Embeddings Among Concurrent Programming Languages

Ehud Shapiro
Department of Applied Mathematics and Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

Preliminary Draft

July 29, 1991

Abstract

In this paper we continue an investigation into the expressive power of concurrent programming languages. In a previous paper we developed a framework for language comparison based on language embeddings, which are mappings that preserve some aspects of the syntactic and the semantic structure of the language. We provided separation results by demonstrating the nonexistence of various kinds of embeddings among languages. In this paper we complement the separation results with positive results, by demonstrating embeddings among various well-known concurrent languages. We describe the language embeddings via embeddings between transition systems. The embeddings reveal interesting connections between hitherto unrelated concurrent languages and models of concurrency. Together, the positive and negative results induce a preordering on the family of concurrent programming languages that quite often coincides with previous intuitions on the "expressive power" of these languages.

1 Introduction

In this paper we continue an investigation into the expressive power of concurrent programming languages, begun in [47]. In the previous paper we proposed relating languages by investigating the existence of compilers among them that preserve certain aspects of their syntactic and semantic structure. In particular, we have considered compilers that are:

1. Homomorphic with respect to parallel composition, i.e., compile $p \parallel q$ by compiling p and q independently and then composing the results, and

2. Preserve semantic distinctions, i.e., compile programs that are semantically distinct into programs that are semantically distinct.

We call a compiler having the first property an *embedding* and say that an embedding with the second property is *sound*.

One might expect to find a sound embedding among any two Turing-complete programming languages. It turns out this isn't so. In [47] we show that some concurrent languages cannot be soundly embedded in others. In this paper we complement the negative results with positive results: we demonstrate sound embeddings among various well-known concurrent programming languages and models. Together, the positive and negative results induce a preordering on the family of concurrent programming languages that quite often coincides with previous intuitions on the "expressive power" of these languages.

Our results are summarized in Figure 1. Although far from being complete, the emerging picture reveals interesting connections between hitherto unrelated models of concurrency.

The rest of the paper is organized as follows. Section 2 presents the framework of language comparison. Section 3 describes our notion of transition systems, their parallel composition and their observables. Section 4 lists the concurrent languages we wish to compare and explains some of their properties. Section 5 contains a menagerie of sound embeddings that justify some of the positive results shown in Figure 1. The negative results shown in the figure were proved in [47]. Section 6 discusses related work and Section 7 concludes the paper.

2 The Framework

2.1 Languages and Their Embedding

In this work we attempt to compare different concurrent programming languages, defined using different formalisms and within different semantic frameworks. In order to be able to do this we ignore some aspects of the languages and make some unifying assumptions.

The description of a sound embedding among two programming languages requires specifying the following three ingredients for each language:

1. A set of programs, P.
2. A (possibly partial) parallel composition operation $\|: P \times P \to P$.
3. A semantic equivalence relation $\simeq$ over P.

Since we do not need to know other ingredients of the languages under consideration, we abstract other details and identify a programming language L with its associated triple $(P; \|; \simeq)$. We call such a triple an *algebraic language*.

Legend:
$L \rightarrowtail L'$: there is a sound embedding of L in L', $L \preceq_{LS} L'$
$L \not\rightarrowtail L'$: there is no sound embedding of L' in L, $L' \not\preceq_{LS} L$
Properties: $\forall L, L', L''$
$L \rightarrowtail L'$, $L \not\rightarrowtail L'$, $L \leftrightarrowtail L'$ or $L \not\leftrightarrowtail L'$
$L \rightarrowtail L' \rightarrowtail L'' \Longrightarrow L \rightarrowtail L''$
$L \not\rightarrowtail L' \rightarrowtail L'' \Longrightarrow L \not\rightarrowtail L''$
$L \rightarrowtail L' \not\rightarrowtail L'' \Longrightarrow L \not\rightarrowtail L''$

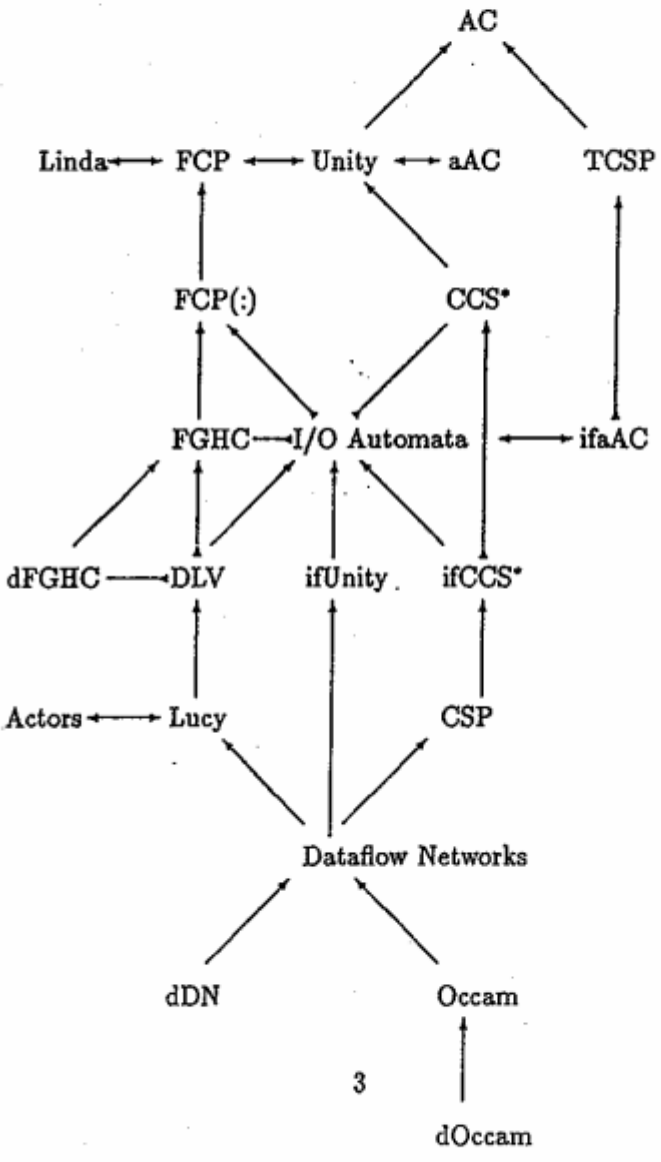

Figure 1: Languages with composition and sound embeddings among them

We adopt the following conventions. We use $p \in L$ as synonym with $p \in P$. For $p, q \in L$ we put $p \parallel q \in L$ to assert that $p \parallel q$ is defined. Equality (or equivalence) of expressions means that one side is defined iff the other side is, and if both are defined then they are equal (or equivalent). For the remaining of this paper L and L' denote languages with components $(P; \parallel; \simeq)$ and $(P'; \parallel'; \simeq')$, respectively.

Definition 1 *Let L and L' be two languages.*

- *A language embedding ε of L into L', denoted $\varepsilon : L \rightarrow L'$, is a mapping of P into P' satisfying $(p \parallel q)\varepsilon = p\varepsilon \parallel' q\varepsilon$ for every $p, q \in L$.*
- *A language embedding $\varepsilon : L \rightarrow L'$ is sound if $p \not\simeq q \Longrightarrow p\varepsilon \not\simeq' q\varepsilon$ for every $p, q \in L$.*

Mapping a concrete concurrent language into its algebraic language $(P; \parallel; \simeq)$ is immediate for the first two components, P and $\parallel$. The semantic equivalence $\simeq$ is defined as follows.

2.2 Observables and the Fully-Abstract Congruence

We assume a (possibly partial) partial function Ob that associates with each program $p \in L$ a set of observable behaviors, called *observables*. We require that $Ob(p \parallel q) = Ob(q \parallel p)$ for every $p, q \in L$, and that there is a program $c \in L$, called a *trivial program*, such that $Ob(p \parallel c) = Ob(p)$ for every $p \in L$.

The semantic equivalence $\simeq$ over L is defined to be what is known as the *fully-abstract congruence* induced by Ob and $\parallel$ [11]. Specifically, it is the largest equivalence relation satisfying $p \simeq q \Longrightarrow Ob(p) = Ob(q)$ and $p \simeq p' \ \& \ q \simeq q' \Longrightarrow (p \parallel q) \simeq (p' \parallel q')$.

We note that in [47] we studied embeddings which are sound with respect to the observable equivalence, i.e., the kernel[1] of Ob, rather than with respect to the fully-abstract congruence. It is easy to see that an embedding sound with respect to the observable equivalence is also sound with respect to the fully-abstract congruence:

Proposition 1 *Let $\varepsilon : L \rightarrow L'$ be an embedding satisfying $Ob(p) \neq Ob(q) \Longrightarrow Ob'(p\varepsilon) \neq Ob'(q\varepsilon)$. Then ε is sound.*

Proof: Let $p, q \in L$ such that $p \not\simeq q$. Then there is a program $c \in L$ such that $Ob(p \parallel c) \neq Ob(q \parallel c)$. By assumption $Ob'((p \parallel c)\varepsilon) \neq Ob'((q \parallel c)\varepsilon)$, or equivalently $Ob'(p\varepsilon \parallel c\varepsilon) \neq Ob'(q\varepsilon \parallel c\varepsilon)$, implying that $p\varepsilon \not\simeq' q\varepsilon$. □

Nevertheless, the negative results in [47] for embeddings sound with respect to the observable equivalence hold also for this more general class of sound embeddings. We chose the latter here since it is more robust in that it depends less on the fine details of the observables, and since it allows greater freedom in designing sound embeddings, as shown by Proposition 3 below.

[1] The kernel of a function f is the equivalence relation $\simeq$ satisfying $a \simeq b$ iff $f(a) = f(b)$.

2.3 Embeddings Among Transition Systems

In order to simplify our task we describe mappings among transition systems rather than among programs. This approach, which can be employed since parallel composition is the only program composition operation we consider, allows us to ignore the concrete syntax of the different programming languages. However, in order to deduce the existence of a sound embedding among languages from the existence of certain mappings among their corresponding transitions systems, some assumptions are needed.

With each programming language L we associate a set of transition systems $\mathcal{T}$ and assume an operational semantics $\sigma : P \to \mathcal{T}$ from programs to transition systems. Since Ob, the observables of L, are defined via the operational semantics we assume that Ob is defined over $\mathcal{T}$ as well and that it satisfies $Ob(p) = Ob(p\sigma)$ for every $p \in L$. For the remaining of this paper L and L' denote languages with observables Ob and Ob' and with operational semantics $\sigma : L \to \mathcal{T}$ and $\sigma' : L' \to \mathcal{T}'$, respectively.

For any language L we assume that the transition system of the program $p \parallel q \in L$ is uniform in the transition systems of the programs p and q, i.e., that there is a partial binary operation $\parallel^{\mathcal{T}}$ over $\mathcal{T}$ such that for any two programs $p, p' \in L$, with transition systems T, T', the following diagram commutes:

$$\begin{array}{ccc} p, p' & \xrightarrow{\parallel} & p \parallel p' \\ \Big\downarrow{\scriptstyle \sigma} & & \Big\downarrow{\scriptstyle \sigma} \\ T, T' & \xrightarrow{\parallel^{\mathcal{T}}} & T \parallel^{\mathcal{T}} T' \end{array}$$

Let $\simeq^{\mathcal{T}}$ denote the fully-abstract congruence over $\mathcal{T}$ induced by Ob and $\parallel^{\mathcal{T}}$. We say that L is *$\mathcal{T}$-complete* if for every $T \in \mathcal{T}$ there is a program $p \in L$ such that $p\sigma \simeq^{\mathcal{T}} T$. If L is $\mathcal{T}$-complete then $p \simeq q$ iff $p\sigma \simeq^{\mathcal{T}} q\sigma$. Hence we overload $\parallel$ and $\simeq$ to mean also $\parallel^{\mathcal{T}}$ and $\simeq^{\mathcal{T}}$, respectively, for languages complete with respect to their transition systems.

We note that in such a case the operational semantics $\sigma : L \to \mathcal{T}$ is actually a sound embedding of L into the algebraic language $(\mathcal{T}; \parallel; \simeq)$.

In the following we specify for every programming language L we consider a set of effective transition systems $\mathcal{T}$ and claim that L is complete with respect to $\mathcal{T}$. For languages given directly as transition systems, such as I/O Automata and AC, we restrict ourselves to the effective subsets of these languages.

To show the existence of a sound embedding of L into L' we will describe a sound embedding $\varepsilon : (\mathcal{T}, \parallel, \simeq) \to (\mathcal{T}', \parallel', \simeq')$. This approach is justified by:

Proposition 2 *Let L and L' be languages complete with respect $\mathcal{T}$ and $\mathcal{T}'$, respectively. If there is a sound embedding $\varepsilon : (\mathcal{T}; \parallel; \simeq) \to (\mathcal{T}'; \parallel'; \simeq')$ then there is a sound embedding ε' of L into L'.*

Proof: Assume L, L', $\mathcal{T}$, $\mathcal{T}'$, and ϵ as above. By completeness for every $p \in L$ there is a program $p' \in L'$ for which $p\sigma\epsilon \simeq' p'\sigma'$. Since $(p \parallel q)\sigma\epsilon = p\sigma\epsilon \parallel' q\sigma\epsilon$ for every $p, q \in L$ one can derive from ϵ a mapping $\epsilon' : L \to L'$ satisfying $(p \parallel q)\epsilon' = p\epsilon' \parallel' q\epsilon'$ and $p\epsilon' \simeq' p\sigma\epsilon$ for every $p, q \in L$. Such a mapping is homomorphic by construction and hence is an embedding. To see that ϵ' is sound note that $p \not\simeq q$ implies $p\sigma\epsilon \not\simeq' q\sigma\epsilon$ since both σ and ϵ are sound, which implies $p\epsilon' \not\simeq' q\epsilon'$ by construction of ϵ'. □

3 Transition Systems

3.1 Basics

Definition 2 *Assume a given set $\mathcal{S}$ of states, closed under cartesian product, and a set $\mathcal{A}$ of actions, which includes the internal action τ. A transition system over $\mathcal{S}$ and $\mathcal{A}$ is a pair $\langle I, T \rangle$ where $I \subseteq \mathcal{S}$ is the set of initial states and T, the transition relation, is a subset of $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ satisfying:*

1. $s \xrightarrow{\tau} s \in T$ *for every* $s \in \mathcal{S}$.
2. $s \neq s'$ *if* $s \xrightarrow{a} s' \in T$ *and* $a \neq \tau$.

The two requirements on transition systems are made for convenience. The requirement that transitions other than stutter change internal state is no restriction since a process spinning in one internal state can be modeled by a process flipping between two internal states.

We do not incorporate fairness constraints in our transition systems.

A *computation prefix* is a sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ of alternating states and actions, which begins with an initial state, ends with a state if it is finite, and for each i the triple $s_i \xrightarrow{a_i} s_{i+1}$ is in the transition relation. A *computation* is a computation prefix which is either infinite without a stuttering suffix, i.e., $\forall i \exists j > i$ s.t. $s_i \neq s_j$, or ends in a state from which there are no transitions other than stutter.

3.2 Parallel Composition

For each set of transition systems $\mathcal{T}$ over $\mathcal{S}$ and $\mathcal{A}$ we define the parallel composition of transition systems $\|^{\mathcal{T}}$ using two auxiliary binary operations, $\|^{\mathcal{S}}$ over states and $\|^{\mathcal{A}}$ over actions. Both operations are associative, commutative, and may be partial. We lift both to sets and define $S_1 \|^{\mathcal{S}} S_2 = \{s_1 \|^{\mathcal{S}} s_2 \mid s_i \in S_i$ and $s_1 \|^{\mathcal{S}} s_2$ is defined$\}$, for any $S_1, S_2 \subseteq \mathcal{S}$. We omit the superscripts $\mathcal{T}$, $\mathcal{S}$, and $\mathcal{A}$ when they are clear from the context.

We define $\langle I, T \rangle = \langle I_1, T_1 \rangle \parallel \langle I_2, T_2 \rangle$ to have initial states $I = I_1 \parallel I_2$ and transitions $s_1 \parallel s_2 \xrightarrow{a_1 \| a_2} s_1' \parallel s_2'$ if T_i has the transition $s_i \xrightarrow{a_i} s_i'$ for $i = 1, 2$ and all three components $s_1 \parallel s_2$, $a_1 \parallel a_2$, and $s_1' \parallel s_2'$ are defined.

Transition systems come in several basic variants:

1. **Global Actions:**

   - State composition is cartesian product.
   - Action composition is defined by $a \parallel a = a$ for every $a \in \mathcal{A}$.

   Global actions are employed in I/O Automata and in Theoretical CSP.

2. **Binary Actions:**
   The set of actions $\mathcal{A}$ admits a complementation operation satisfying $\overline{a} \neq a$ and $\overline{\overline{a}} = a$ for every $a \in \mathcal{A} \setminus \{\tau\}$.

   - State composition is cartesian product.
   - Action composition is defined by $a \parallel \overline{a} = \tau$ for every $a \in \mathcal{A} \setminus \{\tau\}$ and $a \parallel \tau = a$ for every $a \in \mathcal{A}$.

   A variant of binary synchronization is employed in CCS. The difference between the original CCS definition and ours is that we allow independent actions to occur simultaneously, whereas the original definition requires strict interleaving. The difference does not matter in our case since it neither affect the observables and hence nor the semantic equivalence. If desired, the difference can be eliminated by using different labels for truely silent actions and for internal communication actions.

3. **Binary Channels:**
   There is a function $c : S \to 2^{\mathcal{A}}$ associating with each state a subset of $\mathcal{A}$. Transitions preserve c, i.e., $s \xrightarrow{a} s' \in T \Longrightarrow c(s) = c(s')$. State composition, when defined, satisfies $c(s_1 \parallel s_2) = c(s_1) \cup c(s_2)$.

   - State composition satisfies $s_1 \parallel s_2 = \langle s_1, s_2 \rangle$ if $c(s_1) \cap c(s_2) = \emptyset$.
   - Action composition is defined as in a binary actions language.

   Binary channels are employed in Occam and CSP.

4. **Shared Store:**
   States consist of two components $S = \mathcal{Z} \times \mathcal{D}$, a local state from $\mathcal{Z}$ and a shared store from $\mathcal{D}$. Actions have the form $\mathcal{A} = \mathcal{D} \times \{\uparrow, \downarrow\}$. Transitions labeled $d^{\uparrow}$ are called *active* and those labeled $d^{\downarrow}$ are called *passive*. We use $T^{\uparrow}$ and $T^{\downarrow}$ to denote the active and passive subsets of T, respectively.

   We require that a transition $\langle s, d \rangle \xrightarrow{\ell} \langle s', d' \rangle$ satisfy $s \neq s'$ if it is active and $s = s'$ if is is passive. The intuition is that active transitions are carried by the process and passive transitions by the environment. Since these rules allow reconstructing the transition label from its source and target states the label can be omitted.

- State composition is defined by $s_1 \parallel s_2 = \langle s_1, s_2 \rangle$ for states $s_1 = \langle z_1, d_1 \rangle$, $s_2 = \langle z_2, d_2 \rangle$ if $d_1 = d_2 = d$. The resulting state is written more concisely as $\langle z_1, z_2; d \rangle$.
- Action composition is defined by $d^{\uparrow} \parallel d^{\downarrow} = d^{\uparrow}$ for every $d \in \mathcal{D}$.

Shared-store transition systems are used for Dataflow Networks, Actors, the concurrent logic languages, Linda, Unity, and AC.

In most shared-store languages "output never blocks". This property is captured by:

Definition 3 *A set of shared-store transition systems $\mathcal{T}$ is asynchronous if for every two transition systems $\langle I_1, T_1 \rangle$, $\langle I_2, T_2 \rangle \in \mathcal{T}$ and any state $\langle z_1, z_2; d \rangle \in \mathcal{S}$ if $\langle z_1; d \rangle \rightarrow \langle z_1'; d' \rangle \in T_1^{\uparrow}$ then $\langle z_2; d \rangle \rightarrow \langle z_2; d' \rangle \in T_2^{\downarrow}$.*

All the shared-store languages, except AC, are asynchronous.

Some shared-store languages are "loosely coupled" [33] in the sense that active transitions from two concurrent processes sharing the same store can always commute. This property is captured by:

Definition 4 *A set of shared-store asynchronous transition systems $\mathcal{T}$ is loosely coupled if for every two transition systems $\langle I_1, T_1 \rangle$, $\langle I_2, T_2 \rangle \in \mathcal{T}$ and any state $\langle z_1, z_2; d \rangle \in \mathcal{S}$ if $\langle z_1; d \rangle \rightarrow \langle z_1'; d_1 \rangle \in T_1^{\uparrow}$ and $\langle z_2; d \rangle \rightarrow \langle z_2'; d_2 \rangle \in T_2^{\uparrow}$ then there is $d' \in D$, uniform in d_1 and d_2, such that $\langle z_1; d_2 \rangle \rightarrow \langle z_1'; d' \rangle \in T_1^{\uparrow}$ and $\langle z_2; d_1 \rangle \rightarrow \langle z_2'; d' \rangle \in T_2^{\uparrow}$. Otherwise we say that $\mathcal{T}$ is tightly coupled.*

Note that if there is a partial order on $\mathcal{D}$ such that active transitions never decrease the shared store then the requirement that the store value d' be uniform in d_1 and d_2 in the definition above induces a least upper bound operation on D.

Dataflow Networks, Actors, the concurrent logic languages Lucy, DLV, and FGHC are loosely coupled. They have natural partial orderings and least upper bound operations which are consistent with our definitions.

Linda, Unity, AC, and the concurrent logic languages FCP(:) and FCP are tightly coupled.

3.3 Observables

For each set of transition systems $\mathcal{T}$ we define a (possibly partial) function Ob that associate with each computation prefix c of a transition $T \in \mathcal{T}$ an observable outcome. For a transition system T, we define $Ob(T)$, the *observables of* T, to be $\{Ob(c) \mid c$ is a computation prefix of T and $Ob(c)$ is defined $\}$.

Observable outcome in action-based transition systems is defined for computations. The outcome $Ob(c)$ of a computation c is a pair consisting of a termination mode and a trace, which is the sequence of actions of the transitions it employed, with internal actions removed.

The observable outcome of a shared-store transition system T is defined for computations in which the environment made no transitions, i.e., for computations of $T^{\dagger}$. It consists of a termination mode and an abstraction of the limit value of the shared store. Both the termination mode and the abstraction used are language specific.

4 Concurrent Programming Languages

In this section we present a catalog of the languages we consider. Due to space limitations we describe them only informally.

For simplicity we assume that programs do not have internal parallelism, i.e., $\|$ is used only at the top level, although many of our results do not seem to be affected by lifting this restriction. The restriction implies that the set of transition systems of a language can be fully specified by defining transition systems for sequential programs and defining the rule for parallel composition, which is what we do.

1. DN: Nondeterministic Dataflow Networks (cf. [23] for the original deterministic model, and [21, 39] for fully-abstract semantics for nondeterministic dataflow networks). Dataflow networks operate on streams, which are two-port asynchronous FIFO channels with an internal unbounded buffer. Transition systems for a dataflow networks employ a shared store records channel histories. They are asynchronous and loosely coupled.

   In DN, CSP Occam, and DLV each transition system has an associated set of input and output ports. In these languages the parallel composition of two transition systems is defined iff they do not own a common port.

   The store abstraction used in the observables of T hides the history of the internal channels of T.
2. CSP (cf. [18] for the original definition and [4] for an investigation of various sublanguages). CSP processes communicate using synchronous channels, which are two-port unbuffered channels in which sending and receiving must occur simultaneously. Transition systems for CSP are action based and employ static binary synchronization. Closure conditions ensure that a transition system cannot preview the value to be received on an input channel.
3. Occam [20]. Occam is similar to CSP. The main difference is that nondeterministic choices in CSP can be guarded by both input and output actions, whereas in Occam they can be guarded by input actions only. This is reflected in the transition system for sequential programs by the

requirement that if from a given state an "output" transition is enabled, then no other transition is enabled at that state.

4. Actors [16]. The Actors model consists of processes, called actors, communicating via mailboxes. A mailbox is a many-to-one asynchronous channel, without a guarantee for order of message arrival. Each actor has a unique identifier and may "know" the identifiers of other actors, which are said to be its acquaintances. An actor a can send a message to b's mailbox only if b is an acquaintance of a. A message may include actor identifiers known to the sender, allowing the receiver to become acquainted with these actors as well.

   Actors are modeled using state-based transition systems, whose shared store records the messages sent to each mailbox.

   The parallel composition of two transition systems is defined iff they do not share a common actor.

   The abstraction used in the observables hides all internal communication and leaves visible only messages sent to outside actors.
5. Lucy [24] is a concurrent constraint language designed to be comparable to actors. Its only compound structure is a many-to-one bag and only output ports to a bag can be sent in messages. Its design is successful in the sense that from a transition system point of view it is essentially identical to actors.
6. DLV: Processes communicating via directed logic variables [25] (e.g. Doc [17] and a subset of Janus [43]). A directed logic variable is a two-port communication channel that can transmit at most one message, which may contain embedded ports. Unlike Actors and Lucy, both input and output ports can be sent in messages. A state-based transition system for directed logic variables is given in [25].

   The store abstraction hides the values given to internal channels.
7. FGHC: Flat Guarded Horn Clauses [50]. Processes communicating using shared logic variable, which are single-assignment variable that take terms, which may contain variables, as values. The shared store consists of a set of equality constraints. Here and in the following concurrent logic languages parallel composition is totally defined. The store abstraction projects the final constraint on the initial variables.
8. FCP: Flat Concurrent Prolog [44, 45]. (See [53] for a definition of FCP(:,?), a cleaned-up variant of that language.) A concurrent logic language employing atomic unification and read-only variables.
9. FCP(:) [42, 53]. A variant of FCP with atomic unification but no read only variables, initially presented with a different syntax under the name FCP($\downarrow$, |)[41].
10. IOA: I/O Automata [28] employ global synchronization over a set of actions A. Each transition system T has an associated set of actions $T^A \subseteq A$,

and has a transition $s \xrightarrow{a} s$ for any state s and any $a \in A \backslash T_A$. The parallel composition of two transition systems T_1 and T_2 is defined iff $T_1^A \cap T_2^A = \emptyset$.

11. CCS* [31, 32]. CCS consists of processes employing dynamic binary synchronization. Parallel composition for CCS is totally defined.

    Since the official semantic equivalence for CCS cannot be derived naturally from observable outcomes of computations we refer to the language that we study as CCS*. It is closely related to the language EPL of [15].
12. Linda [7]. Processes communicating by inserting and deleting tuples in a shared Tuple Space. Transition systems for Linda are state-based, with the shared store consisting of sets of tuples.
13. SV: Processes communicating by reading from and writing to shared variables [27] (*e.g.*, Unity [9]). Their transition systems are state-based, with the shared store being the vector of values of the shared variables.
14. AC [13]: Processes communicating via a shared store that takes values from a partially ordered set. The value of a store can only be increased in the ordering.
15. TCSP: Theoretical CSP [19]. TCSP employs global synchronization. Its parallel composition operation is totally defined.

In addition, we are interested in subsets of some languages. These are defined specifically for each language, but have some properties in common:

- *dL*: The deterministic subset of L.
- *ifL*: The interference-free subset of a language L.

The deterministic subset *dL* of L is the set of L programs in which each sequential process is deterministic, has at most one transition emanating from each state. Note that *dL is* closed under the parallel composition operation of L and that the parallel composition of deterministic processes may exhibit nondeterministic behavior. We investigate dDN, dFGHC, and dCSP.

In some languages parallel composition is only partially defined to avoid naming conflicts, to ensure that communication is one-to-one, or to allow only a single writer per variable. A common aspect of these restrictions is captured by the following property, called interference freedom (a slightly more general definition of this property is given in [47]).

Definition 5 *Let $L = (S; \|; \simeq)$ be a language. A program $p \in L$ is trivial if for every $q \in L$ such that $(p \parallel q) \in L$, $(p \parallel q) \simeq (q \parallel p) \simeq q$.*

We say that L is interference free if for every nontrivial program $p \in L$, $(p \parallel p) \notin L$.

Some languages investigated here, e.g. Dataflow Networks, Occam, CSP, Actors, Lucy, DLV, and I/O-automata, are interference free. In addition, we investigate the following interference-free sublanguages:

16. ifCCS*: The sublanguage of CCS* in which the parallel composition of processes is allowed if and only if they employ a disjoint set of actions.
17. ifSV: The single-writer sublanguage of SV (called, in the case of Unity, the read-only sublanguage).
18. ifAC: The sublanguage of AC in which parallel composition of two transition systems T_1, T_2 is allowed if and only if the store augmentations of their active subsets $T_1^\dagger$, $T_2^\dagger$ are disjoint.

We also investigate two other sublanguages of AC:

19. aAC, the asynchronous sublanguage of AC.
20. ifaAC, the interference-free sublanguage of aAC.

5 A Menagerie of Sound Embeddings

We now present some justifications for the arrows in Figure 1

5.1 Observable Behavior and Sound Embeddings

If $\simeq$ is the fully-abstract congruence induced by the observables, as in our case, then the following proposition can be used to establish that an embedding is sound. The proposition is useful in that it allows establishing the soundness of an embedding without referring explicitly to the fully-abstract congruence.

Proposition 3 *Let L and L' be languages with semantic equivalences being the fully abstract congruences induced by the observable functions Ob and Ob', respectively, and let $\varepsilon : L \to L'$ be an embedding. Assume that for every $p, q \in L$ such that $Ob(p) \neq Ob(q)$ there is a program $c' \in L'$ such that $Ob'(p\varepsilon \parallel' c') \neq Ob'(q\varepsilon \parallel' c')$. Then ε is sound.*

Proof: Assume L, L' and ε satisfy the conditions of the proposition, and let $p, q \in L$ be programs such that $p \not\simeq q$. By the definition of $\simeq$ there is a program $c \in L$ such that $Ob(p \parallel c) \neq Ob(q \parallel c)$. By assumption there is a program $c' \in L'$ such that $Ob'((p \parallel c)\varepsilon \parallel' c') \neq Ob'((q \parallel c)\varepsilon \parallel' c')$, which can also be written as $Ob'(p\varepsilon \parallel' c\varepsilon \parallel' c') \neq Ob'(q\varepsilon \parallel' c\varepsilon \parallel' c')$. This implies that $p\varepsilon \not\simeq' q\varepsilon$, establishing that ε is sound. □

Definition 6 *Define $\preceq$ to be the relation satisfying $L \preceq L'$ iff there is a sound embedding of L in L' and $L \sim L'$ to be a shorthand for $L \preceq L'$ & $L' \preceq L$.*

In the following embeddings silent transitions are mapped to the corresponding silent transitions, and hence they are not mentioned explicitly.

5.2 Embedding Interference-Free CCS* in I/O Automata

In this section we relate two abstract models of concurrency, CCS* and I/O Automata. There is no sound embedding of CCS* in I/O Automata, as shown in [47], since the latter is interference free whereas the former isn't. Here we show, however, that ifCCS*, the interference-free subset of CCS*, can be embedded in I/O Automata. The technique we use was first employed in an embedding of CSP in FCP [40, 45]

For a set $\mathcal{A}$, $2^{\mathcal{A}}$ denotes the set off finite subsets of A and $\mathcal{A}^*$ denotes the set of finite sequences over $\mathcal{A}$. For sequences x and y over $\mathcal{A}$ (where elements of $\mathcal{A}$ are identified with singleton sequences) $x \diamond y$ denotes the concatenation of y to x if x is finite, and is undefined otherwise.

Let p be a CCS program with transition system $\langle S, I, A, T\rangle$. Let $D_{\mathcal{A}}$ denote the set of sequences over $\mathcal{A} \cup 2^{\mathcal{A}}$, ordered by prefix. We call an element of $2^{\mathcal{A}}$ an *offer* and an element of $D_{\mathcal{A}}$ a *ready trace*.

Definition 7 *Outstanding offer, enabled action, admissible ready trace*
Let $d = r_1, r_2, \ldots, r_n$ be a ready trace. An offer r_i is outstanding in d if for no action a s.t. $\overline{a} \in r_i$, $a = r_j$, $j \geq i$.

An action a is enabled in d if $\overline{a} \in r_i$ for some outstanding offer r_i in d.

We say that d is admissible if every action $a = r_i$ in d is enabled in the ready trace $d_i = r_1, r_2, \ldots, r_{i-1}$.

We define $p\varepsilon$ to be the I/O Automaton with transition system $\langle S \times 2^{\mathcal{A}} \times D_{\mathcal{A}}, I \times \emptyset \times \Lambda, D_{\mathcal{A}}, T'\rangle$, where T' is defined as follows.

Let $s \in S$ be a state with $s \xrightarrow{a_i} s_i$ for $1 \leq i \leq n$ being the only transitions from s in T, and let $\vec{a} = \{a_1, a_2, ..., a_n\}$. Then for every ready trace d and every i, $1 \leq i \leq n$, T' has the following transitions:

Output transitions:

- Offer (Offers $\vec{a}$ if no a_i is enabled):

  $\langle s, \emptyset, d\rangle \xrightarrow{\vec{a}} \langle (s, \vec{a}, d \diamond \vec{a})$, where no a_i is enabled in d.

- Act$_i$ (Do a_i if $\overline{a_i}$ is enabled):

  $\langle s, \emptyset, d\rangle \xrightarrow{a_i} \langle s_i, \emptyset, d \diamond a_i\rangle$ if $\overline{a_i}$ is enabled in d.

Input transitions:

- Accept$_i$ (Accept choice of $\overline{a_i}$ if a_i was previously offered by self):

  $\langle s, \vec{a}, d\rangle \xrightarrow{\overline{a_i}} \langle s_i, \emptyset, d\rangle$

- Input: In addition, for any state s, offer $\vec{a}$, and ready trace d, we have:

  $\langle s, \vec{a}, d\rangle \xrightarrow{a} \langle s, \vec{a}, d \diamond a\rangle$

  where a is either an input action such that that $\overline{a} \notin \vec{a}$ or an input offer.

Note that only admissible ready traces can be generated by a computation. Note also that if the actions that can be taken by $p, q \in CCS^*$ are disjoint (implying that their parallel composition in ifCCS* is defined) then the actions that can be taken by $p\varepsilon, q\varepsilon$ are also disjoint, implying that the paralel composition of $p\varepsilon, q\varepsilon \in IOA$ is defined.

We claim that if $p, q \in CCS$ are observably distinct then so are $p\varepsilon$ and $q\varepsilon$. This is shown via a mapping from computations of the I/O automaton to computations of the corresponding CCS program, which maps an Act_i and an $Accept_i$ transition to a CCS transition synchronizing on a_i and maps *Offer* and *Input* to stutter.

Proposition 4 $if CCS^* \preceq IOA$.

5.3 Embedding I/O Automata in Theoretical CSP

I/O Automata and TCSP are similar in using global synchronization. The differences between these models is that in I/O Automata one process initiates an action and the others must follow suit. In TCSP all processes must agree in order for an action to take place.

However, it turns out that the identity mapping from IOA to TCSP overcomes this difficulty, and is indeed a sound embedding. The interference freedom of IOA ensures that only when the output transition is enabled then the corresponding synchronous action in TCSP will occur.

This may justify the view that, ignoring syntax and fairness, IOA is a special case of TCSP.

Proposition 5 $IOA \preceq TCSP$.

5.4 Embedding I/O Automata in Unity and in ifaAC

Let p be an I/O Automaton with transition system $\langle S, I, A, T\rangle$. Then $\varepsilon : IOA \rightarrow Unity$ maps p to the Unity program $p\varepsilon$ with the transition system $\langle S', I', T'\rangle$, where $S' = Z \times A^*$, $Z = S \times A^*$, $I' = I \times \Lambda \times \Lambda$. That is, states of a Unity program have the structure $\langle s, d; d'\rangle$, where s is the internal state, d is the value of the local variable and d' is the value of the shared variable. T' has the following transitions, for every state $s \in S$ and sequence $d \in A^*$:

- **Output:** $\langle s, d; d\rangle \rightarrow \langle s', d \diamond a; d \diamond a\rangle$ if there is an output transition $s \xrightarrow{a} s' \in T$.
- **Receive:** $\langle s, d; d'\rangle \rightarrow \langle s', d \diamond a; d'\rangle$ if $d' = d \diamond a \diamond d''$ for some action a and sequence d'' and T has an input transition $s \xrightarrow{a} s'$.
- **Input:** $\langle s, d; d'\rangle \rightarrow \langle s, d; d \diamond a\rangle$ for every internal state (s, d), store value d' and $a \in A$.

We claim that if $p, q \in IOA$ are observably different then so are $p\varepsilon$ and $q\varepsilon$.

Proposition 6 *IOA* $\preceq$ *Unity.*

It turns out that the same mapping ε constitutes an embedding of I/O Automata in AC. Note that the image of the embedding is actually included in ifaAC, the interference-free, asynchronous sublanguage of AC, since an active transitions of $p\varepsilon$ augments the store by an action taken by p, and actions taken by composed I/O automata are disjoint by definition of the model of I/O Automata.

Proposition 7 *IOA* $\preceq$ *ifaAC.*

5.5 Embedding ifaAC in I/O Automata

Let p be an $ifaAC$ program with transition system $\langle S, I, T\rangle$, with $S = Z \times D$. Define $aug(T) = \{(d, d') | \langle s; d\rangle \rightarrow \langle s'; d'\rangle \in T\}$. Then for any $T \in ifaAC$, $T\varepsilon$ is the I/O Automaton $\langle S, I, aug(T), T'\rangle$, where the output actions are $aug(T^{\uparrow})$, the input actions are $aug(T^{\downarrow})$, and T' has transitions $\langle s; d\rangle \stackrel{(d,d')}{\rightarrow} \langle s'; d'\rangle$ for every $\langle s; d\rangle \rightarrow \langle s'; d'\rangle \in T$.

It is easy to see that active transitions of T are mapped to output transitions of the I/O automaton $T\varepsilon$ and passive transitions of T to input transitions of $T\varepsilon$.

Proposition 8 *ifaAC* $\preceq$ *IOA.*

Corollary 1 *IOA* $\sim$ *ifaAC.*

5.6 Embedding I/O Automata in FCP(:)

The programming techniques used here to embed I/O Automata in FCP(:) and of Unity in FCP(:,?) are explained in [45].

Let p be an I/O Automaton with transition system $\langle S, I, A, T\rangle$. Then $\varepsilon : IOA \rightarrow FCP(:)$ maps p to the FCP(:) program $p\varepsilon$ with the clause

```
p(S,As) :- S=s, As=[a|As'] : true | p(s',As').
```

for every input transition $s \stackrel{a}{\rightarrow} s' \in T$, and the clause

```
p(S,As) :- S=s : As=[a|As'] | p(s',As').
```

for every output transition $s \stackrel{a}{\rightarrow} s' \in T$. Such a program may be infinite, but by assumption on the effectiveness of the transition system for p its an equivalent finite program.

5.7 Embedding Unity in aAC

The embedding $\varepsilon : Unity \rightarrow aAC$ maps a Unity program p with transition system $\langle S, I, T \rangle$, $S = Z \times D$ to the aAC transition system $\langle S', I', T' \rangle$ with $S' = Z \times D^*$, $I' = I \times \Lambda$, and T' having the transition $\langle s; d \diamond v \rangle \rightarrow \langle s'; d \diamond v \diamond v' \rangle$, for every $d \in D^*$ and every transition $\langle s; v \rangle \rightarrow \langle s'; v' \rangle \in T$.

Proposition 9 *Unity* $\preceq$ *aAC.*

The opposite embedding of aAC in Unity is the indetity mapping, establishing:

Proposition 10 *Unity* $\sim$ *aAC.*

We were unable to construct an embedding of sound embedding of AC in Unity or in aAC.

5.8 Embedding Unity in FCP(:,?)

The embedding $\varepsilon : Unity \rightarrow FCP(:,?)$ maps Unity program p with transition system $\langle S, I, T \rangle$, $S = Z \times D$ to the FCP(:,?) program with the clause:

```
p(S,Vs) :- Vs=[v|As'] : true | p(S,As').
```

for every passive transition $\langle s; v \rangle \rightarrow \langle s; v' \rangle \in T$, and the clause:

```
p(S,Vs) :- S=s : As=[V?|As'] | V=v, p(s',Vs').
```

for every active transition $\langle s; v \rangle \rightarrow \langle s'; v' \rangle \in T$. Such a program is infinite in general, but by assumption on the effectiveness of the transition system for p it is equivalent to a finite program.

5.9 Embedding CCS* in aAC

The embedding $\varepsilon : CCS^* \rightarrow AC$ is very similar to the embedding of ifCCS* in I/O Automata. It takes a CCS program p with transition system $\langle S, I, A, T \rangle$ to the aAC transition system $\langle S', I', T' \rangle$, with $S' = S \times D_A$, where D_A is the domain of ready traces over A defined above, $I' = I \times \Lambda$, and T' having the following transitions:

For every state $s \in S$, where $s \xrightarrow{a_i} s_i$ for $1 \leq i \leq n$ are the only transitions from s in T, $\vec{a} = \{a_1, a_2, ..., a_n\}$, and every ready trace d:

- Offer (Offers $\vec{a}$ if no a_i is enabled):
  $\langle s, \emptyset; d \rangle \rightarrow \langle s, d \diamond \vec{a}; d \diamond \vec{a} \rangle$ if no a_i is enabled in d.
- Act_i (Do a_i if it is enabled):
  $\langle s, \emptyset; d \rangle \rightarrow \langle s_i, \emptyset; d \diamond a_i \rangle$ if a_i is enabled in d.

- **Accept$_i$** (A silent transition that changes the internal state if one of the actions offered by the process was taken by the environment):

  $\langle s, d'; d\rangle \rightarrow \langle s_i, \emptyset; d\rangle$ if $d = d' \diamond d''$ and $\overline{a_i}$ occurs in d''.

- **Input:** In addition, for every internal state (s, d'), store value d and every single-element legal store augmentation x, we have:

  $\langle s, d'; d\rangle \rightarrow \langle s, d'; d \diamond x\rangle$

The effect of this implementation is that every synchronous CCS transition is realized by three AC transitions: Offer, Act and Accept.

We claim that if $p, q \in CCS^*$ are observably distinct then there is an aAC program c for which $p\varepsilon \parallel c$ and $q\varepsilon \parallel c$ are observably distinct in aAC.

Proposition 11 $CCS^* \preceq AC$.

5.10 Embedding Theoretical CSP in AC

The embedding $\varepsilon : TCSP \rightarrow AC$ maps a TCSP program p with transition system $\langle S, I, A, T\rangle$ into an AC transition system $\langle S', I', T'\rangle$ where $S' = S \times A^*$, $I' = I \times \Lambda$, and T' has the following transitions for every $s \xrightarrow{a} s' \in T$:

- **Initiate:** $\langle s, d; d\rangle \rightarrow \langle s', d \diamond a; d \diamond a\rangle$,
- **Participate:** $\langle s, d; d\rangle \rightarrow \langle s, d; d \diamond a\rangle$, and
- **Update:** $\langle s, d; d \diamond a\rangle \rightarrow \langle s', d \diamond a; d \diamond a\rangle$

The **Initiate** transition allows a process in internal state s to initiate action a. The **Participate** passive transition allows a process in internal state s to passively participate in a, without changing its internal state, if some other process initiates it, and then use the **Update** silent transition to update its internal state from s to s'.

We claim that ε maps observably distinct TCSP programs to observably distinct AC programs.

Proposition 12 $TCSP \preceq AC$.

We were unable to construct a sound embedding of AC in TCSP.

6 Other Known Embeddings and Non-Embeddings

6.1 Known Embeddings

Various sound embeddings are known from the literature (although not under this name, and not proven formally within our framework). An embedding of

Dataflow Networks in Theoretical CSP is shown by Josephs, Hoare, and Jifeng [22]. The same method applies to embedding Dataflow Networks in CSP. It cannot, however, be used to embed DN in Occam, since Occam cannot implement unbounded buffers. An embedding of Dataflow Networks in Occam is given by de Boer and Palamidessi [3]. An embedding of Dataflow in ifUnity (i.e., the read-only subset of Unity) is given by Chandi and Misra [9]. It seems that another method employed there can be used to embed CSP in ifUnity.

The technique used by Milner to embed the value passing calculus of CCS in pure CCS [32] can be applied to embedding CSP in ifCCS*. There is a straightforward embedding of Dataflow Networks in concurrent logic languages. Even the weakest languages in the family, those employing Directed Logic Variables, admit this embedding. An embedding of Actors in Lucy is given by Saraswat et al. [24]. There is a simple embedding of Lucy's bags in terms of Directed Logic Variables. There is a known embedding of DLV in FGHC, which has been used in an actual implementation of a subset of Janus in FGHC. A straightforward embedding of FGHC in FCP is given in [45]. That survey includes other embeddings among concurrent logic languages which are not considered here. An embedding of Linda in FCP is described in [46]. Embeddings of Dataflow Networks, Shared-Variable languages, and concurrent logic languages in the model of Asynchronous Concurrency were studied by Gaifman, Maher, and Shapiro [13].

6.2 Non Embeddings

SV in ifSV

Single-Writer Shared Variables (ifSV) are interference-free, whereas Shared Variables aren't. This precludes sound embeddings of ifSV in SV by our earlier results [47]. An implementation ε of SV in ifSV that realizes a shared variable by a process that manages it is not an embedding, since if p is a program that writes on some variable x, $p\varepsilon$ must write on some variable, say y. Assuming $(p \parallel p)\varepsilon = p\varepsilon \parallel p\varepsilon$ contradicts the interference-freedom of ifSV.

SV in IOA

I/O Automata are interference-free, whereas Shared Variables aren't. This precludes sound embeddings of SV in IOA [47]. In particular the implementation ε of shared variables in I/O Automata, shown by Goldman and Lynch [14], is not an embedding, since it is not homomorphic with respect to parallel composition.

SV in CCS

Milner [32] shows an implementation of a shared variable language in CCS. The embedding is homomorphic in a very restricted sense, which does not cover SV. Concurrent processes can communicate only via variables declared outside their

scope. They can interact with the outside world only via two fixed variables, which are identical for all programs. Hence independent programs cannot be composed in a meaningful way. Attempting to extend the embedding to allow 'open' composition, by allowing arbitrary external variable names, would not work since it is not clear with which agent to associate the process implementing a variable. Having multiple copies of a variable process would result in an unsound embedding. The embedding of a shared-variable language with "open" composition in CCS is possible if the language is restricted to be single-writer, i.e., to ifSV. In this case the embedding associates the variable process with the (unique) agent that writes on it.

CSP vs. CCS

Brookes [6] describes an embedding of CSP in CCS-like synchronization trees. Of course there is also an embedding of CCS in synchronization trees. However, it seems that from these two embeddings one cannot reconstruct an embedding of CSP in CCS, or vice versa.

6.3 Other Related Work

A review of some previous work on the subject, including [26, 32, 22, 26, 36, 37, 49, 8, 30, 35, 45, 1, 2, 10, 38, 4], was given in [47]. Two recent references not mentioned there are [34, 52]. Mitchell [34] introduces a notion of language translation which is essentially the same as the notion of fully-abstract embeddings described in [47], and uses it to demonstrate that Lisp is not universal in the sense that it cannot embed any language having an abstraction context, i.e., a context that its application to observably distinct programs may result in equivalent programs.

Vaandrager [52], following earlier work by de Simone [48], uses structural operational semantics to compare I/O Automata to other algebraic models of concurrency. We have yet to understand the relationship of this line of research to ours.

7 Conclusion and Future Work

The work presented here is preliminary in many ways:

- There are many missing arrowheads in Figure 1. Each represents an open problem. Our tentative conjectures as to which direction the missing arrowheads will point are reflected by the relative height of the languages in the figure.
- Many important languages and models of concurrency are missing from our figure. Incorporating them into the framework is a job that remains to be done.

- We have not considered operators other than ||. In particular, hiding in some languages is incorporated in parallel composition and in others it is a separate operator. Allowing hiding as an additional operator seems a natural extension.
- We have ignored fairness in our treatment. Incorporating the different fairness criteria into the comparison may refine our understanding of the different models.
- In [47] we have introduced a separation schema and applied it in a limited way. Additional applications of the schema are needed in order to separate languages which we have failed to prove equivalent.

Acknowledgements

Comments by Frank de Boer, David Harel, Haim Gaifman, Elia Lalovich, Yael Moscowitz, Catuscia Palamidessi, Amir Pnueli, and Joseph Sifakis are gratefully acknowledged. Part of this work was done while the author was visiting ICOT, the Institute for New Generation Computer Technology.

References

[1] de Boer, F.S., and Palamidessi, C., Concurrent Logic Programming: Asynchronism and Language Comparison, *Proc. of 1990 North American Conference on Logic Programming*, S. Debray and M. Hermenegildo (Eds.), MIT Press, pp.175-194, 1990.

[2] de Boer, F.S., and Palamidessi, C., Embedding as a Tool for Language Comparison, CWI Technical Report, 1990.

[3] de Boer, F.S., and Palamidessi, C., Embedding as a Tools for Language Comparison: On the CSP Hierarchy, To appear in *Proc. of CONCUR'91*, LNCS, Springer, 1991.

[4] Bougé, L., On the Existence of Symmetric Algorithms to Find Leaders in Networks of Communicating Sequential Processes, *Acta Informatica*, 25, pp. 179–201, 1988.

[5] Brock, J.D., and Ackerman, W.B., Scenarios: A Model of Non-Determinate Computation, in Diaz and Ramos (eds.), *Formalization of Programming Concepts*, Lecture Notes in Computer Science 107, pp. 252–259, Springer-Verlag, 1981.

[6] Brookes, S.D., On the Relationship of CCS and CSP, *Proc. of 10th Colloq. on Automata, Languages and Programming*, Lecture Notes in Computer Science 154, Springer-Verlag, pp.83-96, 1983

[7] Carriero, N., and Gelernter, D., Linda in Context, *Comm. ACM*, 32(4), pp. 444–458, 1988.

[8] Chandra, A.K., and Manna, Z., The Power of Programming Features, *J. Computer Languages*, 1, pp. 219–232, 1975.

[9] Chandy, K.M., and Misra, J., *Parallel Program Design*, Addison-Wesley, 1988.

[10] Felleisen, M., On the Expressive Power of Programming Languages, *Proc. ESOP'90*, N. Jones (Ed.), LNCS 432, pp. 134–151, Springer-Verlag, 1990.

[11] Gaifman, H., and Shapiro, E., Fully Abstract Compositional Semantics for Logic Programs, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 134–142, 1989.

[12] Gaifman, H., Maher, M.J., and Shapiro, E., Reactive Behavior Semantics for Concurrent Constraints Logic Programs, in Lusk, E.L., and Overbeek, E. (eds.), *Proc. 1989 North American Conference on Logic Programming*, pp. 553–572, MIT Press, 1989.

[13] Gaifman, H., Maher, M.J., and Shapiro, E., Replay, Recovery, Replication, and Snapshots of Nondeterministic Concurrent Programs, To appear in *Proc. PODC'91*, ACM,1991.

[14] Goldman, K.J., and Lynch, N.A., Modelling Shared State in a Shared Action Model, *Proc.* 5^{th} *Annual Symposium on Logic in Computer Science*, pp. 450–463, IEEE, 1990.

[15] Hennessy, M., *Algebraic Theory of Processes*, MIT Press, 1988.

[16] Hewitt, C., A Universal, Modular Actor Formalism for Artificial Intelligence, *Proc. International Joint Conference on Artificial Intelligence*, 1973.

[17] Hirata, M., Programming Language Doc and its Self-Description, or, $X = X$ is Considered Harmful, *Proc.* 3^{rd} *Conference of Japan Society of Software Science and Technology*, pp. 69–72, 1986.

[18] Hoare, C.A.R., Communicating Sequential Processes, *Comm. ACM*, 21(8), pp. 666–677, 1978.

[19] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, New Jersey, 1985.

[20] INMOS Ltd., *OCCAM Programming Manual*, Prentice-Hall, New Jersey, 1984.

[21] Jonsson, B., A Fully Abstract Trace Model for Dataflow Networks, 16^{th} *Annual ACM Symposium on Principles of Programming Languages*, pp. 155–165, 1989.

[22] Jifeng H., Josephs, M.B., and Hoare, C.A.R., A Theory for Synchrony and Asynchrony, *Proc. of IFIP TC 2 Working Conf. on Programming Concenpts and Methods*, Sea of Gallilee, Israel, April, 1990.

[23] Kahn, G., The Semantics of a Simple Language for Parallel Programming, *Information Processing 74*, pp. 993–998, North-Holland, 1977.

[24] Kahn, K., and Saraswat, V.A., Actors as a Special Case of Concurrent Constraint (Logic) Programming, Xerox Technical Report, 1990.

[25] Kleinman, A., Moscowitz, Y., Pnueli, A., and Shapiro, E., Communication with Directed Logic Variables, *Proc. of ACM POPL*, 1991.

[26] Landin, P.J., The Next 700 Programming Languages, *Comm. ACM*, 9(3), pp. 157–166, 1966.

[27] Lynch, N.A., and Fischer, M.J., On Describing the Behavior and Implementation of Distributed Systems, *TCS*, 13, pp. 17–43, 1981.

[28] Lynch, N.A., and Tuttle, M.R., Hierarchical Correctness Proofs for Distributed Algorithms, *Proc. ACM Symposium PODC'87*, 1987.

[29] Manes, E.G., *Algebraic Theories*, Graduate Texts in Mathematics 26, Springer-Verlag, 1976.

[30] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[31] Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[32] Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989.

[33] Misra, J., Loosely-Coupled Processes, in E.H.L. Aarts, J. van Leeuwen, M. Ram (eds.), *Proc. of PARLE'91: Parallel Architectures and Languages Europe*, Vol. 2, LNCS 506, Springer, pp.1-26, 1989.

[34] Mitchell, J., On Abstraction and the Expressive Power of Programming Languages, To appear in *Proc. of the Int'l Conference on Theoretical Aspects of Computer Science*, Sendai, Japan, LNCS, Springer, 1991.

[35] Paterson, M.S., and Hewitt, C.E., Comparative Schematology, *Conf. Rec. ACM Conference on Concurrent Systems and Parallel Computation*, pp. 119–127, 1970.

[36] Reynolds, J.C., GEDANKEN – A simple typeless language based on the principle of completeness and the reference concept. *COmmunication of the ACM*, Vol 13 No. 5, pp.308-319, 1970.

[37] Reynolds, J.C., The essence of Algol, in *Algorithmic Languages*, de Bakker and van Vliet (Eds.), North-Holland, Amsterdam, pp.345-372, 1981.

[38] Riecke, J.G., Fully Abstract Translations between Functional Languages (Preliminary Report), *Proc. of ACM POPL*, 1991.

[39] Russel, J.R., Full Abstraction for Nondeterministic Dataflow Networks, *Proc.* 30^{th} *IEEE FOCS*, pp.170-175, 1989.

[40] Safra, S., Partial Evaluation of Concurrent Prolog and its Implications, M.Sc. Thesis, Technical Report CS86-24, Dept. of Computer Science, Weizmann Institute of Science, 1986.

[41] Saraswat, V.A., Partial Correctness Semantics for CP[↓,—,&], *Proc. 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, New-Delhi, LNCS 205, Springer, pp.347-368, 1985.

[42] Saraswat, V.A., A Somewhat Logical Formulation of CLP Synchronization Primitives, in Bowen, K., and Kowalski, R.A. (eds.), *Proc. 5th International Conference Symposium on Logic Programming*, pp. 1298–1314, MIT Press, 1988.

[43] Saraswat, V.A., Kahn, K., and Levy, J., Janus: A Step Towards Distributed Constraint Programming, *Proc. 1990 North American Conference on Logic Programming*, S. Debray and M. Hermenegildo (Eds.), MIT Press, 1990.

[44] Shapiro, E. (Editor), *Concurrent Prolog: Collected Papers*, Vols. 1 & 2, MIT Press, 1987.

[45] Shapiro, E., The Family of Concurrent Logic Programming Languages, *ACM Computing Surveys* 21(3), pp. 412–510, 1989.

[46] Shapiro, E., Embedding Linda and other joys of concurrent logic programming, Technical Report CS89-07, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1989.

[47] Shapiro, E., Separating Concurrent Languages with Categories of Language Embeddings, *Proc. STOC'91*, ACM, pp.198-208, 1991.

[48] De Simone, R., Higher-Level Synchronizing Devices in MEIJE-SCCS, *Theoretical Computer Science*, Vol. 37, pp.245-267, 1985.

[49] Sussman, G.J., and Steele, G.L., Jr., Scheme: An Interpreter for Extended Lambda Calculus, Memo 349, MIT AI Lab., 1975.

[50] Ueda, K., *Guarded Horn Clauses*, Ph.D. Thesis, Information Engineering Course, University of Tokyo, Tokyo, 1986.

[51] Ueda, K., and Furukawa, K., Transformation Rules for GHC Programs, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 582–591, 1988.

[52] Vaandrager, F.W., On the Relationship Between Process Algebra and Input/Output Automata, Proc. *LICS'91*, IEEE, pp.387-398, 1991.

[53] Yardeni, E., Kliger, S., and Shapiro, E., The Languages FCP(:) and FCP(:,?), *J. New Generation Computing*, 7, pp. 89–107, 1990.

Ehud Shapiro

Office:

Department of Applied Mathematics
and Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel
Tel: (972)-8-483327
udi@wisdom.weizmann.ac.il

Home:

53b Nachmani St.
Tel-Aviv, Israel

Tel: (972)-3-625695

Personal

Born February 28, 1955 in Jerusalem, Israel; married + 1; Israeli citizen.

Education

Ph.D. in computer science, May 1982, Yale University.

B.A./B.Sc. in mathematics and philosophy, with distinction, June 1979, Tel Aviv University.

Employment

October 1984 – present:	Senior Scientist, Department of Applied Mathematics and Computer Science, Weizmann Institute of Science.
June 1982 – October 1984:	Post-doctoral Fellow, Department of Applied Mathematics, Weizmann Institute of Science.
October 1973 – April 1977:	Service in the Israeli Defense Forces.

Research Interests

Logic programming. Concurrent programming. Programming languages, techniques, methodologies, semantics, and tools. Parallel and distributed computer systems, architectures, and algorithms. Inductive inference.

Honors

Ph.D. thesis selected as an 1982 ACM Distinguished Dissertation.