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

$$X \overset{\text{def}}{=} a \cdot X \quad \text{and} \quad X_i \overset{\text{def}}{=} a \cdot X_{i+1}$$

for an infinite set of process variables $X_i : i \in \mathbb{N}$. The mapping that maps each $X_i$ into $X$ is trivially a forward simulation from $X_0$ to $X$; however, since none of the given equations relates some $X_i$ to $X_j$ with $j \leq i$, we cannot prove that $X_0 \leq a \cdot X_0$, so the recursive substitutivity

29
rule does not apply. The above mapping is also a backward simulation from $X_0$ to $X$, therefore also backward simulation is incomparable with DIOA deduction.

All the examples above also work for the simple trace preorder. The reader is referred to [DS92] for its axiomatization.

### 6.2 Treatment of Fairness

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

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

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

The generality of our approach to fairness in the simulation proof also remains to be considered; however, in this case there is already good evidence that this approach works well in practice [LS92, LLS93]. The approach based on the Execution Correspondence Lemma
provides a convenient way to base a fairness proof on a simulation proof; it may be that there are some fairness proofs that are inherently unable to be split in this way, but we do not know of any such examples. The use of forcing conditions provides a useful generalization of the usual I/O automaton fairness notion, but it seems likely to us that further generalizations will be required in order to describe some realistic liveness requirements. What those extensions might be, and whether they will work well in conjunction with the Execution Correspondence Lemma, remain to be seen.

Note that the arguments of this subsection only hold for fairness sensitive semantics such as the semantics of I/O automata. If the semantics is not based on a fairness sensitive relation, then the problems of this subsection disappear. Examples of non fairness sensitive relations are bisimulation [Mil89] and testing [DH84, Hen88].

6.3 Representation of Automata

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

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

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

In our small example, the state-based method gives a more elegant and concise representation of the circuits than the action-based method, but this will not be true in general. The choice of which representation is better will vary among different automata, depending upon whether the automaton table is most easily described by columns or by rows. Our experience shows that, for complex systems, the action-based description is usually the better one [LLS93].

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

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

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

\[
X_0 \overset{\text{def}}{=} \text{up} \cdot X_1 \\
X_n \overset{\text{def}}{=} \text{down} \cdot X_{n-1} + \text{up} \cdot X_{n+1} \quad \text{if } n > 0.
\]

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

\[
X_n \overset{\text{def}}{=} \text{up} \cdot X_{n+1} + (\text{if } n > 0 \text{ then } \text{down} \cdot X_{n-1}).
\]

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

\[
\begin{align*}
\text{if } \text{precondition}(a_1) \text{ then } \text{effect}(a_1) \text{ else } \\
\text{if } \text{precondition}(a_2) \text{ then } \text{effect}(a_2) \text{ else } \ldots
\end{align*}
\]

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

\[
egin{align*}
Y_{10} \overset{\text{def}}{=} \text{up} \cdot X_{11} \\
X_n \overset{\text{def}}{=} \text{down} \cdot X_{n-1} + \text{up} \cdot X_{n+1} \quad \text{if } n > 10.
\end{align*}
\]

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

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

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

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

In the process algebraic proof given in this paper the axioms that have to be applied during each step are partially determined by the equations defining the specification automaton. Our proof steps were essentially repeated applications of the expansion axiom followed by some simplifications based on the given specification. This heuristic is generally applicable when dealing with (finite state) circuit descriptions. It is also applied in [Jos92, Seg92] and in several of the examples of [Bae90]. In these cases, algebraic manipulators like those of [MV91, Lin91] can be used. However, when the problem becomes large or is described by an infinite state machine, the remarks at the end of Section 6.3 show that some form of simulation has to be defined even for an algebraic proof, therefore the difficulties involved in the mechanization of simulation and algebraic proofs are comparable.

6.5 Additional Benefits Obtained from the Proof

Experience with large simulation-based verifications [WLL88, LP92, LLS93] has shown that the formal description of the simulation relation in a simulation proof constitutes an important
piece of documentation of the key ideas of the implementation, in much the same way that an
invariant assertion does; invariants and simulations typically express the key intuitions that
make the implementation work. Similarly, due to the remarks at the end of Section 6.3, an
algebraic proof can embed some form of mapping which can be used as a documentation.

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

In our experience simulation proofs are flexible in the sense that a given proof can usually
be modified fairly easily in order to verify new properties of an implementation. A typical
verification task, for example the one in [LLS93], involves the definition of specification and
implementation automata and the proof that the implementation meets the specification. Dur-
ing the proof some errors might be discovered and the involved automata might need to be
modified. Also, after the proof is completed, the specification and/or implementation automata
might be slightly modified in order to make them cleaner and more general. The simulation
relation and the correctness proof might then have to be correspondingly modified. In general
the structure of the simulation proof seems to provide us with a lot of guidance in carrying
out such modifications, since its general structure is usually preserved. To the extent that
an algebraic proof embeds a simulation proof, the same advantages for modifiability would
accrue.

7 Conclusion

Using a simple example based on delay insensitive circuits, we have compared two widely used
verification techniques for concurrent and distributed systems. The assertional methods based
on I/O automata have been successfully used for the verification of very complex systems
[LT87, WLL88, LP92, LLS93] while the algebraic techniques of process algebras [Mil89] have
generally been used for relatively small examples [Bae90, Jos92].

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

The example we have used is one of the typical examples of the process algebraic community;
therefore, it should not be surprising that the process algebraic analysis looks shorter than the
simulation-based one. Starting from the presented example, however, our discussion has shown
that scaling algebraic proofs to more complex systems leads to the use of simulation-based
verification techniques.
Although we have emphasized verification in this paper, it is important to remember that verification is not the only purpose, nor even the main purpose, of process algebra. Rather, process algebra is intended to provide compositional semantics for programs. Of course, one important use for such a semantics is to provide a basis for carrying out formal correctness proofs for systems. Since one of the most practical verification methods is simulation, it is important that an algebraic semantics be designed with a view toward compatibility with simulation proofs. Given a program that is supposed to implement a given specification, a process algebraic characterization of the semantic model can be used to compositionally compute the semantics of the given program, then a simulation-based technique can be used to prove the correctness of the implementation. Perhaps, we could also add an intermediate step in which the meaning of a program is algebraically simplified before starting with the assertional part of the correctness proof.

References


A note on Model Checking Context Free Processes
(Preliminary Report)

S. Purushothaman Iyer*
Dept of Computer Science
NC State University
Raleigh, NC 27695-8206
June 1, 1993

Abstract

We consider the problem of model checking context free processes against Büchi automaton
on infinite strings as specifications. We show that computation of context free processes are
definable using PDA on infinite trees. By adopting a automata theoretic framework we provide
solution that does not need the notion of "higher-order" semantics as used in the literature [1, 3].
Furthermore it allows us to consider a generalization of context free processes called macro
processes.

1 Introduction

There has been a lot of interest lately in context free processes, given that bisimulation is decidable
for such processes [4]. Spurred by the bisimulation result, the problem of model-checking context
free processes against alternation-free modal $\mu$-calculus formulas have been considered [1, 3]. In
the former the authors consider an iterative version of model-checking and in the latter a tableau
based technique. Part of the motivation for this paper is to explore the third alternative, that
of automata-theoretic based technique. In this paper we will consider the problem of model-
checking context free processes against specification expressed using Büchi-automaton on infinite
strings. The automata-theoretic approach allows us to provide a simple solution that does not
involve the notion of "higher-order" semantics as used in [1, 3]. Along the way we provide a tight
characterization that unfoldings of context free processes (considered as trees) are definable using
Pushdown automaton on infinite trees. Finally, we also consider the problem of model checking a
generalization of context free processes called macro processes.

In [1, 3] the authors present context free processes as a set of procedural graphs, one for each
procedure or (equivalently) non-terminal. The graphs that define the procedure are transition
systems whose transitions are labeled by either actions or procedure names. In this context the
procedures can be thought of as parameterless procedures. We generalize these procedural graphs

*Supported in part by NSF under grant NSF CCR-90-0412
to take parameters and call the new process calculus as macro processes. Under the natural restriction that these parameters are context free processes, viz. procedure graphs that do invoke only parameterless procedures, we show that the computation of macro processes are definable using stack automaton on infinite trees. These representations pave the way for stating modal checking as emptiness problems of appropriate automaton. As the specification is defined using Büchi automaton on infinite strings, the model checking problem for context free (macro) processes reduces to emptiness of pushdown automaton (stack automaton, respectively) over infinite strings. The central contributions of the paper are, therefore, (a) the concept of macro processes, (b) the representation of the computation of context free and macro processes as trees definable by pushdown automaton and stack automaton, respectively, and (c) model checking for context free and macro processes using an uniform automata theoretic technique, as suggested in [9].

In Section 2 we provide the necessary definitions regarding automata on infinite objects and recall some results. In Section 3 we show how the computational trees of context free processes can be represented using Pushdown automaton on infinite trees and in Section 4 we do the same for macro processes.

2 Pushdown Automaton

Let $\Sigma$ be an alphabet. Let $[k]$ be the set $\{0, 1, \ldots, (k - 1)\}$. A $k$-ary tree $t$ over $\Sigma$ is a partial mapping $[k]^* \to \Sigma$ such that the following condition is satisfied:

If $t(x.i)$ is defined, where $x \in [k]^*$ and $0 \leq i \leq (k - 1)$, then $t(x)$ and $t(x.j)$, where $0 \leq j < i$, are also defined.

A Büchi automaton on infinite trees over $\Sigma$ is a 4-tuple $(S, q_0, \delta, F)$, where $S$ is the set of states, $q_0$ is the start state, $\delta$ is the transition function and $F$ is the set of final states. There are two rules among the transition function $\delta$: the read rule and the $\epsilon$ rule. The read rule is of the form:

$$(p_0, p_1, \ldots, p_{k-1}) \in \delta(q, a), \text{ where } a \in \Sigma, q \in S \text{ and } \forall 0 \leq i < k : p_i \in S$$

The $\epsilon$ rule, on the other hand, is of the form:

$$p \in \delta(q, \epsilon)$$

Given a tree $t$, a run of the tree is obtained by decorating each node $x$ of the tree by a finite sequence of states $q_0^x q_1^x \ldots q_n^x$ satisfying some conditions. Formally, a run is a function $r : [k]^* \to S^*$ such that:

- $r(\epsilon) = q_0 q_1 \ldots q_m$, where $q_0$ is the start state.
- $\forall 0 < i \leq n, \ q_i^x \in \delta(q_{i-1}^x, \epsilon)$, i.e., state $q_i^x$ is reached by using an $\epsilon$-rule from $q_{i-1}^x$.
- Let $x$ be a node of the tree with children $x.0, x.1, \ldots, x.(k - 1)$, and let $t(x) = \sigma$. Let $r(x) = \alpha. q_{n_x}^x$, where $\alpha \in S^*$, and $r(x.i) = q_i^x. \alpha_i$, $\forall 0 \leq i \leq (k - 1)$, such that $\alpha_i \in S^*$. Then we should have the relation $(q_0^x.0, \ldots, q_0^x.(k-1)) \in \delta(q_{n_x}^x, t(x))$. Put differently the first state of siblings are obtained from the last state decorating the parent node by the use of a read rule.
A path through a tree is an infinite sequence of nodes \( p = \{x_i\}_{0 \leq i} \) such that \( x_{i+1} = x_i . j \), \( j \in [k] \). A full path through the tree is a path such that \( x_0 = \epsilon \). Given an infinite sequence of states \( \{q_i\}_{0 \leq i} \), define \( \text{infinity}(\{q_i\}_{0 \leq i}) \) as the set of states that occur infinitely often in that sequence.

Let \( \{x_i\}_{0 \leq i} \) be a full path of a run \( r \). Define the state trace \( \tau \) of the full path \( \rho = \{x_i\}_{0 \leq i} \) as \( \Pi_{0 \leq i} r(x_i) \), where \( \Pi \) denotes concatenation. A run \( r \) is acceptable provided the trace of every full path of \( r \) contains final states that appear infinitely often. Formally, a run is acceptable provided for every full path \( \rho \) of \( r \), \( \text{infinity}(\tau(\rho)) \cap F \neq \emptyset \). A tree \( T \) is accepted by the a Büchi automaton provided the tree has an acceptable run.

Note that the definition of Büchi automaton we use is different from the standard definition. The \( \epsilon \) rules, the only addition to the standard definition, do not increase the expressive power as they can be removed by subset construction.

A Büchi automaton on infinite strings is a special case where the arity \( k \) of the tree is 1.

A pushdown automaton on infinite trees works similar to a Büchi automaton, except that it has the stack to deal with. A \( \omega \)-PDA that accepts infinite trees over \( \Sigma \) is a 6-tuple \( (S, \Gamma, q_0, Z_0, \delta, F) \), where \( \Gamma \) is the set of stack symbols, \( Z_0 \) is the bottom stack marker and \( F \) is the set of final states. The transition function \( \delta \) again has read transitions and \( \epsilon \) transitions. An \( \epsilon \) transition is now of the form:

\[
(q', \gamma) \in \delta(q, Y, \epsilon),
\]

where \( q, q' \in S \), \( \gamma \in \Gamma^* \), and \( Y \in \Gamma \).

A read rule is of the form

\[
((q_0, \gamma_0), (q_1, \gamma_1), \ldots, (q_{k-1}, \gamma_{k-1})) \in \delta(q, Y, \sigma), \text{ where } \forall i \in [k]q_i, \gamma_i \in S, \forall i \in [k] \gamma_i \in \Gamma^*, Y \in \Gamma, \text{ and } \sigma \in \Sigma
\]

Much like earlier, given a tree \( t \), a run \( r \), of \( t \), is a mapping from nodes of the tree to finite sequences of pairs of the form \((q_0, \gamma_0)(q_1, \gamma_1)\ldots(q_n, \gamma_n)\) where \( q_i \) are states and \( \gamma_i \in \Gamma^* \) are sequences of stack symbols. The \((q_i, \gamma_i)\) are related to \((q_{i-1}, \gamma_{i-1})\) by \( \epsilon \) transitions. Furthermore, if \( x \) is a node with children \( z.0, \ldots z.(k-1) \), then the first pair for each of the children are related to the last pair of \( r(x) \) by a read rule. Formally, we have

- \( r(\epsilon) = (q_0, Z_0)(q_1, \gamma_1)\ldots(q_n, \gamma_n) \) where \( \forall 1 \leq i \leq n : \gamma_i \in \Gamma^* \).
- If \( r(x) = (p_0, \gamma_0)(p_1, \gamma_1)\ldots(p_n, \gamma_n) \) then we have \( \forall i \in [n-1] : \gamma_i = \gamma'Y, \gamma_{i+1} = \gamma'\gamma, \text{ and } (p_{i+1}, \gamma) \in \delta(p_i, Y, \epsilon) \).
- If \( t(x) = \sigma \) and \( r(x) = \alpha.(p, \gamma Y) \) then \( \forall i \in [k]r(z.i) = (p_i, \gamma\gamma_i).\alpha_i \) such that \( (p_0, \gamma_0)\ldots(p_{k-1}, \gamma_{k-1}) \in \delta(p, Y, \sigma) \) and \( \alpha, \alpha_i \in (S \times \Gamma^*)^* \).

Define the state trace for a node as \( \tau(x) = q_0 \ldots q_n \), where \( r(x) = (q_0, \gamma_0)\ldots(q_n, \gamma_n) \), and for a full path \( \rho = \{x_i\}_{0 \leq i} \) as \( \tau(\rho) = \Pi_{0 \leq i} r(x_i) \). A run is considered acceptable provided the state trace of every full path has final states infinitely often in it.

The following example from [5] illustrates the power of PDA on infinite trees.
Example

Consider the PDA on infinite trees \( M = (S, \Sigma, \Gamma, q_0, Z_0, \delta, F) \) over \( \Sigma = \{a, b, c\} \), where \( S = \{q_0, q_1, q_2\} \), \( \Gamma = \{B, Z_0\} \), and \( \delta \) is defined by:

1. \( \delta(q_0, a, Z_0) = \{(q_1, BZ_0)(q_2, BZ_0)\} \);
2. \( \delta(q_1, b, B) = \{(q_1, \varepsilon)(q_1, \varepsilon)\} \);
3. \( \delta(q_1, c, Z_0) = \{(q_1, Z_0)(q_1, Z_0)\} \);
4. \( \delta(q_2, a, B) = \{(q_1, BB)(q_2, BB)\} \)

(this example is a modified version of Example 1 from [7]). Let \( F = \{q_1, q_2\} \). \( T(M) \) contains only one 2-ary tree \( t \) where the rightmost branch of \( t \) contains the \( \omega \)-word \( a^\omega \), and each of other branches begins with \( \iota(>0) \) \( a \)'s, followed by \( \iota(0) \) \( b \)'s, and then followed by infinitely many \( c \)'s (refer to Figure 1).  

We recall some results that we will make use of in the rest of the paper. The first result we need is regarding the emptiness for PDA:

**Theorem 2.1 (Harel and Raz [2])** The emptiness problem of PDA on infinite trees is solvable in triple-exponential time.

The proof by Harel and Raz is based on a pumping lemma for the trees accepted by a PDA. In contrast, by a direct analysis of context free term grammars, it can be shown that

**Theorem 2.2 ( [5])** The emptiness problem of PDA on infinite trees can be solved in single exponential (in the number of states and the number of stack symbols) time.

**Theorem 2.3** The class of trees acceptable by PDA on infinite trees is closed under intersection with the class of trees acceptable by Büchi automaton.
Proof (sketch): By a simple product of the two machines. The states of the new machine are product states, and the final states of the new machines are obtained as the product of the final states of the two machines.

Finally, we also need the following result regarding complementation:

**Theorem 2.4 ([8])** The class of sets of infinite strings accepted by Büchi automaton is closed under complementation.

## 3 Context Free Processes

Context free processes arise by treating a context free grammar (in Greibach Normal form) as a process specification. An alternative characterization has been used in [1, 3] (which we will follow) where a context free process is specified as a set of parameterless procedures. Each procedure can be thought of as an extended finite state process where the transitions can be labeled by actions and by procedure names.

Let $V$ be the set of actions symbols and $P$ be the set of procedures that make up a process. A context free process will be represented by a set of procedure graphs [1, 3]. Each procedure $P \in P$ is represented by a procedure graph $PG_P$ as a 5-tuple $(S_P, V_P, \delta_P, i_P, e_P)$ where $S_P$ is the set of states, $V_P$ is the set of action symbols, $\delta_P \subseteq S_P \times (V_P \cup P) \times S_P$ is the transition relation, $i_P$ is the initial state and $e_P$ is the end state. We will assume, without loss of generality, that $\forall P, Q \in P : S_P \cap S_Q = \emptyset$.

We also make the following additional assumptions [1, 3]:

- The indegree of the node representing the state $i_P$ is zero, and the outdegree of the node representing $e_P$ is also zero.
- Each use of a procedure name is guarded by an action.
- $V = \cup_{P \in P} V_P$.

Finally, we can now define a context free process as $(\{PG_P | P \in P\}, V, P_0)$, where $P_0$ is the root procedure.

The computational graph of a context free process is obtained by expanding on the procedural calls. Starting with $P_0$, a sequence of graphs can be obtained by substituting for procedure calls. Formally,

**Definition 3.1** The *computation graph* associated with a context free process is the one obtained as a result of repeated expansions such that it does not contain any transitions labeled by procedure names. The substitution of a procedural graph $PG_P$ for a transition $u \xrightarrow{P} v$ can be carried by identifying the start state $i_P$, of $PG_P$, with the state $u$ and the end state $e_P$, of $PG_P$, with $v$.

A *computational tree* corresponding to a context free process can be obtained by unrolling the computational graph corresponding to that process. If a node $x$ has successor nodes $x_1, x_2, \ldots, x_n$ then by creating fresh copies of the nodes $x_i, 1 \leq i \leq n$, we can obtain the computational tree.

5
3.1 The Translation

In this subsection we will show that corresponding to every context free process there exists a PDA on infinite trees that accepts precisely the computation tree of that process. Clearly, the arcs of the computation tree of a context free process are labeled by actions, whereas the arcs of trees accepted by a PDA are not labeled. This apparent distinction can be taken care of easily by encoding both the arcs and the nodes of the computation tree as labels of the tree being accepted by the PDA. But in the following we will only preserve the states of the computational tree in the translation. Our motivation for doing so is that we consider Büchi specifications to be the outgrowth of translating formulae in a linear time temporal logic which are defined over a single program letter. As any logic which allows more than one program letter and whose models are Büchi automaton definable can be easily handled, our assumption is reasonable.

In order to discuss the properties that hold at a particular state, we will use a finite set of propositions ℒ, and a labeling of states by these propositions. Consider the Büchi specification we are given. Clearly, given our assumption above, the strings accepted by these Büchi specification can be thought of as being labeled by subsets of propositions from ℒ. By doing so, we are concentrating more on the temporal unfolding of the properties (or propositions) that hold over a period of time, rather than the identity of the state reached. Thus in the rest of the paper, we will take the set of input symbols Σ, over which the computational trees and automaton (specification) are defined, to be 2ℒ.

Consider a state s ∈ SP − {iP, eP} in some procedure P of a context free process P. The state s appears in a number of places in the computational tree, say s₁, s₂, ..., sₘ. Clearly, all of these states si have the same computational content as the original copy s. Formally, we have

**Definition 3.2**  Let S be the set of all states of a computational tree. An assignment function ℓ : S → ℒ is legal provided for all s ∈ S and s' ∈ S which are instances of the same state of a procedural graph we have ℓ(s) = ℓ(s').

Note that none of the start or end states of any procedural graph would find its way into the computational tree. Thus, given a legal assignment function ℓ on a computation tree for a context free process P, there exists an unique assignment ℓ' : ∪P∈P(SP − {iP, eP}) → ℒ such that image of state s under ℓ' is the same as the image of any instance s', of s, under ℓ.

Note that by the first and the second condition given earlier regarding the structure of procedural graphs, the computational tree will be of finite degree. The outdegree of any node in the computational tree is bounded by k² where k is the maximum outdegree of any node in the procedure graphs (due to the restriction that every use of procedure should be guarded). Let T be a transition (s, a, s'). Define count(T) = 1 if a is an action symbol, or k where k is the outdegree of the node iₐ, the start node of process a. Overloading the definition of count, define, for any node s,

\[ \text{count}(s) = \Sigma_{i=1}^{k} \text{count}(T_i), \quad T_i \text{ is the } i^{th} \text{ transition out of node } s. \]

Let P be a context free process. Let ℓ : ∪P∈P(SP − {iP, eP}) → ℒ be an assignment function. Define a PDA M_P = (S, Γ, δ, s₀, Z₀, F) corresponding to P as follows:

- The set of states S = ∪P∈P(SP − {iP, eP}) ∪ {s₀}, where s₀ ∉ S.
• The stack symbols \( \Gamma = S \cup \{Z_0\} \). The stack symbols turn out to be return addresses, and hence we use \( S \).

• The final set of states \( F = S \).

• The specification of \( \delta \) proceeds as follows:

  Consider a procedure graph for \( P \), and a state \( s \neq e_P \) in a procedure \( P \). Let \( s_1, \ldots, s_n \) be the successors of \( s \). For every stack symbol \( Z \) construct a read transition for \( (p_1, \gamma_1) \ldots (p_m, \gamma_m) \in \delta(s, Z, t(s)) \), where \( m = \text{count}(s) \).

  Let \( T_i = (s, a_i, s_i) \) be the \( i^{th} \) transition out of \( s \). Let \( n = \text{count}(T_1) + \ldots + \text{count}(T_{i-1}) \).

  -- If \( a_i \) is an action symbol and \( s_i \neq e_P \) then \( p_{n+1} = s_i \) and \( \gamma_{n+1} = Z \). If \( a_i \) is an action symbol and \( s_i = e_P \) then make \( p_{n+1} = Z \), the return address, and \( \gamma_{n+1} = \varepsilon \), thus removing the return address from the stack.

  -- Consider the case where \( a_i \) is not an action symbol but is a procedure name \( Q \). In this case, corresponding to the single transition \( T_i \) we have to create \( \text{count}(i_Q) \) transitions corresponding to jumping to each of the successors of the initial state \( i_Q \) of procedure \( Q \). Let, as in the earlier case, \( n = \text{count}(T_1) + \ldots + \text{count}(T_{i-1}) \) and \( n' = \text{count}(i_Q) \). Let these \( n' \) transitions be \( (i_Q, b_1, t_1), \ldots, (i_Q, b_{n'}, t_{n'}) \). Clearly, \( b_j, 1 \leq j \leq n' \) has to be either actions symbols or \( \varepsilon \).

  Let \( b_j \) be \( \varepsilon \). In which case we necessarily have \( t_j = e_Q \). If \( s_i = e_P \) then we not only need to return from \( Q \) but also from \( P \), and therefore, \( p_{n+j} = Z \) and \( \gamma_{n+j} = \varepsilon \). If \( s_i \neq e_P \) then we return back to \( P \) immediately, and therefore, \( p_{n+j} = s_i \) and \( \gamma_{n+j} = Z \).

  Let \( b_j \) be an action symbol. There are four cases depending upon whether \( s_i = e_P \) or not, and on whether \( t_j = e_Q \) or not. We have:

  \[
  (p_{n+j}, \gamma_{n+j}) = \begin{cases} 
  (Z, \varepsilon) & \text{if } s_i = e_P \text{ and } t_j = e_Q \\
  (t_j, Z) & \text{if } s_i = e_P \text{ and } t_j \neq e_Q \\
  (s_i, Z) & \text{if } s_i \neq e_P \text{ and } t_j = e_Q \\
  (t_j, s_i Z) & \text{if } s_i \neq e_P \text{ and } t_j \neq e_Q 
  \end{cases}
  \]

  -- Using the discussion above we also create a read transition for \( \delta(s_0, Z_0, t(s_0)) \), where \( s_0 \) corresponds to the start state \( i_{P_0} \) of the root procedure \( P_0 \).

• The set of final states \( F = S \).

By the discussion given above we have:

**Theorem 3.1** The computational tree of a context free process is definable using a PDA on infinite trees.

In all of this discussion we have considered PDA that accept infinite trees wherein on any branch of a run the set of final states are visited infinitely often. In the string case (both finite and infinite)
PDAs that accept by empty stack and PDAs that accept by final state are equally expressive. But
this is not so in the infinite tree case. PDAs that accept infinite trees by final states are more
powerful than PDAs that accept by empty stack [5]. Furthermore, we have the hierarchy that trees
acceptable by Büchi automaton are strictly less powerful than those accepted by PDA with empty
stack condition. We now have the following tight characterization:

**Theorem 3.2** There are context free processes \( P \) corresponding to which there is no PDA \( M_P \)
with empty stack condition that would accept the computation tree of \( P \).

Based on the observation that the computation tree that corresponds to the context free process
given by

\[
S ::= aSb|\varepsilon
\]

has an infinite branch where the stack keeps growing unboundedly.

As we are interested in model-checking the context free process against a Büchi-automaton
specification on infinite strings, we need to linearize the tree into strings. This is easily done as
follows:

**Theorem 3.3** Let \( M = (S, \Gamma, q_0, Z_0, \delta, F) \) be a PDA on infinite trees over the alphabet \( \Sigma \). The
linearization of \( T(M) \) is precisely the set of infinite strings accepted by \( M' = (S, \Gamma, q_0, Z_0, \delta', F) \), a
PDA on infinite strings over alphabet \( \Sigma \), where \( \delta' \) is defined as:

- For each \( \varepsilon \)-transition \(( q', \gamma ) \in \delta(q, Z, \varepsilon) \) we have a \( \varepsilon \)-transition \(( q', \gamma ) \in \delta'(q, Z, \varepsilon) \).

- For each read transition \((( q_0, \gamma_0 ), \ldots, ( q_{k-1}, \gamma_{k-1} )) \in \delta(q, Z, a) \) we have \( k \) read transitions
  \(( q_i, \gamma_i ) \in \delta'(q, Z, a), \forall 0 < i < (k - 1) \).

Clearly, given that (a) Büchi automaton on infinite trees are closed under complementation,
and (b) PDA on infinite trees (strings) are closed under intersection with Büchi automaton on
infinite trees (strings), it is easy to see that a context free process can be model-checked against a
specification in as Büchi automaton on infinite strings.

**4 Macro Processes**

In this section we will consider a generalization of context free processes, called *macro processes*,
whose trees are definable by a Stack automaton. Note that Stack automaton (on finite strings) can
accept languages such as \( a^n b^n c^n \).

A macro process is again specified using a set of procedural graphs. Each procedure, in addition
to being defined by a graph, can take parameters. The transitions can now be labeled by actions,
procedure names (or procedure calls), and formal parameters. There are two restrictions: The first
restriction is that a variable has to be guarded by an action symbol (much like procedure names
of context free processes). The second restriction is that the actual parameters to a procedure call
can only be a context free process, or equivalently the name of a context free process.
A stack automaton is a generalization of a PDA that can read into the stack. The main change from the definition of a PDA is in the specification of the transition function $\delta$. It is now of the form

$$(q, \gamma, d) \in \delta(p, Z, a)$$

where $d$ is the direction in \{up, down\}. Furthermore stack symbols can be added or removed only when the stack head is at the top.

A stack automaton on infinite trees [2] is similar to PDA on infinite trees, except that the stack head can read into the stack. Furthermore, to define the acceptance condition only those configurations containing final states are taken into account where the stack head is at the top of the stack.

The translation of macro processes to stack automaton on infinite trees works as follows:

- When a procedure is called not only the return address but also the address of the actual parameters are stored.

- A transition that involves looking up a variable can be translated to a $\varepsilon$ transitions that looks deep into the stack for the beginning address of the actual parameter under question.

- Finally, each actual parameter (being a context free process) can be represent by a set of procedural graphs.

By the discussion above, we have the following:

**Theorem 4.1** The computation trees of a macro process are definable by a stack automaton on infinite trees.

Given that the emptiness problem for stack automaton is solvable, we have

**Theorem 4.2** The problem of model checking macro processes against Büchi automaton specifications is solvable.

5 Conclusion

We have explored an automata theoretic approach to model checking in this paper, which allows us to solve the problem of model checking context free processes and macro processes in an uniform way. Importantly we have shown that a larger class of process specifications (viz. macro processes) can be model checked. Finally, we have also offered characterizations of the computational trees of context free processes and macro processes.

References


State Refinement in Process Algebra  
(Extended Abstract)

Andrew C. Uselton and Scott A. Smolka*  
Department of Computer Science  
SUNY at Stony Brook  
Stony Brook, NY 11794-4400, USA  
{uselton,sas}@cs.sunysb.edu

Abstract

We introduce a state refinement operator into BPA with recursive specifications and present a comprehensive technical development of the resulting theory, BPA + SR. Our main technical results are that bisimulation is a congruence in BPA + SR and that guarded recursive specifications have unique solutions. We also have that bisimulation remains a congruence if the merge operator of ACP is added to BPA + SR. This is significant since action refinement, another approach to refinement in process algebra, does not in general preserve semantic equivalences based on interleavings of atomic actions.

State refinement, to our knowledge, represents the first attempt to capture the essence of Harel's statecharts — viz., hierarchically structured state transition behavior — within a purely process algebraic setting. A succinct, hierarchical specification of an alarm clock is given to illustrate the utility of state refinement in process algebra.

1 Introduction

Statecharts [Har87] is a highly structured and economical description language for complex systems, such as communication protocols and digital control units. Statecharts extend conventional state transition diagrams with three elements dealing with the notions of hierarchy, concurrency and communication. The last two notions are already present in classical process algebra (see, for example, [Mil89]), but the statechart notion of hierarchy — in which states can have substructure, and substates can have subsubstructure, etc. — has till now been lacking.

In this paper, we introduce a state refinement operator into BPA with recursive specifications [BK84, BW90], and present a comprehensive technical development of the resulting theory. State refinement, to our knowledge, represents the first attempt to capture the essence of statecharts — viz., hierarchically structured state transition behavior — within a traditional process algebraic setting.

*Research supported in part by NSF grant CCR-9120995.
State refinement is a binary operator on processes written \( s[t] \). The intuitive meaning behind this expression is that state \( t \) refines state \( s \) or, equivalently, \( s \) is a state with substructure \( t \). Operationally, the behavior of \( s[t] \) is given essentially by the following rules (but see Sections 2 and 3 for the complete set of action rules):

\[
\begin{array}{c}
\frac{s \xrightarrow{a} s'}{s[t] \xrightarrow{a} s'} & \frac{t \xrightarrow{a} t'}{s[t] \xrightarrow{a} [t][t']}
\end{array}
\]

Intuitively, these rules say that \( s[t] \) can perform a transition of \( s \), effectively bypassing the possibilities offered by its substructure \( t \), or \( s[t] \) can behave like its substructure, performing a transition of \( t \). The choice is nondeterministic.

Note that state refinement defies the traditional classification of process algebra operators into static and dynamic: it is quintessentially a hybrid, the first rule capturing its dynamic behavior and the second rule its static behavior.

State refinement bears an interesting relationship to the mode transfer operator of Bergstra [Ber89]. These operators were developed independently of one another and serve very different purposes, yet they turn out to have similar (but not identical) action rules and, likewise, similar axiomatizations. Mode transfer has its roots in the disabling operator of LOTOS [Int87, BB87], and was proposed by Bergstra to develop the idea of disabling in a process-algebraic setting.

Consider the state-refined process \( s[t] \), i.e., \( s \) has substructure \( t \). The analogous expression in mode transfer notation is \( t \rightarrow s \), representing the process for which \( t \) is the normal mode of behavior and, at any instant before the termination of \( t \), control can transfer to \( s \). Note that the order of the operands is switched in the two expressions, reflecting the very different ways in which we think about state refinement versus mode transfer.

One might try to place state refinement and mode transfer on equal footing by viewing \( s[t] \) as a process whose substructure can be “interrupted” by a “higher level” transition, i.e., one of \( s \). But this does not quite work and the problem arises when we consider the termination of \( t \). In the case of \( t \rightarrow s \), transfer of control to \( s \) is no longer an option, and the entire process comes to a halt. This is not the case for state refinement: the termination of \( t \) in \( s[t] \) is not so fatal as \( s \) is still free to proceed with its “own” transitions. Only its substructure has ceased to exist.

It is exactly this difference in behavior in response to \( t \)’s termination that leads to the difference in action rules (and, consequently, axioms) for state refinement and mode transfer. The rule in question for state refinement is given by:

\[
\frac{t \xrightarrow{a} \sqrt{}}{s[t] \xrightarrow{a} s}
\]

while the corresponding rule for mode transfer is given by:

\[
\frac{t \xrightarrow{a} \sqrt{}}{t[> s] \xrightarrow{a} \sqrt{}}
\]

Otherwise, the action rules for these two operators are the same.
As adroitly pointed out by Bergstra [Ber89], mode transfer (and hence disabling) is a "practical" operator in that it offers a succinct notation for an oft-occurring system behavior. In fact, eliminating mode transfer from a specification in favor of sequential nondeterminism, in general, results in an expression exponentially larger than the original (the "expansion theorem" for concurrency witnesses the same phenomenon). For these same reasons, we believe state refinement can be viewed as a desirable operator to add to a process algebra.

It is interesting to also compare state refinement to another approach to refinement in process algebra: action refinement [AH89, vG90, Gor91, Ace92]. Like state refinement, action refinement supports hierarchical specifications of state transition behavior, this time through the refinement of uninterpreted actions into processes. Note that the homogeneity of state refinement, in which a process is refined by a process, is not present in action refinement.

A problem that arises in the setting of action refinement is that semantic equivalences based on an interleaving interpretation of concurrency are not preserved by action refinement [CdMP87]. For example, let $s \overset{df}{=} a \parallel b$ and $t \overset{df}{=} a \cdot b + b \cdot a$. In an interleaving model, $s = t$ yet $\rho_f(s) \neq \rho_f(t)$ where $\rho_f$ is a refinement operator that (syntactically) applies the map $f(a) \mapsto c \cdot d$, $f(b) \mapsto b$ to atomic actions.

Because of its non-preservation of interleaving-based equivalences, action refinement is usually coupled with a model of concurrency based on some notion of partial order semantics (see, e.g., [NEL88, vGG88, DD90, vG90, Vog90, JM92]). State refinement does not have this problem. In the term model presented in this paper, and even in the presence of the merge operator of ACP [BK84], bisimulation is a congruence for all operators, including state refinement.

Summary of Technical Results

The technical development of BPA + SR proceeds along fairly standard lines (see, e.g., [BW90]) and we encountered few technical difficulties in integrating state refinement into process algebra. This is to be expected given the experience of Bergstra [Ber89] with the closely related mode transfer operator. Seamless integration of the state refinement and mode transfer operators in process algebra may be viewed as testimony to their "well behavedness."

Our main technical results are as follows:

- We first present the axiom system BPA + SR, our extension of BPA to include state refinement, and its initial algebra.
- A term model based on bisimulation is then given for BPA + SR, and we show that BPA + SR is a sound and complete axiomatization of bisimulation for finite processes.
- We have that bisimulation is a congruence for all BPA + SR operators.
- Recursive specifications are added to BPA + SR, and we show that bisimulation remains a congruence and recursive specifications have unique solutions.

The structure of the rest of this extended abstract is as follows. Section 2 presents the equational specification BPA + SR and its term model. Section 3 adds recursive specifications to BPA +
SR. To illustrate the utility of state refinement in process algebra, Section 4 presents a succinct, hierarchical specification of an alarm clock. Finally, Section 5 concludes and points out some important directions for future work.

Further Related Work

Statecharts was given an operational semantics in [HLN+90]. In Statecharts concurrent components synchronize through the broadcasting of all actions. In order to give a semantics to the multiple simultaneous transitions that take place in a single change of state there are two notions of step. A micro-step reflects a change from one local state to another. A macro-step is a maximal sequence of micro-steps combined through the broadcast of the actions taking place in the micro-steps. The semantics presented in this paper does not include either concurrent components or broadcast actions. The introduction of a merge operator and the appropriate form of synchronization are left to a future paper.

A compositional, denotational semantics for Statecharts was presented in [HGD89]. In that paper the Statecharts language was extended to include incomplete Statecharts, or Unvollendettes. Unvollendettes have unconnected incoming transitions to which control can arrive from outside. Similarly, control can leave via unconnected outgoing transitions. The semantic domain is a space of functions that can map histories of computation to histories of computation.

The Statecharts-inspired language Argos of [Mar91] uses an action refinement–like operator (schema) that applies a set of processes $p_1, \ldots, p_n$, one each to the $n$ states of a finite automaton $q$. Such refinement is written $A_q(p_1, \ldots, p_n)$. The semantics is given in terms of reactive systems in which inputs are enabled and the automata are all deterministic.

Hoare’s CSP [Hoa85] also has an interrupt operator, whose defining equations are similar to those for mode transfer and disabling.

The scope operator of the ACSR language [LBGG94] also supports an interrupt mechanism. For processes $P, Q, R$, and $S$ and action $b$ and integer (time) $t$, the process $P \Delta^b_t(Q, R, S)$ behaves like $P$ until either it terminates, in less than $t$ seconds; its exit action $b$ occurs, in which case it behaves as $Q$; it times out at $t$ seconds and then behaves as $R$; or it is interrupted by any action of $S$. Ignoring options $Q$ and $R$, the scope operator treats processes $P$ and $S$ in much the same way that state refinement would. That is, $S[P]$ is approximately modeled by $P \Delta^b_\infty(0, 0, S)$, so long as action $b$ is not allowed to occur. Taking these restrictions into account, the SOS rules and equational laws of the scope operator are virtually identical to those of state refinement.

Refinement and recursion also occur in the Petri Box Calculus of [BDH92]. The Box Calculus is an algebra of Box expressions with many constructs in common with CSP. In particular recursion is written $\mu X. B$, and refinement is written $B_1[X \leftarrow B_2]$, where $B_1$ and $B_2$ are Box expressions. There is a homomorphism from Box expressions to the semantic domain of Boxes — equivalence classes of Petri nets. The Petri Box Model is a true concurrency model in which causality information is preserved. Recursion is treated in a standard way, and refinement should be understood to be action refinement. The expression $B_1[X \leftarrow B_2]$ replaces all occurrences of transitions labeled $X$ in the Petri net for $B_1$ with an appropriately structured Petri Net from the equivalence class for the Box $B_2$, resulting in more transitions.
Another notion of refinement in Petri Nets is place refinement [Vog90] where a place replaced by a net. Thus the connections coming to the place are now attached to the initial places of the net, and likewise for the connections from the original place. This technique does admit of a modular design methodology, but it differs from the hierarchy of states in a Statechart that we pursue in the present paper.

A previous effort by one of the authors [Use93] to give an operational semantics to Statechart-like drawings differs from the present work in two significant respects. Syntactically that algebra, Concurrent Hierarchical Automata (CHA), captured the ideas of boxes and arrows directly. Semantically, CHA permitted arrows (transitions) that crossed the boundaries of boxes (states). The current paper partly supersedes that work, since its treatment is more in keeping with traditional process algebra. But CHA, being an algebra of diagrams, does retain some interest in its own right.

2 BPA+SR

2.1 Equational Specification

BPA [BK84, BW90] (Basic Process Algebra) is an equational specification upon which many other process algebras have been constructed. In this paper, we add a state refinement operator to BPA to obtain the equational specification BPA + SR.

Let A be a set of atomic actions with constants a, b, c, ... ∈ A. The signature of BPA + SR over the sort P (for processes) is given by:

Σ(BPA + SR) = \{a → P|a ∈ A\} ∪ \{+: P × P → P\} ∪ \{·: P × P → P\} ∪ 
{[]: P → P}

As usual, the closed terms of BPA + SR are inductively defined as the constants and any finite number of applications of the operators to terms already in the language. We will use p, q, ..., possibly subscripted, to range over closed terms. Open terms are inductively generated from the constants and a collection of variables (usually lower case as in x, y, z and possibly subscripted). We will use s, t, ..., possibly subscripted, to range over open terms.

Intuitively, expressions s + t and s · t represent the alternative composition and sequential composition of s and t, while s[t] denotes s refined by t. The equational specification BPA + SR, obtained by adding three axioms for state refinement to BPA, is now given:

\[
\begin{align*}
  z + y &= y + z & \text{A1} \\
  (z + y) + z &= z + (y + z) & \text{A2} \\
  z + z &= z & \text{A3} \\
  (z + y) · z &= z · z + y · z & \text{A4} \\
  (z · y) · z &= z · (y · z) & \text{A5}
\end{align*}
\]
\[
\begin{array}{ll}
\text{SR1} & x[a] = x + a \cdot x \\
\text{SR2} & x[a \cdot y] = x + a \cdot (x[y]) \\
\text{SR3} & x[y + z] = x[y] + x[z]
\end{array}
\]

Axioms A1–5 are well-understood. The intuition behind the state refinement axioms is as follows:

- **Refinement elimination**: SR1 states that \( x \) refined by an atomic action \( a \) may choose to perform \( a \) after which its only option is to behave as \( x \), or it may choose to behave like \( x \) immediately, forgoing the opportunity to perform \( a \).

- **Refinement and sequence**: in SR2, \( x \) is now refined by \( a \cdot y \) and if \( x[a \cdot y] \) chooses to perform \( a \), then \( x \) continues to be refined, by \( y \).

- **Refinement distributes over choice**: in SR3, \( x[y + z] \) initially has the choice to proceed as \( x \) or as \( y + z \). But this is the same as choosing between \( x[y] \) and \( x[z] \).

An **associativity** law and a **quasi-idempotency** law can also be derived for state refinement by appealing to the above axioms. For closed terms \( p, q, r \), we have:

\[
p[q][r] = p[q[r]]
\]

and

\[
x + x[y] = x[y]
\]

The **mode transfer** operator of [Ber89] has axioms that, when transliterated into the syntax of state refinement, would be identical to the state refinement axioms with the exception of SR1. The mode transfer variant of SR1 is:

\[
x[a] = x + a
\]

As discussed in Section 1, mode transfer was introduced with the intention of examining the **disabling** operator of LOTOS in a process algebraic setting. Disabling provides a means of describing a process that may interrupt another process. Axiomatically, the only substantive difference between state refinement and mode transfer is that in state refinement, a terminated refining expression “just goes away” and the refined expression may proceed. With mode transfer, and hence disabling, the termination of an interruptible process precludes the interrupt from occurring.

The initial algebra \( I \) of BPA + SR is defined in the usual way. Equations A1–5 and SR1–3 say what terms are **equal**. From the techniques of **substitution**, **forming contexts**, as well as symmetry, reflexivity, and transitivity, we get a notion of what terms in BPA + SR are **provably equal**, written:

\[
\text{BPA + SR} \vdash p = q
\]

for closed terms \( p, q \in \text{BPA + SR} \). The equivalence over the terms of BPA + SR thus produced is written \( \equiv_{\text{BPA + SR}} \) and defined as the smallest relation such that:

\[
\text{BPA + SR} \vdash p = q \text{ implies } p \equiv_{\text{BPA + SR}} q
\]
Finally the set of equivalence classes of terms over BPA + SR defines the initial algebra 
\[ I = P / _{\equiv_{\text{BPA+SR}}} \]:
\[ I \models p = q \iff \text{BPA + SR} \vdash p = q \]
where satisfaction, on the left, means \( p \) and \( q \) are elements of the same equivalence class of the initial algebra.

**Theorem 2.1 (Elimination of [ ])** For any closed term \( p \in \text{BPA + SR} \) there is a closed term \( p' \in \text{BPA} \) that \( p \) can be proved equal to.

\[ \text{BPA + SR} \vdash p = p' \]

Theorem 2.1 can be proven by showing that BPA + SR, treated as a term rewriting system, is strongly normalizing and confluent with normal forms equal to BPA basic terms [BW90].

**Theorem 2.2 (Conservative Extention)** For closed terms \( p, q \in \text{BPA} \), any fact \( p = q \) that is provable in BPA + SR is also provable in BPA, and vice versa.

\[ \text{BPA + SR} \vdash p = q \iff \text{BPA} \vdash p = q \]

### 2.2 Term Model

We now consider a term model for BPA + SR based on bisimulation [Par81, Mil89]. Following [BW90], we first define the action rules of BPA + SR which collectively constitute a transition system specification (TSS), in the terminology of [GV89]. The TSS is obtained by augmenting the action rules for BPA with four rules for state refinement, and is given in Figure 1.

Bisimulation equivalence over \( P \), the closed terms of BPA + SR, is defined as follows.

**Definition 2.1** A bisimulation is a binary relation \( R \) on \( P \) such that if \( < p, q > \in R \), then:

- if \( p \xrightarrow{a} p' \) there exists a \( q' \) such that \( q \xrightarrow{a} q' \) and \( < p', q' > \in R \),
- if \( q \xrightarrow{a} q' \) there exists a \( p' \) such that \( p \xrightarrow{a} p' \) and \( < p', q' > \in R \),
- and \( p \xrightarrow{a} \bot \) if and only if \( q \xrightarrow{a} \bot \).

Two terms are bisimilar, written \( p \equiv q \), if there is a bisimulation relating them, and \( \equiv \) is the largest such relation.

**Theorem 2.3** Bisimulation is a congruence relation on \( P \) in BPA + SR.

Our term model for BPA + SR is now given by \( P / \equiv \). Furthermore, BPA + SR is a sound and complete axiomatization of bisimulation in \( P / \equiv \) for finite processes:

**Theorem 2.4**
1. $P /\models BPA + SR$

2. For all closed expressions $p, q$ over $\Sigma(BPA + SR)$:
   
   $P /\models p = q \implies BPA + SR \vdash p = q$.

Thus the term model is isomorphic to the initial algebra for $BPA + SR$. Soundness of axioms SR1–3 (part 1) can be established by formalizing the intuitive arguments given above (immediately following the statement of the axioms). Completeness (part 2) is proved by resorting to Theorem 2.1 (\cite{[]-elimination}) and the completeness of BPA with respect to finite processes [BW90].

## 3 Recursive Specifications in $BPA + SR$

### 3.1 Recursive Specifications

As in BPA, a recursive specification $E$ in $BPA + SR$ is a system of mutually recursive equations of the form:

$$\begin{align*}
X_1 &= s_1(X_1, \ldots, X_n) \\
& \vdots \\
X_n &= s_n(X_1, \ldots, X_n)
\end{align*}$$

where the $X_i$ are process variables, $X_1$ is the root variable, and $s_i(X_1, \ldots, X_n)$ is an open BPA + SR term with free variables $X_1, \ldots, X_n$. 

---

Figure 1: Transition system specification for $BPA + SR$. 

<table>
<thead>
<tr>
<th>Act:</th>
<th>$a \xrightarrow{\cdot} \sqrt{\cdot}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt1–2:</td>
<td>$\frac{z \xrightarrow{\cdot} z'}{x + y \xrightarrow{\cdot} z'}$</td>
</tr>
<tr>
<td>$\sqrt{\text{Alt1–2}}$:</td>
<td>$\frac{z \xrightarrow{\cdot} \sqrt{\cdot}}{x + y \xrightarrow{\cdot} \sqrt{\cdot}}$</td>
</tr>
<tr>
<td>Seq, $\sqrt{\text{Seq}}$:</td>
<td>$\frac{z \xrightarrow{\cdot}, z'}{x \cdot y \xrightarrow{\cdot} z' \cdot y}$</td>
</tr>
<tr>
<td>SR1–2:</td>
<td>$\frac{z \xrightarrow{\cdot} z'}{x[y] \xrightarrow{\cdot} z'}$</td>
</tr>
<tr>
<td>$\sqrt{\text{SR1–2}}$:</td>
<td>$\frac{z \xrightarrow{\cdot} \sqrt{\cdot}}{x[y] \xrightarrow{\cdot} \sqrt{\cdot}}$</td>
</tr>
</tbody>
</table>
A solution of a BPA + SR recursive equation \( X = s(X) \) is a process \( p \), in some model, by which the equation is satisfied, i.e., \( p = s(p) \) in the model. The notion of solution extends in the natural way to a recursive specification \( E \), with processes \( p_i \) substituted for the \( X_i \) giving a set of equations that are satisfied. The constant \( < X_i \mid E > \) denotes the solution to the \( i \)th equation, and \( < s \mid E > \) denotes the term \( s \) in which the solutions for the variables \( X_i \) in \( E \) are substituted for the occurrences of the \( X_i \) in \( s \).

As often happens, there can be a problem with an equation in which the variable \( X \) in the term \( s(X) \) is unguarded. \( X \) is guarded in \( s(X) \) if it occurs in some subterm \( a \cdot t \) of \( s \), and a term is completely guarded if all the variables of the term occur only guarded. Similarly, a collection of mutually recursive equations \( E \) is completely guarded if all the occurrences of all the variables in the right hand sides are guarded. Finally, \( E \) is guarded if we can use the axioms of BPA + SR and use substitution of right hand sides for their variables to rewrite the right hand sides to make it completely guarded. Thus the equations

\[
X = (a \cdot X)[b] \text{ and } X = b[a \cdot X]
\]

are completely guarded, and

\[
X = a \cdot X[Y] \\
Y = b \cdot Y
\]

is guarded but not completely guarded. It is easy to tell if a recursive specification \( E \) is guarded by looking for cycles of unguardedness in the equations. Such cycles would lead to infinite unw windings, and the absence of cycles guarantees guardedness [BW90].

### 3.2 The Term Model with Recursion

We now extend the sort \( P \) of process expressions to include the constants \( < X_i \mid E > \). We will present the additional action relations needed and extend the results of the previous section to demonstrate that bisimulation is still a congruence in the term model \( P/\equiv \). A complete axiomatization for a model including recursive specifications is not possible in general, so this section concludes by exploring the additional principles that are valid for the term model.

The TSS for BPA + SR with recursion has the additional action rules:

\[
\begin{array}{c}
\frac{< s \mid E > \xrightarrow{a} \sqrt{.} \quad < s \mid E > \xrightarrow{a} y}{X = s \in E} \\
\frac{< X \mid E > \xrightarrow{a} \sqrt{.} \quad < X \mid E > \xrightarrow{a} y}{X = s \in E}
\end{array}
\]

**Theorem 3.1** Bisimulation is a congruence relation on \( P \) in BPA + SR.

and

**Theorem 3.2** \( P/\equiv \) is a model for BPA + SR.

We also have that:
Theorem 3.3 A guarded recursive specification $E$ has a unique solution in $P/\equiv$, and it is $< X \mid E >$.

Following [BW90], the proof of this last theorem proceeds as follows:

- We first note that the term model satisfies the Recursive Definition Principle (RDP) – Every recursive specification has a solution. A solution to the recursive specification $E$ is $< X \mid E >$.

- Next we introduce the notions of finite projections and bounded nondeterminism.
  - The finite projections of a process may be expressed by (conservatively) extending BPA + SR with the projection operator $\pi_n(p)$, which stops $p$ after $n$ steps. Projection can be axiomatized in conjunction with BPA + SR, and the new axiom for state refinement is:
    $$\pi_n(x[y]) = \pi_n(\pi_n(x)[\pi_n(y)])$$

  - A process $p$ for which the predicate $B(p)$ is true has bounded nondeterminism. $p$ does not exhibit infinitely wide branching.

- These two notions allows us to state the Restricted Approximation Induction Principle AIP$^-$:
  $$\pi_n(x) = \pi_n(y) \& B(x) \implies x = y$$  \quad \forall n \geq 1$$

Our term model satisfies AIP$^-$: for a process $p$ with bounded nondeterminism, whose finite projections are all equal to those of a process $q$, we may conclude that $p$ and $q$ are themselves equal.

- The principle AIP$^-$, in turn, implies the Recursive Specification Principle RSP – a guarded recursive specification has at most one solution.

- Finally, RDP and RSP together give the above theorem.

4 Example: The Stony Brook Alarm Clock

To illustrate the utility of state refinement in process algebra, we consider a simplified version of Harel’s Citizen Quartz Multi-Alarm wristwatch [Har87], namely, the Stony Brook Alarm Clock.\footnote{The Citizen Quartz Multi-Alarm wristwatch is a product of the Citizen Watch Company of America, Inc., while the Stony Brook Alarm Clock is vaporware.}

The user interface to the alarm clock consists of a display, a buzzer, and three buttons labeled $a,b,c$, and is depicted in Figure 2.

The clock has three major modes: Time, in which the time is displayed; SetTime, in which the time can be set; and SetAlarm, in which the time at which the alarm buzzes can be set. The user can switch between the major modes by pressing the $a$-button.

Each major mode has substructure to it. Time has three secondary modes: Display, the normal mode of operation in which the time is displayed; Display&Buzz when the alarm goes off; and Blink,
Figure 2: User interface to the Stony Brook Alarm Clock.

Figure 3: State transition diagram of the Stony Brook Alarm Clock.
\[
\begin{align*}
SBAC &= \text{Time[Display]} \\
Time &= a \cdot \text{SetTime[SetHour[1]]} \\
SetTime &= a \cdot \text{SetAlarm[SetHour[1]]} \\
SetAlarm &= a \cdot \text{Time[Display]} \\
Display &= \text{wakeup} \cdot \text{Display&Buzz} + \text{power.glitch} \cdot \text{Blink} \\
Display&Buzz &= \text{buzz.off} \cdot \text{Display} \\
SetHour &= b \cdot \text{SetMinute[1]} \\
SetMinute &= b \cdot \text{SetSecond[1]} \\
SetSecond &= b \cdot \text{SetHour[1]} \\
1 &= c \cdot 2 \\
2 &= c \cdot 3 \\
\vdots \\
60 &= c \cdot 1
\end{align*}
\]

Figure 4: Recursive BPA + SR specification of the Stony Brook Alarm Clock.

which is reached after a power glitch. In state Blink, the display blinks at 12:00 and the only option left open to the user is to switch major modes.

SetTime also has three secondary modes SetHour, SetMinute, and SetSecond, in which the hour, minute, and second of the displayed time can be adjusted. The substructure of SetAlarm is identical to that of SetTime. The user switches between secondary modes by pressing the b-button.

Finally, secondary modes SetHour, SetMinute, and SetSecond have substructure to them reflecting the different numeric values the settings can take on. For example, SetHour has 12 tertiary modes reflecting the 12 possible settings (1–12) for the hour. The user switches among these tertiary modes via the c-button.

The hierarchical state transition structure of the alarm clock is depicted in Figure 3 and the corresponding BPA + SR recursive specification is given in a top-down manner in Figure 4. Note that the “twelfth” state of SetMinute and SetSecond is slightly different than the “twelfth” state of SetHour: pressing the c button in state 12 of SetHour causes a transition back to state 1, while pressing the c button in state 12 of SetMinute and SetSecond results in a transition to state 13.

5 Conclusions and Future Work

Classical process algebra allows hierarchy in system specifications through the use of nested parallel compositions. State refinement adds a new dimension to such hierarchical descriptions, viz., within a system’s state transition structure. As such, state refinement represents the first attempt, of
which we are aware, to capture the essence of statecharts within a process algebraic setting.

We have presented a technical development of state refinement (axiomatization, initial algebra, action relations, term model, unique solutions to guarded recursive specifications, etc.) within the framework of BPA with recursion, and shown that this new feature introduces no new significant technical difficulties. This is to be desired and, moreover, expected given the experience of Bergstra [Ber89] with the closely related mode transfer operator.

The next logical step in our technical development of state refinement is to introduce new operators into our setup, such as merge, restriction, and abstraction. We also plan to incorporate state refinement into the Concurrency Factory, a graphical CASE tool for concurrent systems under development at Stony Brook and N.C. State.

Acknowledgements

The authors are grateful to Gene Stark for valuable discussions on state refinement, Rob van Glabbeek for helpful pointers to the literature, and Oleg Sokolsky for creation of the alarm clock figures.

References


Parametric Preorders for Process Description Languages

Daniel Yankelevich
Dep. of Computer Science, N. C. State University
Raleigh, NC 27695-8206
email: dany@csc.ncsu.edu

Abstract

In this paper we introduce a parametric context for preorders of agents and we show how it may be used to express that an agent respects the behaviour of another one, but it is, in some sense, better.

Based on the parametric approach to concurrency semantics of [12,26], we here present a framework which provides a common basis for preorders. Taking as a base a fixed transition system and the same definition of prebisimulation, different observations yield many useful preorders: in fact, the location preorder of [6], a preorder similar to the causal preorder of [1], a preorder which refines the efficiency preorder of [3] and a preorder aiming to capture the efficiency of a protocol when a particular architecture is taken into account are defined as instances of our general framework. These relations formalize intuitive concepts such as to be more parallel than, to be more distributed than and to be more efficient than. The formal relation with the original proposals is established.

A sound an complete axiomatization of the preorder, parametric with respect to the observation chosen, is also given.

1 Introduction

Process Description Languages [20, 15] have been widely used for the specification and verification of protocols and distributed systems. They are used both to describe specifications and implementations in an abstract way.

Specifications and programs in this approach are terms of an algebra, called here agents. The semantics of these languages are either defined by means of axioms or operationally, by means of a bisimulation relation. Thus, the semantics define equivalence classes of agents.

In this approach the verification of the correctness of an implementation is usually done by proving the equivalence of the agents describing the specification and the implementation respectively. This means that all the possible implementations of a specification have to be equivalent to it. Since the semantics of concurrent processes is very fine (and hence, each equivalence class is smaller than in the sequential case), it would be mandatory, for practical applications, to use preorders instead of equivalences.

For instance, in many examples a system is specified by a nondeterministic agent and the implementation is described by an agent with many parallel components. In spite of the many advantages of truly concurrent semantics (see many papers in [13, 8]) the proof of correctness of such an implementation cannot be done using a truly concurrent equivalence. This is not a weakness of true concurrency, but rather of the use of equivalences instead of preorders.

Preorders may also be of help in a stepwise refinement strategy for the development of concurrent systems. The notion of stepwise refinement has been of use in the sequential programming field for

*Research supported in part by Hewlett-Packard, Pisa Science Center and by NSF Grant CCR-9014775
the development of complex programs and systems in a methodical way from formal specifications. The main idea is that there is an order between formal objects (programs, specifications), and that the "most abstract", initial specification can be substituted by a "more concrete" (and more efficient) program. This program may not be equivalent to the specification, but it has to be at least as good as the specification. If there are many conceptual levels, this operation can be iterated many times. In particular, if the same language is used for the specification, the design and the implementation, this technique can be applied in all the development process.

In this paper we show how preorders may be used to express that an agent respects the behaviour of another one, but it is, in some sense, better.

In general, the observer of a system is able to say that one run is better than another by performing an evaluation. For instance, if (s)he is interested on spatial distribution, (s)he could prefer a computation which is more distributed than another, even if the two computations have the same effects (in terms of sequences of actions or changes in data).

If we are interested in causality, a computation would be observed as a partial order of events [11]. In this case, the computations observed as the partial orders of Figure 1b and 1c introduce more causal constraints between events than the one observed as Figure 1a. Hence, we could say that the computation of Figure 1a is (causally) more parallel than both the other computations. Following this reasoning, we can induce a more parallel relation on CCS agents which, for instance, would say that $\alpha.\beta.NIL|\gamma.NIL$ is more parallel than $\alpha.\beta.\gamma.NIL + \alpha.\gamma.\beta.NIL$ from the point of view of causality.

There are some proposals in the concurrent field aiming to define notions of preorder capturing the intuition given above. However, each proposal lays on a different formal context, and contains some particular features (for example a particular transition system, a variant of a bisimulation equivalence or a particular language). Moreover, each approach has the explicit aim of formalizing one of the possible relations between specifications: to be more parallel than [1], to be more distributed than [6], to be an implementation of [19], to be more nondeterministic than [9], to be more defined than [25], to be more efficient than [3].

Moreover, in the author's opinion, it is quite possible that one preorder is not enough for the analysis of protocols and systems. In general, one is interested in measuring a system in different dimensions. A common context where these different notions may be formalized could permit to reuse definitions and possible general algorithms to perform the evaluation.

Process Description Languages are abstract languages, which do not consider particular network topologies, but leave open in which architecture the protocol or system is supposed to be used. In some sense, process description languages consider that all point-to-point communications are allowed, and moreover that all of them have the same complexity; while in a particular architecture some communications may be more difficult or even forbidden. Information about routing costs cannot be taken into account if the particular underlying network is not considered.

Based on the parametric approach to concurrency semantics of [12, 26], we here present a frame-
work which provides a common basis for preorders, with the explicit aim of formalizing issues such as efficiency and performance. In such a framework, many of the concepts above can be formalized for process description languages.

A general context for the definition of preorders would help in comparing different proposals, factorizing common features and singling out the differences.

A parametric approach permits to take advantage of previous work: common issues are shared and do not have to be redone. Many definitions are fixed once and for all, and one can concentrate on the particular issues of a model. Moreover, it possibly leads to general results (theorems, axiomatizations or parametric tools) for the common basis that may be re-used in each model. A parametric theory could also help in the development of new preorders or in testing new ideas, since it is possible to slightly change some parameters while leaving others fixed. The formalization of new measurements can be simplified from the use of a common description formalism.

The only proposal for a common framework to formalize different preorders is, to the best of the author's knowledge, the work of Thomsen [24], where a preorder on a process algebra is induced by preorders on actions by means of an extended bisimulation. This paper extends the work of [24] by inducing preorders on agents from preorders of observations instead of preorders of actions. This implies that it is possible to specify a preorder between the observations of whole computations, and not only between single actions. This extension is useful, because the preorder may capture some information about how events are related (for instance, causal and spatial relations) and not only about how they are labeled.

As an instance of the general methodology proposed in this paper different preorders can be obtained. Taking as a base the same (fixed) transition system and the same definition of prebisimulation (an extension of bisimulation relations to deal with preorders, similar to the extended bisimulation of [24] and to the F-bisimulation of [1]) different observations yield many useful preorders.

Such a parametric framework gives an alternative characterization of known proposals and also provides us with a context where it is possible to define and experiment different preorders between specifications just by changing the assumptions about what is interesting for a particular observer. In fact, some intuitive choices made in previous proposals can be better analyzed.

A sound an complete axiomatization of the preorder, parametric with respect to the observation chosen, is given.

Section 3 presents the main definitions for preorders.

A (very simple) observation that could be used to capture the concept of efficiency of a protocol is presented in Section 5. The preorder induced by this observation refines the efficiency preorder of [3]. A formal comparison between the two preorders is also done.

Two preorders on terms, one based on localities [6] and the other on causality [10], expressing the notion of distribution and parallelism respectively are introduced in sections 6 and 7. These examples show the usefulness of the approach, and also clarify the conceptual relation between both preorders. In this context, the preorders induced by causal observations and by location observations express the same basic idea, that of introducing more dependencies between events. This relation does not rise clearly from the original proposals, which have been done in rather different frameworks.

An example of the use of parametric preorders to formalize the notion of efficiency when a particular architecture is taken into account is given in Section 8.

Some conclusions and further work are discussed in Section 9.

Because of lack of space, we do not present here the formal proofs of the theorems. The interested reader can refer to [26], where the complete proofs are presented together with some extended discussions.
2 A Parametric Approach to Concurrency Semantics

A parametric approach to the definition of concurrent systems has been proposed in [12], which, in its general lines, can be described in four steps:

1. Define a transition system capturing the operational behaviour of the system. This operational description, which can be given by means of SOS rules [22], has to be very concrete, i.e. it has to capture all the information about transitions that the language is intended to describe.

2. Build the computations of the system as paths in the transition system, and structure them as an observation tree (ordering them by prefix).

3. Determine the observations of computations. These observations appear as labels of the nodes of the observation tree, describing the relevant aspects to be considered.

4. Define an equivalence between observation trees, based on the observations defined.

In this section, the first, the second and the fourth step are fixed once and for all. We use CCS [20] as our example language. The transition system is a variation of the original one, in which labels are enriched in order to give some information about where an action takes place. We also show how observation trees are defined and how bisimulation is defined over them, recalling definitions from [12, 21].

The first step tells us that a transition system capturing the desired operational behaviour of the system has to be defined. For CCS, such a transition system is very well known: it is the original transition system as defined by Milner [20]. However, aiming to define different observations and to reflect in an operational way information about the physical distribution of events, the labels of the transitions have to be slightly extended. The SOS definition of the transition system has the same rules as the original one, but the labels are of the form $\alpha@s$ ($\alpha$ at $s$), where $\alpha$ is an action, and $s$ is a term describing the spatial information of the event. This terms are called spatial terms, and have been introduced, together with the transition system of Definition 2, in [21].

**Definition 1** Spatial terms are defined by the following syntax: $S ::= \bullet | \emptyset | (S|S)$ and respect the following axiom: $(\emptyset|\emptyset) = \emptyset$

Intuitively, spatial terms are here used to describe information about the physical distribution of events. For instance, the term $((\bullet|\emptyset)$ describes a place having another idle component at the right, and $(\bullet|\bullet)$ describes the locality of an event which is performed simultaneously on both sides of the system. The locality of the occurrence of the action $\alpha$ in $(\alpha|\beta)|\gamma$ is described by the term $(\bullet|\emptyset)|\emptyset$.

**Definition 2** The spatial transition system for CCS is defined by the axiom and rules of Table 1. As usual, $\mu$ denotes any label, and $\alpha$ an observable label (i.e. different from $\tau$).

A computation of the transition system is a sequence of transitions $p_0 \overset{\mu_1}{\rightarrow} s_1 \overset{\mu_2}{\rightarrow} s_2 \ldots \overset{\mu_n}{\rightarrow} s_n p_n$.

In the second step of the general methodology, the computations from a state (corresponding to an agent) are ordered by prefix, generating a tree-like structure. These structures are fixed, independently of the observation, and are called observation trees. They have been introduced previously, with the name of Nondeterministic Measurement Systems (NMS) in [10]. In [12] the theory has been extended, in particular with the definition of a complete axiomatization of observational congruence for the finite case, which is parametric with respect to the observation. Different semantics are characterized according to the observations they utilize. Once the transition system has been fixed, and choosing as the equivalence the same observational equivalence, different observations yield very different semantic models [21]. The only requirement on observations is that they have to be monotonic in the following sense: if one observes a computation and obtains some information, observing a "longer" computation (i.e. a greater computation in the prefix ordering) one obtains more information.
### Spatial Transition Rules for CCS

<table>
<thead>
<tr>
<th>Act</th>
<th>Rule</th>
<th>Rel</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu. E \xrightarrow{\mu^1} E )</td>
<td>( E \xrightarrow{\mu^1} E )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\Phi} E_2 )</td>
</tr>
<tr>
<td>( \mu \notin { \alpha, \bar{\alpha} } )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\Phi} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
</tr>
<tr>
<td>Sum1</td>
<td>Sum2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
</tr>
<tr>
<td>Com1</td>
<td>Com2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
</tr>
<tr>
<td>Syn</td>
<td></td>
<td>Rec</td>
<td></td>
</tr>
<tr>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
<td>( E_1 \xrightarrow{\mu^1} E_2 )</td>
</tr>
</tbody>
</table>

#### Table 1: CCS transition system

**Definition 3** An observation **obs** is an observation domain \( \langle D, < \rangle \), where \( D \) is a set (called set of observations) and \( < \) is a partial order on \( D \), together with a monotonic function \( o \) from computations ordered by prefix to the observation domain.

We will say that \( O \in D \) is the observation of a computation \( c \) if \( o(c) = O \).

Observation domains are considered modulo isomorphism, i.e., two isomorphic observations are identified.

**Definition 4** The observation tree \( \langle N, \leq, o \rangle \) corresponding to a CCS agent \( E \) with observation **obs** is the tree of all the computations from \( E \) ordered by prefix, where each computation is labeled by its observation **obs**.

A node \( n' \) is an immediate successor of a node \( n \) if \( n < n' \) and there is no \( n'' \) such that \( n < n'' < n' \), and it is a successor if \( n \leq n' \) and for all \( n'' \) such that \( n \leq n'' \leq n' \), the observation of \( n'' \) is equal to the observation of \( n \) or to the observation of \( n' \). A node \( n' \) is a proper successor of a node \( n \) if \( n' \) is a successor of \( n \) and \( n' \neq n \).

**Definition 5** Let \( t = \langle N, \leq, o \rangle \) and \( t' = \langle N', \leq', o \rangle \) be two observation trees and \( R \) be a symmetric binary relation on \( N \cup N' \). The relation \( R \) is a bisimulation iff \( n_1 \) \( R \) \( n_2 \) implies that the observations of \( n_1 \) and \( n_2 \) coincide and that for every immediate successor \( n'_1 \) of \( n_1 \) there exists a successor \( n'_2 \) of \( n_2 \) such that \( n'_1 \) \( R \) \( n'_2 \).

Two trees are bisimilar if there exists a bisimulation such that the roots of the trees are related in the bisimulation.

### 3 Parametric Preorders

In this section we define preorders of observations trees based on preorders of observations. With this aim, each observation domain is enriched with a preorder. The intuition behind this preorder is that \( O \ll_D O' \) means that the observation \( O' \) is at least as good as \( O \). In some sense, if an observer
expects to observe \( O_i \) (s)he can also accept to observe \( O' \), but notices the difference of quality between the observations (i.e. \( O' \) has some property which makes it "better").

**Definition 6** An observation \( \text{obs} \) is an observation domain \( \langle D, <, \preceq_D \rangle \) such that \( D \) and \( < \) are as in Definition 3, and \( \preceq_D \) is a preorder on \( D \), together with a monotonic function \( o \) from computations ordered by prefix to the observation domain with the order \( < \).

This preorder on observations induces a preorder on observation trees, by means of a relation called prebisimulation. The intuition is the same as for bisimulation: the relation formalizes an experimental context. The main difference is that there is a preorder between answers to experiments. This relation is similar to the extended bisimulation of [24] and to the \( \mathcal{F} \)-bisimulation of [1].

**Definition 7** Let \( t = \langle N, \leq, o \rangle \) and \( t' = \langle N', \leq', o \rangle \) be two observation trees with observation domain \( D \) and \( R \) be a binary relation \( R \subseteq N \times N' \). The relation \( R \) is a prebisimulation iff \( n_1 \ R \ n_2 \) implies that

- \( o(n_1) \preceq_D o(n_2) \) and
- for every immediate successor \( n'_1 \) of \( n_1 \) there exists a successor \( n'_2 \) of \( n_2 \) such that \( n'_1 \ R \ n'_2 \) and
- for every immediate successor \( n'_2 \) of \( n_2 \) there exists a successor \( n'_1 \) of \( n_1 \) such that \( n'_1 \ R \ n'_2 \).

Two trees \( T \) and \( T' \) are pre-bisimilar (and we write \( T \preceq_{\text{obs}} T' \)) if there exists a prebisimulation such that the root of \( T \) is in prebisimulation with the root of \( T' \).

Once a preorder on observations is given, it induces automatically a preorder on terms.

**Notation 8** Given two agents \( E \) and \( E' \), we say that \( E \preceq_{\text{obs}} E' \) iff the \( \text{obs} \) observation trees of \( E \) and \( E' \) are pre-bisimilar.

It is easy to see that the prebisimulation relation actually induces a preorder.

**Proposition 9** For each observation \( \text{obs} \) the relation \( E \preceq_{\text{obs}} E' \) between agents is a preorder.

In [12] many equivalences between observation trees besides weak bisimulation are introduced, for example branching bisimulation and jumping bisimulation. Each of them could be extended to an associated prebisimulation. In this work, only the weak bisimulation is taken into account, but all the results can be easily extended to each of the remaining equivalences, in order to have a richer class of preorders. The relation chosen corresponds to the fourth step of the parametric approach outlined in the previous section. In this case the relation is not an equivalence but a preorder.

### 4 Proof System

In this section a complete axiomatization for the parametric preorder is given for finite observation trees. The proof system extends the parametric axioms for bisimulation introduced in [12] with one rule which lifts the preorder of the observation domain to the observation trees.

In [12] a particular syntax for representing (finite) observation trees as terms of an algebra is introduced. Given a domain of observations \( D \) (in this case a domain of labeled forests), an observation term over \( D \) is defined in the following way:

\[
P ::= \text{NIL} \mid A \cdot P \mid P \oplus P \quad \text{where } A \in D.
\]
In addition, the operations are partial, and this restriction is expressed via a type system. Typed observation terms are denoted by P, Q, G and observations (which appear in the prefixing) with A, B.

The interpretation of the operations on trees is as follows. Constant NIL denotes the empty tree. The operation A.P prefixes the tree P with a node labeled with A, and it is defined only if the root of P (if any) is labeled with an observation equal or greater than A (because the observation function has to be monotonic on computations ordered by prefix). The operation ⊕ between observation trees merges two nonempty trees by collapsing their roots, hence it is defined only if both trees have the same observation associated to the root.

These requirements are established precisely with a typing system. Constant NIL has type *, written NIL:* . If P:B and A ≤ B or B=*, then A.P:A. If P:A and Q:A, then P⊕Q:A.

The observation tree denoted by an observation term P will be written [P].

Observation terms, modulo axioms (A1), (A2) and (A3) of Table 2, are isomorphic to finite observation trees. This means that [P] = [Q] iff (A1)-(A3)-P=Q.

In [12], a complete axiomatisation for branching congruence, (weak) bisimulation congruence and strong bisimulation is given (which is parametric with respect to the observation). The set of axioms (A1)-(W3) shown in Table 2, are sound and complete with respect to weak bisimulation. That is, [P] and [Q] are observationally congruent iff (A1)-(W3)-P=Q.

As expected, prebisimulation preorders are not always precongruences with respect to the operations over observation trees. For the axiomatization we consider a precongruence, and not the preorder. This precongruence corresponds to a rooted prebisimulation.

**Definition 10** A prebisimulation R is rooted iff it relates root nodes to root nodes. That is, if R is a rooted prebisimulation between T and T' and n is the root of T and n' is the root of T', then n R n' and n R n'' implies n'' = n' and n'' R n' implies n'' = n.

Let T and T' be observation trees over an observation obs. We write T ⊨^c obs T' if there is a rooted bisimulation relating T and T'.

A rooted prebisimulation induces a precongruence with respect to the operations over observation trees, i.e. it is preserved by summation and prefixing.

**Fact 11** Let P : A and P' : A' be observation terms such that P ⊨^c_D P'. Then, if B and B' are observations such that B ≤ A and B' ≤ A' and B ≪ B' we have that B.P ⊨^c_D B'.P', and Q : A and Q' : A' implies that P ⊕ Q ⊨^c_D P' ⊕ Q'.

The proof system for precongruences of observation trees consists of a set of inequations of the form P ⊨ Q plus one inference rule. We will use implicitly some axioms and inference rules, namely the ones expressing reflexivity, transitivity, and compatibility with the constructs. Also, we use the equation P = Q to stand for the pair of inequations P ⊨ Q and Q ⊨ P.

These axioms give the basis for a rewriting-strategy to the bisimulation and the prebisimulation problem. Axioms (A1)-(A4) have been used for implementing a rewriting decision procedure for strong bisimulation in [17].

The proof of the completeness of the axiomatization is done in the vein of [14], where a standard way of axiomatizing bisimulation-like relations over labelled trees is presented, and does not follow the proof presented in [12]. Because of lack of space we do not present here the formal proof, it can be found in [26].

The next theorem shows that the axioms and rules are sound for the precongruence.

**Theorem 12** (Soundness) Let P:A, Q:B be observation terms such that P ⊨ Q. Then, [P] ⊨^c_D [Q].

The proof of completeness of the system is simplified by the fact that observation terms are already in sumform, i.e., they have sums, prefixing and NIL as the only constructs. In fact, an observation term P of type A is always of the form P = ∑ᵢ∈I A.Pᵢ.
Proof System for Preorder of Observation Trees

| (A1) | $P \oplus Q = Q \oplus P$ |
| (A2) | $(P \oplus Q) \oplus G = P \oplus (Q \oplus G)$ |
| (A3) | $A.NIL \oplus P = P$ |
| (A4) | $P \oplus P = P$ |
| (W1) | $A.B.(B.P \oplus Q) = A.(B.P \oplus Q)$ |
| (W2) | $A.P \oplus P = A.P$ |
| (W3) | $A.(B.P \oplus Q) \oplus A.P = A.(B.P \oplus Q)$ with $P:B$ |
| (Par) | \[
\frac{P \sqsubseteq Q}{A.P \sqsubseteq B.Q} \quad \text{provided that } A \ll_D B
\]

Table 2: Axioms for Parametric Preorder

**Theorem 13** (Completeness) Let $P:A$ and $Q:B$ be observation terms over a domain $D$. If $[P] \subseteq_D [Q]$, then $P \sqsubseteq Q$.

5 Efficiency Preorder

We are particularly interested on preorders reflecting a measurement of efficiency, since they can provide a formal basis to support design decisions and to choose particular implementations. The definition of these preorders depends strongly on what kind of efficiency should be taken into account. For instance, the following preorder, introduced in [3], measure the number of $\tau$ actions performed. Hence, it could be used to count the number of communications performed in a protocol.

We denote $\leq_{\text{eff}}$ the preorder introduced in [3] to formalize the notion of efficiency, where $\tau$'s are supposed to have some duration in time. Now, we will introduce an observation domain to achieve a similar (but not the same) formalization of the same intuitive concept.

**Definition 14** The domain of observation $\text{eff}$ for efficiency preorders is $\langle D, <, \ll \rangle$ where

- $D = \{ s : s \in \mathbb{N} \cdot (A \cdot \mathbb{N})^* \}$, where $\cdot$ is the constructor for sequences.
- $s < s'$ iff $s$ is a prefix of $s'$
- For $n, n' \in \mathbb{N}$, $n \ll n'$ if $n' \leq n$; and $s \cdot (\alpha \cdot n) \ll s' \cdot (\alpha \cdot n')$ if $s \ll s'$ and $n' \leq n$

Given a computation $p_0 \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} p_n$, its observation is the sequence $n_1 \cdot \alpha_1 \cdot n_2 \cdot \alpha_2 \cdot \ldots \cdot n_m \cdot \alpha_m \cdot n_{m+1}$, where $\alpha_1 \ldots \alpha_m$ is the sequence of visible actions in the computation, $n_i$ is the number of $\tau$ actions in the computation that occurs between $\alpha_{i-1}$ and $\alpha_i$, $n_1$ is the number of initial $\tau$'s and $n_{m+1}$ is the number of final $\tau$'s.

The intuition is the following: let $c_1, c_2$ be two computations. Then, $o(c_1) \ll o(c_2)$ with efficiency observations, iff $c_1$ and $c_2$ performs the same sequence of visible actions and $c_1$ has possible more $\tau$'s occurrences between two visible actions than $c_2$.

**Theorem 15** Let $E, F$ be CCS agents. Then, $E \preceq_{\text{eff}} F$ implies that $E \subseteq_{\text{eff}} F$.

The previous theorem shows that the preorder $\preceq_{\text{eff}}$ is coarser than $\leq_{\text{eff}}$. Since $\preceq_{\text{eff}}$ relates more terms, it is possible to establish more judgements of the form $E$ is more efficient than $F$ using it. The following example shows that the converse is not true.
Example 16 Let \( p = \alpha.\tau.\tau.(\beta + \tau.\gamma) \) and \( q = \alpha.\tau.(\beta + \tau.\tau.\gamma) \).

It is easy to check that \( \preceq_{\text{eff}} q \), and thus that \( q \) is more efficient than \( p \). Notice that for an external observer, the "time" (number of \( \tau \)'s) between the occurrence of any two observable actions is always minor (or equal) in \( q \) than in \( p \). However, \( p \not\subseteq_{\text{eff}} q \).

Moreover, consider the agents of the following example.

Example 17 Let \( sp = \alpha.\beta.\delta + \beta.\alpha.\delta \) and \( i = (\alpha.\gamma | \beta.\gamma.\delta) \backslash \gamma \). Then, \( i\preceq_{\text{eff}} sp \) and \( i \subseteq_{\text{eff}} sp \), while \( sp \not\subseteq_{\text{eff}} i \) and \( sp \not\subseteq_{\text{eff}} i \).

The agent \( i \) is clearly a good implementation of the specification \( sp \), since the two initial actions can be done in parallel. However, \( sp \not\subseteq_{\text{eff}} i \), showing that \( i \) is too inefficient to be chosen as an implementation of \( sp \). The claim is not that this preorder is not useful (in fact, in [3] a nice example of the use of \( \subseteq_{\text{eff}} \) is presented) but rather that in general it is necessary to take into account more than one preorder at time, i.e. to evaluate agents according to many preorders, taking care of parallelism, distribution and efficiency. For the case of this example, we will have that \( i \) is both more parallel and more distributed than \( sp \).

6 Location Preorder

Location equivalence has been presented in [5, 6] as a bisimulation based semantics able to take into account the spatial distribution of processes. The intuitive idea is that each action occurs at a location and locations may have sublocations (for example, a fork operation at some place may create two sublocations of that place). For CCS, the assumption is that in \( E|E' \) the two subprocesses \( E \) and \( E' \) are at different locations.

In addition, in [6] the authors extend the approach with a parametric location bisimulation, parametric with respect to a relation between locations. A particular relation between locations gives rise to a preorder between terms, which expresses the intuitive fact of being more distributed than. A process \( E \) is greater than \( E' \) in the preorder if \( E' \) is a shuffle of \( E \). For example, \( \alpha.a.a.\alpha \) is a sequential shuffle of \( \alpha.a.\alpha \) and then the latter is greater than the former.

An alternative characterization of location equivalence in the context of observation trees has been given in [21], where an observation domain has been presented for location observations. In this section this observation domain is extended with a preorder in order to introduce a location preorder between terms. A formal proof of the coincidence of the location preorder presented here and a version of the location preorder of [6] can be found in [26].

First, the definition of the location observation is recalled from [21]. With this aim, a partial order between spatial terms is introduced, which represents the idea of sublocality: a spatial term is a sublocality of another if the second is a refinement of the former, i.e. it can be obtained from the former by substituting the \( \bullet \) with a spatial term.

Definition 18 Let \( \ll \) be the least relation defined by the following rules. It is only defined between spatial terms with exactly one occurrence of a \( \bullet \).

\[
\begin{align*}
\bullet & \ll s \\
\frac{s_1 \ll s_2}{s_1|\emptyset \ll s_2|\emptyset} \\
\frac{s_1 \ll s_2}{\emptyset|s_1 \ll \emptyset|s_2}
\end{align*}
\]

This relation defines a tree-like partial order between spatial terms. The root is \( \bullet \) and a spatial term \( s \) is greater than \( s' \) if \( s \) is obtained from \( s' \) by replacing the dot with a spatial term. This partial order captures the intuitive idea that \( \bullet \) represents the whole system and terms of the form \( s|\emptyset \) or \( \emptyset|s \) are parts of the system, or sublocalities.

Notice that this order is only defined for spatial terms having exactly one \( \bullet \), i.e. terms associated to events occurring in one place, which do not correspond to communications.
The observation domain for localities consists of (labeled) forests of events, i.e. of structures \( (E, <, \ll, \ell) \). This observation domain is called \textbf{loc}.

**Definition 19** Given a computation \( p_0 \xrightarrow{\mu_1, e_1} p_1 \xrightarrow{\mu_2, e_2} \ldots \xrightarrow{\mu_n, e_n} p_n \), we associate to it a structure \( (E, <, \ll, \ell) \), where \( (E, <) \) is a forest of events and \( < \) is a total generation ordering on events, defined in the following way:

- \( E = \{ e_i | i = 1 \ldots n \land \alpha_i \neq \tau \} \) is a set of (at most n) different events
- \( e_i < e_j \iff (s_i \ll s_j \land i < j) \)
- \( e_i < e_j \iff i \leq j \)
- \( I(e_i) = \alpha_i \)

This observation is called \textbf{loc}.

In [21] it has been shown that the bisimulation equivalence of observation trees with \textbf{loc} observations coincides with location equivalence as defined in [6].

Now, the observation domain is enriched with a preorder between observations.

**Definition 20** The observation domain for Location Preorder is \( (D, <, \ll_{\text{loc}}) \), where \( D \) is the set of forests as defined above, \( < \) is the prefix order on forests\(^1\), and \( \ll_{\text{loc}} \) is defined in the following way:

\( (Ev, <, \ll, \ell') \ll_{\text{loc}} (Ev', <', \ll', \ell') \) iff there exists a bijection \( f : Ev \rightarrow Ev' \) preserving the ordering \( <' \) and the function \( \ell' \), and such that \( e_1 <' e_2 \Rightarrow f(e_1) < f(e_2) \)

This definition tells us that a forest is greater than another if the child-of relation of the former (seen as a set of pairs) is included in the relation of the latter. Intuitively, if \( O \ll_{\text{loc}} O' \), the forest of \( O \) introduces more constraints between the events than the forest of \( O' \), and hence \( O \) is less distributed.

**Fact 21** Relation \( \ll_{\text{loc}} \) is a preorder.

The following theorem shows the correspondence of this preorder with the location preorder of [6] (see [26]).

**Theorem 22** Let \( E, F \in CCS \). Let \( \triangleright \) be the preorder of [6]. Then, \( E \triangleright F \) if and only if \( E \ll_{\text{loc}} F \).

The following examples show that location preorder provides a very natural concept of distribution. The definition given here, involving only the sublocation relation between occurrences of events as reflected by the associated forest to each observation, is very direct and simple and does not require previous definitions or derived relations.

**Example 23** For any pair of agents \( E, E' \); \( \alpha.(E | \beta.E') + \beta.(\alpha.E | E') \ll_{\text{loc}} \alpha.E | \beta.E' \).

**Example 24** We have that \( (\alpha.(\beta + \gamma)|\beta.\gamma) \ll_{\text{loc}} (\alpha.\beta|\beta.\gamma) \beta \) but \( (\alpha.\beta|\beta.\gamma) \beta \ll_{\text{loc}} (\alpha.(\beta + \gamma)|\beta.\gamma) \beta \). The trees corresponding to these agents, labeled with observations in \textbf{loc}, are shown in Figure 2.

Using the axiomatic system of section 4, it is possible to show that \( (\alpha.(\beta + \gamma)|\beta.\gamma) \ll_{\text{loc}} (\alpha.\beta|\beta.\gamma) \beta \). In fact, as it can be seen from the picture, the observation term corresponding to the agent \( (\alpha.(\beta + \gamma)|\beta.\gamma) \beta \) is \( 0.(\alpha.a.a|\gamma \oslash a.a) \) and to the agent \( (\alpha.\beta|\beta.\gamma) \beta \) is \( 0.a.a|\gamma \), where \( 0 \) denotes the empty forest, \( \alpha \) the tree with one node labeled \( \alpha \), \( a.a \) the tree with two nodes, where \( \gamma \) is a successor of \( \alpha \), and \( a.a \gamma \) the forest with two one-node trees, one labeled \( \alpha \) and one labeled \( \gamma \).

Using the axiomatization proposed, we can prove that

\[ 0.(\alpha.a.a|\gamma \oslash a.a) \sqsubseteq \text{Par} \ 0.(\alpha.a.a|\gamma \oslash a.a) \sqsubseteq \text{W}_2 \ 0.a.a.a|\gamma \sqsubseteq \text{W}_1 \ 0.a.a|\gamma \]

\(^1O < O' \) iff \( O \) is isomorphic to a left-closed subforest of \( O' \)
7 Causal Preorder

An observation domain reflecting the causal relationship between events is introduced. The elements of this domain are mixed orders, that is, partial orders of events together with a total generation ordering.

The equivalence induced by bisimulation of observation trees with this domain is the mixed ordering bisimulation \cite{10}, sometimes called *history preserving bisimulation* or *global cause bisimulation* (however all the definitions have been shown to coincide \cite{2, 7, 18}).

First, a new relation between spatial terms is defined, which takes into account the crossed causes inherited by means of $\tau$'s. While localities are not affected by events labeled with $\tau$'s, causes can be inherited by means of communications. Hence, the sublocality relation has been extended to deal with spatial terms with more than one $\bullet$.

**Definition 25** Let $\ll_c$ be the least relation defined by the following rules.

\[
\begin{align*}
  s \neq \emptyset & \quad s_1 \ll_c s_2 & \quad s_1 \ll_c s_2 \\
  \bullet \ll_c s & \quad s_1|s' \ll_c s_2|s'' & \quad s'|s_1 \ll_c s''|s_2
\end{align*}
\]

This relation represents the idea of overlap: two spatial terms are overlapping if the events they are associated with involve some common place of the system. Hence, some “active” position of one spatial term coincides, or is a refinement of, an active position of the other spatial term. The relation tells us that there is a common location which is active in both events.

A particular case of overlapping localities is when both $s'$ and $s''$ in the rules are $s_1$ and $s_2$ are restricted to have exactly one $\bullet$: in that case the sublocality order is obtained.

Now, the observation domain for mixed order equivalence can be presented. Elements of the domain are mixed orders (labeled partial orders plus a total generation order).

**Definition 26** Given a computation $p_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \xrightarrow{\mu_n} s_n$, we associate to it a mixed labeled ordering $(E, <, \prec, t)$, where:

- $E = \{e_i|i = 1 \ldots n\}$ is a set of different events
\[ s_i \ll_c s_j \text{ and } i \leq j \implies e_i \prec e_j \]
\[ e_i \prec e_k \text{ and } e_k \prec e_j \implies e_i \prec e_j \]
\[ e_i \prec e_j \iff i \leq j \]
\[ l(e_i) = \alpha_i \]

The mo observation is given by the mixed order obtained by restricting \((E, \prec, \prec, l)\) to events with observable labels.

We extend the domain with a preorder. The intuition is that a mixed order is better (greater in the preorder) than other if it corresponds to a "more parallel" computation.

**Definition 27** The observation domain for Causal Preorder is \((D, \prec, \ll_{mo})\), where \(D\) is the class of labeled mixed orders as defined above, \(<\) is the prefix order on mixed orders, and \(\ll_{mo}\) is the least preorder on observations satisfying:

\[
(Ev, \succ, \prec, l) \ll_{mo} (Ev', \succ', \prec', l') \iff \text{there exists a bijection } f : Ev \to Ev \text{ that respects the ordering } \prec' \text{ and the labelling } l' \text{ and such that } e_1 \prec e_2 \Rightarrow f(e_1) \prec f(e_2)
\]

The preorder between mixed orders can be understood as follows: if \(O \ll_{mo} O'\), the partial order of \(O\) introduces more causal dependencies between the events than the partial order of \(O'\).

**Example 28** The following inequality holds: \(\alpha.\beta + \beta.\alpha \ll_{mo} \alpha.\beta + \alpha.\beta + \beta.\alpha\)

Notice that the definition is very similar to the preorder for localities. In fact, considering forests as a special case of partial orders, the relations \(\ll_{mo}\) and \(\ll_{loc}\) coincide. Hence, the definition of the preorder above expresses the same basic idea in two different observational contexts.

A similar preorder for CCS has been introduced in [1] by means of an \(\mathcal{F}\)-bisimulation (which resembles the prebisimulation above) and a very similar order between partial orders (expressing the concept of being more sequential than). However, both preorders do not coincide, since the one of [1] is based on the pomset mechanism for bisimulation. Actually, the preorder \(\ll_{mo}\) implies the preorder of [1], i.e. it is a coarser relation.

**Theorem 29** Let \(E, F \in CCS\). Let \(\ll_{cau}\) be the causal preorder of [1]. Then, \(E \ll_{mo} F\) implies \(E \ll_{cau} F\).

The converse of the previous theorem is not true, as shown by the following example.

**Example 30** Let \(p = \alpha.(\beta + \gamma) + (\alpha | \beta) + (\alpha | \gamma)\) and \(q = \alpha | (\beta + \gamma) + p\). Then, \(q \ll_{cau} p\), but \(p \ll_{mo} p\).

Since the \(\text{loc}\) equivalence and the \(\text{mo}\) equivalence have been shown to be different (see [18, 21, 26]), it can be expected that also location preorder and causal preorder are different, and hence they actually capture different concepts. In fact, the following example shows that \(\ll_{mo} \not\subseteq \ll_{loc}\).

**Example 31** Let \(p = (\alpha.\gamma | \tilde{\gamma}, \beta) \parallel \gamma)\) and \(q = \alpha.\beta\). Then, \(p \ll_{mo} q\) but \(p \ll_{loc} q\).

The following example shows that \(\ll_{loc} \not\subseteq \ll_{mo}\).

**Example 32** Let \(p = \alpha | \beta\) and \(q = (\alpha.\gamma | \tilde{\gamma}, \beta) \parallel \gamma + (\gamma.\alpha | \beta.\tilde{\gamma}) \parallel \gamma)\). Then, \(p \ll_{loc} q\) but \(p \ll_{mo} q\).
8 Network Topologies

In this section we show how the formal framework presented in this chapter can be instanciated in order to describe efficiency with respect to particular architectures.

The efficiency of a particular protocol or system often depends on the structure of the architecture where the protocol is supposed to be used. In CCS the assumption is that the underlying net is completely connected, and any component can communicate with any other. Whilst this assumption is useful in order to specify protocols and design systems abstractly, at some point in the development of software it is necessary to take efficiency into account, and in particular the form of the network used.

These requirements are too concrete to be included in a specification or specification language, and are better understood as an extra, a posteriori condition. Hence, we would like to keep the specification language simple and abstract and to provide a mechanism to express the network information in an orthogonal, independent way. This also implies that protocols are proved to be correct independently of the network, and after that one can choose which protocol to use for a particular network based on some formal reasoning.

The aim of this section is to include information about the underlying architecture in the observation. There are two components on the topological information:

- the description of the network, and
- the mapping from the (logical) sequential components of the specification, to the (actual) nodes of the net.

Graphs have often been used to describe computer networks or parallel architectures. However, such a description is not explicit enough for our purposes. In those descriptions, arcs of the graph represent a physical connection (a wire) between two physical nodes. We seek for a representation where all the information, also about routing of messages and communication costs can be expressed. Moreover, we want to have a description of events such as one-to-many communications. Thus we use labeled hypergraphs.

**Definition 33** A labeled hypergraph is a quadruple $G = (V, E, \text{lab}, \text{vert})$ such that $V$ and $E$ are sets (the set of nodes and edges of $G$), $\text{lab} : E \rightarrow A$ assigns to each edge of $G$ a label in the alphabet $A$, and $\text{vert}$ is a mapping associating to every edge $e$ a sequence of $k$ vertices $(v_1, \ldots, v_k)$.

We consider complete\(^2\), labeled hypergraphs whose alphabets $A$ are equipped with partial orders $\rightarrow$.

Actually, this presentation assumes a two-step approach to the specification of protocols. In one step (which may itself be divided in many stages) the usual specification with no information about topologies is done. In the second step, the relevant information about the underlying architecture is given. For the latter step, we should need a language to linearize the hypergraph structures, in order to define a specification language for this approach. It is not the aim of this section to discuss in detail this methodology, but a language to describe these graphs, with compositional properties, could be based on the graph expressions of [4].

The idea of the hypergraph definition is that the labelling describes the desired information about the underlying architecture, for instance the communication cost associated with each edge. Notice that communication in CCS is just one-to-one, and thus only simple arcs are of interest, but in general the synchronization algebra may allow one-to-many or many-to-many communications (see for instance the language CBS [23]).

**Example 34** A star architecture with four nodes is represented by the graph of Figure 3. Notice that only edges with a single source and target are included, since are the only interesting ones for

\(^2\)An hypergraph is complete iff for each sequence $v$ of nodes, there exist and edge $e$ such that $\text{vert}(e) = v$.\n
the case of CCS. Labels are used to describe information about routing and communication cost. In this example, a communication between two neighbour nodes has unitary cost, while a communication which pass through the center of the star costs 2. Arcs with two arrows represent edges in both directions with the same label. Edges with only one node (such that vert is a one element sequence) have as default label a 0.

Notice that in this way we are able to treat non connected topologies also for the case of CCS, in spite of the fact that a basic assumption of the language is that the underlying network is completely connected.

A mapping just tells us how the sequential components of an agent are distributed over the nodes of the net. Hence, each logical component is mapped into an actual node. Note that many logical components can be assigned to the same node. These mappings make sense mostly for agents describing networks of components: since logical components are here described by spatial terms, we are forced to map all the components described by the same spatial term to the same actual node. For instance, in the agent $\alpha.NIL|\beta.NIL + \gamma.NIL|\delta.NIL$, the $\alpha.NIL$ and $\gamma.NIL$ components must be mapped to the same processor. For more complex agents the formal definitions can be extended to deal also with choices. In that case, spatial terms need to be enriched with information about choices.

**Definition 35** Let $G = (V, E, lab, vert)$ be a labelled hypergraph, and let $ST_U$ be the set of spatial terms with one bullet.

A mapping is a function $M : ST_U \rightarrow V$.

**Example 36** The function $M(\emptyset \bullet) = c_1, M((\bullet \emptyset)\emptyset) = c_2, M((\emptyset \bullet)\emptyset) = c_3$, and $M(st) = c_3$ otherwise, defines a mapping from the agent $(a, \beta, \delta, \beta)|\bar{a}$ to the architecture described above.

**Definition 37** Given a labelled hypergraph $G$, the observation domain $D^G$ associated to it is $(C, <, \ll)$, such that $C$ is the set of sequences of pairs of the form $(\mu, a)$ with $\mu$ a label and $a \in A$ (we write these pairs $\mu@a$); $<$ is the prefix ordering on sequences, and $\mu_1@a_1; \ldots; \mu_n@a_n \ll \mu'_1@a'_1; \ldots; \mu'_m@a'_m$ if and only if $m = n$ and for each $i \in [1 \ldots n]$, $\mu_i = \mu'_i$ and $a_i \sim a'_i$. We call such sequences mapped computations.

Given a mapping $M$, the observation function mapping computations to mapped computations is defined in the following way: $O^M_G(\mu_1@s_1; \ldots; \mu_n@s_n) = \mu_1@lab(M(s_1)); \ldots; \mu_n@lab(M(s_n))$.

In this way, given a mapping and a graph, we have a canonical observation for them.

A strategy that can be followed to analyze the behaviour of protocols in a particular architecture is to fix the graph (which describes the information about the architecture to be analyzed), fix the
term describing the protocol, and change the observation function. Notice that for each mapping we have a different observation function. For instance, it is possible to show which mapping in a class is the best for a particular protocol and architecture or to study the behaviour of the same protocol in different networks.

**Example 38** Let us specify in CCS a protocol in which a controller sends a message to $n$ components, waits for a response and executes a concluding action. The controller is specified by the agent $\alpha^n.\beta^n.\gamma.NIL$, where $\alpha$ is the message, $\beta$ are the responses and $\gamma$ is the concluding action. Each component (suppose a failure free system) is specified as $\bar{a}.\beta.NIL$. For $n = 3$ we will formally analyze this protocol for the star architecture of the previous examples and for the architecture described by the (hyper)graph of Figure 4, an hypercube with four nodes.

The controller is mapped to the $s$ node, and the components to the nodes $c_1, c_2, c_3$. Both observation trees have nine maximal computations. In the observation tree corresponding to the star architecture all the maximal computations have observation $(\tau@1); (\tau@1); (\tau@1); (\tau@1); (\tau@1); (\tau@1); (\tau@1); (\gamma@0)$, while in the observation tree of the hypercube, there are two $\tau@2$ steps in each computation. Hence, it can be proven that the protocol behaves better in the star than in the hypercube with the given mappings.

9 Conclusions and Further Work

In this context, many other preorders can be defined just by changing the observation for computations. Part of the work is fixed once and for all, and it is not redone. Moreover, general theorems can be proved which automatically are valid for all the preorders defined in this way. For instance, sufficient conditions on the observations can be identified under which the kernel of the preorder coincides with the associated bisimulation equivalence.

The technique used in the definition of the preorders is standard and an axiomatization is also provided.

An interesting point for further research is the use of parametric preorders to compare the performance of agents in different topologies, or with different routing algorithms, etc. This research should include the application of the technique outlined in section 8 to some examples, and the extension of mappings to deal with dynamically placed processes and process migration. Notice that using the simple observations of section 8 we can already prove that a given protocol is more efficient in an hypercube than in a star network.

The framework of this work allow us to define many different preorders, experiment with them, and take the most useful, or choose in each context which observation to use. The help offered by this framework is increased with the development of parametric tools, such as the one proposed in [16], because they provide, without much effort, an automatic support for reasoning in a particular theory.
In fact, tools for efficiency measurement performing the analysis in different dimensions, constitute a potential application of this work.

This approach allows us also to combine different points of view just combining the observations associated with them. For instance, it would be easy to define a preorder capturing both causal dependencies and spatial distribution by combining the location and causal observations defined in section 6 and 7.

Acknowledgments: I wish to thank Luca Aceto, because of his encouragement and his help in the completeness proof.

References


