A fully abstract model for concurrent logic languages

Boer, de, F.S.; Palamidessi, C.

Published: 01/01/1990

Citation for published version (APA):

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
A fully abstract model for concurrent logic languages

by

F.S. de Boer    C. Palamidessi

90/16

October, 1990
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.
A Fully Abstract Model for Concurrent Logic Languages

Frank S. de Boer¹ and Catuscia Palamidessi²

¹Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
email: wsinfdb@tuewsd.win.tue.nl

²Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy
email: katuscia@dipisa.di.unipi.it

²Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
email: katuscia@cwi.nl

September 7, 1990

Abstract

One of the main aims of this paper is to show that the nature of the communication mechanism of concurrent logic languages is essentially different from the classical paradigms of CCS and TCSP. We define indeed a compositional semantics based on linear sequences, whilst more complicated structures, like trees and failure sets, are needed to model compositionally CCS and TCSP. Moreover, we prove that this semantics is fully abstract, namely that the information encoded by these sequences is necessary.

Our observation criterium consists of all the finite results, i.e. the computed constraint together with the termination mode (success, failure, or deadlock). The operations we consider are the parallel composition of goals and the union of neatly intersecting programs. We define a compositional operational model delivering sequences of input-output constraints, and we obtain a fully abstract denotational semantics by requiring additionally some closure conditions, that model the monotonic nature of communication in concurrent constraint languages.


General terms: Languages, Theory.

Additional key words and phrases: concurrent logic languages, constraints, operational semantics, denotational semantics, compositionality, full abstraction.

1 Introduction

This paper addresses the problem of a compositional and fully abstract semantics for concurrent logic languages. Compositionality is considered one of the most desirable characteristics of a formal semantics, since it provides a basis for program verification and modular design. The difficulty in obtaining this property depends upon the operators of the language, the behaviour we want to describe (observables), and the degree of abstraction we want to reach. A compositional model is

*Part of this work was supported by the ESPRIT BRA projects "Integration" and "SPEC".*
called *fully abstract* (with respect to some operators and observables) if it identifies programs that behave in the same way under all the possible contexts. A fully abstract model can be considered to be the semantics of a language since all the other compositional semantics can be reduced to it by abstracting from the redundant information. Full abstraction is important, for instance, to decide correctness of program transformation techniques. If a fully abstract model distinguishes the transformed program from the original one then the transformation is not correct (in the sense that it does not preserve the same behaviour under composition).

The basic operators of a logic language are the conjunction of goals and the union of clauses. The observables usually consist of the termination mode (success or failure), and the computed answer substitution. For concurrent logic languages compositionality has been studied mainly with respect to the conjunction of goals, whilst union of clauses has been considered only in the simple case of *neatly intersecting programs* [9]. This is rather natural since in a concurrent framework the main operation is the parallel composition of processes. On the other hand, the class of observables has to be enriched by deadlock.

The compositional description of deadlock is one of the main semantic problems of concurrent languages. For languages like CCS and TCSP it is well-known that (linear) sequences are not sufficient. On the other hand, trees encode redundant branching information. In order to abstract from it two main approaches have been proposed. One is based on equivalence relations on trees (for instance, bisimulation [16]), and the other on grouping the branching information in sets (for instance, *failure sets* [2]). In general, failure set semantics is more abstract than bisimulation and it is proved to be fully abstract for TCSP and CCS.

With respect to compositionality, concurrent logic languages have been regarded just as a particular case of the classic paradigms. Therefore, the problem has been approached by the standard methods. De Bakker and Kok [3, 14] and De Boer et al. [4, 5] use tree-like structures labeled with functions on substitutions. More simple tree-like structures, labeled by constraints, are used by Gabrieielli and Levi [10] and by Saraswat and Rinard [19], who also define an equivalence based on bisimulation. Gerth et al. [9] and Gaifman et al. [11] approach the problem of full abstraction by refining the failure set semantics of TCSP.

We think that concurrent logic languages require a different approach. In this paper we study the language defined in [11], that can be considered as a special case of concurrent constraint programming [18, 19]. This paradigm represents a considerable improvement with respect to "classical" concurrent logic languages. The notion of constraint [13] allows on one side to increase the expressiveness, and, on the other side, to model in a logical manner the synchronization mechanism [15, 18]. As discussed in [18], the notion of store in constraint programming leads naturally to a new paradigm for concurrent programming. All processes share a common store, that represents the constraint established until that moment. Communication is modeled by adding (telling) consistently some constraint to the store. Synchronization is achieved by checking (asking) if the store entails (implies) a given constraint, if not, the process suspends.

It is interesting to compare this logic paradigm with CCS. We can translate CCS by interpreting the action $a$ as telling the constraint $x = a$, and the complementary action $\bar{a}$ as asking if $x = a$ is entailed by the store. The main difference is that complementary actions do not synchronize anymore. Indeed, telling a constraint will never suspend. In other words, the communication mechanism of concurrent logic languages is intrinsically *asynchronous*. The following example shows that this leads to an essentially different deadlock behaviour.

**Example 1.1** Let $p_1 = \bar{a}b + \bar{a}c + \bar{a}d$ and $p_2 = \bar{a}b + \bar{a}(\bar{c} + \bar{d})$. In any compositional semantics for CCS these two processes must be distinguished. Indeed, they behave differently under the context $p = a(b + c)$. The process $p_1$ can deadlock, by choosing the third branch, whilst $p_2$ cannot. In the formalism of [11], this example can be translated as follows.

\[
\begin{align*}
\{ & p_1(x, y) \leftarrow \text{ask}(x = a) \mid \text{ask}(y = b). \\
& p_1(x, y) \leftarrow \text{ask}(x = a) \mid \text{ask}(y = c). \\
& p_1(x, y) \leftarrow \text{ask}(x = a) \mid \text{ask}(y = d). \}
\end{align*}
\]
In this translation, both $p_1$ and $p_2$ have the same behaviour. The process $p_2$ can deadlock by choosing the second clause, because $p$ can independently decide to produce $y = b$ (after $x = a$). Figure 1 illustrates this example.

![Figure 1: In logic programming $p_1$ and $p_2$ cannot be distinguished by $p$.](image)

Actually, in concurrent logic languages we cannot express a context "strong" enough to distinguish between $p_1$ and $p_2$ above. The reason is that, due to the asynchronous nature of `tell`, the choice guarded by `tell` is a local choice. This is shown by the following example.

**Example 1.2** In an asynchronous reading of CCS along the line of the translation given above, $p_1 = a(b + c)$ is equivalent to $p_2 = ab + ac$ under every context. After the production of $a$, $p_1$ can proceed to produce either $b$ or $c$ in the same way as $p_2$ does.

This example may induce to believe that simple sequences of constraints are sufficient for obtaining compositionality. This is not the case, because the choice guarded by `ask` is a global choice.

**Example 1.3** Even in the asynchronous case, the process $p_1 = \bar{a}(\bar{b} + c)$ is not equivalent to $p_2 = \bar{a}\bar{b} + \bar{a}c$. They are distinguished by the context $p = ab$ ($p_2$ can deadlock whilst $p_1$ cannot).

However, the nature of the global choice in concurrent logic languages is essentially different from the one of CCS. Indeed, it only depends upon the result of the past behaviour of the system, i.e. upon the constraint contained in the store.

This remark indicates a possible way to solve the problem of compositionality. Given a sequence of constraints representing the computation of a process with respect to an arbitrary environment, we add the information about who is the producer of each constraint, either the process or the environment. If the store determined by such a sequence does not provide the process with the necessary information to proceed then the process will deadlock, assuming that the environment does not produce any constraint anymore. The composition of different processes then simply amounts to verifying that the assumptions made by one process about its environment are indeed validated by the other processes.
We define the compositional semantics of a process by means of a transition system. The configurations consist of a process and a store, represented as a sequence of constraints. A transition of the environment is modeled by adding an input constraint to the store. This kind of transition, that does not occur in the usual description of CCS, allows here to obtain a compositional operational semantics based on (sets of) sequences ended by a termination mode. These sequences are essentially different from the scenarios of [17], where input substitutions correspond solely to assumptions about the environment which are necessary for the process to proceed. As a consequence, compositionality is there obtained only for the success set. The input-output sequences we use have been introduced in [9] as one component of the domain of the denotational semantics, the other ingredient being the suspension set. Because of what is stated above, this suspension set could have been reduced to a simple termination mode.

The language described in [9] contains non-monotonic test predicates. However the real intricacies of the asynchronous and declarative nature of communication in logic languages come to surface in the monotonic case. In this case even the sequences contain too much information. Indeed, they encode the order and the granularity in which constraints have been produced, details that cannot be sensed by monotonic contexts. This is mainly due to the fact that monotonic contexts cannot be specified to ask (only) a specific constraint, they can always proceed when stronger constraints are provided. More in general, the reaction of any context is invariant with respect to the logical equivalence of sequences of constraints produced by the process. Therefore, the final step to achieve full abstraction will consist of some closure conditions that represent this equivalence.

To our knowledge, the first proposal of a compositional semantics based on linear sequences for concurrent logic languages has been given in [7, 8]. Those papers, however, deal only with languages based on substitutions. Moreover, the model we present here is more elegant, since the hiding of local variables is formalized in terms of existential quantifiers. In this framework, the closure conditions are more easy to formulate and have a clear logical intuition. As a consequence, a transparent and structured proof of the correctness and the full abstraction of the denotational model can be given. As far as we know, this is the first time that the proof of a full abstraction result for concurrent logic languages is presented.

This paper is organized as follows. In the next section we present the language. In section three we define a compositional operational semantics based on a transition system. In section four we introduce a more abstract denotational model, the correctness of which is proved in section five. In the last section we prove that this denotational model is fully abstract. In order not to interrupt the main flow of the paper we have delegated some delving into underworldly technicalities to the appendices.

2 The language

A constraint system is any system of partial information that supports the notions of consistency and entailment. For the sake of simplicity we consider here constraint systems based on first-order languages, however our results can be extended in a straightforward way to arbitrary constraint systems which support existential quantification. Let $V$ be a set of variables with typical elements $x, y, \ldots$, let $F$ be a set of function symbols $a, b, \ldots, f, g, \ldots$, and let $P$ be a set of predicate symbols. Furthermore, let $\Sigma = (V, F, P)$. A constraint system $\Gamma$ is a first-order theory in $\Sigma$. Given the formulas $\phi, \psi$, and $\phi_1$, we say that $\phi$ is consistent if $\Gamma \models \exists \phi$, where $\exists \phi$ denotes the existential closure of $\phi$, and that $\phi_1$ entails $\phi_2$ if $\Gamma \models \phi_1 \Rightarrow \phi_2$. A simple constraint $\theta$ is a quantifier-free formula in $\Sigma$. The set $\text{Con}$ of constraints, with typical element $c$, consists of formulas of the form $\exists X \theta$, where $X$ is a set of variables. The constraints of the form $\exists(x) \theta$ and $\exists \varnothing \theta$ will be denoted by $\exists x \theta$, $\theta$, respectively. For any formula $\phi$ we define $FV(\phi)$ to be the set of the free variables of $\phi$, and $BV(\phi)$ to be the set of the bound variables of $\phi$. We assume $\Gamma$ to be fixed, so we will omit references to $\Gamma$.

We now describe the concurrent logic language based on $\Gamma$ along the lines of the one introduced in [11]. Let $\text{Pred}$ be a set of predicate symbols disjoint from $P$. The set of atoms $\text{Atom}$ in $V, F,$
*Pred*, with typical element *A*, *B*, is defined as usual. *Tell* and *Ask* are the sets of constructs of the form *tell*(*)\(i\)), *ask*(*)\(i\)), respectively. A *program* is a finite set of clauses of the form

\[ p(\vec{x}) \leftarrow \exists Y \ g_1 : g_2 | \vec{B}. \]

where \( p \in \text{Pred}, \vec{x} \) is a sequence of variables, \( g_1 \in \text{Ask}, \ g_2 \in \text{Tell}, \ \vec{B} \) is a multiset of atoms, and \( Y \) is the set of variables occurring in \( g_1, g_2, \vec{B} \) and disjoint from \( \vec{x} \). The atom \( p(\vec{x}) \) is the *head* of the clause, \( g_1 \) and \( g_2 \) together form the *guard*, and \( \vec{B} \) is called the *body*.

The variables of \( Y \) are the *local* variables of the clause. We will omit the symbol "\( \rightarrow \)" when either \( g_1 \) or \( g_2 \) is not present, and omit \( \exists Y \) when \( Y \) is \( \emptyset \). The set of programs will be denoted by \( \text{Prog} \).

A *goal* is an object of the form \( \leftarrow A \), where \( A \) is a multiset of atoms. The set of goals will be denoted by \( \text{Goal} \). The union of the multisets \( A \) and \( B \) will be represented by \( A, B \).

An *instantiation* of a clause \( p(\vec{x}) \leftarrow \exists Y \ g_1 : g_2 | \vec{B} \) is a clause of the form \( p(\vec{t}) \leftarrow \exists Z \ g'_1 : g'_2 | \vec{B'} \), where \( g'_1, g'_2 \) and \( \vec{B'} \) are obtained from \( g_1, g_2 \) and \( \vec{B} \) by simultaneously replacing every occurrence of a variable of \( \vec{x} \) by its corresponding term of the sequence \( \vec{t} \) and every occurrence of a variable of \( Y \) by its corresponding variable of \( Z \). The set of new local variables \( Z \) is assumed to be disjoint from the set of variables of \( \vec{t} \). Given a program \( W \) the set of all the instantiations of its clauses we denote by \( \text{Inst}(W) \).

Given a program \( W \) the operational model of a goal \( \leftarrow A \) can be described as follows: The basic computation step is defined with respect to the store, the accumulated simple constraint, and consists of checking if the store entails a certain constraint and then adding a constraint. More specifically, given a store \( s \), a computation step consists of a selection of an atom \( A \) of \( \leftarrow \vec{A} \) and a clause \( A \leftarrow \exists X \ \text{ask}(\theta_1) : \text{tell}(\theta_2)|\vec{B} \) of \( \text{Inst}(W) \), where \( X \) has no variables in common with \( s \), such that

1. \( s \) entails \( \exists X \theta_1 \),
2. \( s \land \theta_1 \land \theta_2 \) is consistent.

Then \( A \) is replaced by \( \vec{B} \) in the goal \( \leftarrow \vec{A} \) and \( \theta_1 \land \theta_2 \) is added to the store.

An atom \( A \) *fails* if for every clause of \( \text{Inst}(W) \) with head \( A \) the second condition does not hold.

If the second condition holds but the first fails and for no other clauses the two conditions are satisfied then the atom *suspending*.

A goal fails if it contains an atom which fails and it suspends if all its atoms suspend.

The result of a terminating computation consists of the final store where all the variables not occurring in the initial goal are existentially quantified, together with the termination mode: *success, failure*, or *deadlock*, if the final goal is empty, fails, or suspends, respectively.

### 3 A compositional operational semantics

In this section we enrich the informal model of the previous section to obtain a compositional semantics, namely, the definition of the meaning of a goal in terms of its subgoals. To this purpose we describe the behaviour of a goal as a sequence of interactions with its environment. Interactions are modeled as *input/output* constraints. An input constraint is provided by the environment, whereas an output constraint is produced by the goal itself.

**Definition 3.1**

- The set of input constraints is \( \text{Con}_I = \{ c^I : c \in \text{Con} \} \).
- The set of output constraint is \( \text{Con}_O = \{ c^O : c \in \text{Con} \} \).
- The set of input/output constraints, with typical element \( c^I \), is \( \text{Con}_{IO} = \text{Con}_I \cup \text{Con}_O \).

\(^1\)Usually a body (as well as a goal) is defined as a sequence of atoms. For our purposes, however, it will be sufficient to represent it as a multiset.
Given a program $W$, the operational semantics we define is based on a transition system $T = (\text{Conf}, \rightarrow_w)$. We will omit the symbol $W$ when no confusion is possible. The configurations $\text{Conf}$ are pairs consisting of a goal (for the sake of convenience we drop the symbol $\_\_\_\$) and a finite sequence of input/output constraints $s$ ($s \in \text{Con}_{IO}$). We will associate a store with such a sequence in the following way:

**Definition 3.2** We define

\[
\begin{align*}
\text{Store}(\lambda) & = true \\
\text{Store}(\exists X \theta^f \cdot s) & = \emptyset \land \text{Store}(s)
\end{align*}
\]

Here $\lambda$ denotes the empty sequence.

Furthermore we will make extensively use of the following definition of the constraint we obtain from a sequence when abstracting from the labels, interpreting the sequencing operator as conjunction and taking into account the scope of the quantifiers:

**Definition 3.3** We define

\[
\begin{align*}
\text{Estore}(\lambda) & = true \\
\text{Estore}(\exists X \theta^f \cdot s) & = \exists X (\emptyset \land \text{Estore}(s))
\end{align*}
\]

The representation of a store as a sequence of (existentially quantified) constraints, as well as definitions similar to those of $\text{Store}$ and $\text{Estore}$, have also been used in [12] to model Andorra.

It will turn out to be technically convenient to introduce the following notions:

**Definition 3.4** Given a sequence $s$ we define

- $FV(s)$, the free variables of $s$ (the global variables),
- $BV(s)$, the bound variables of $s$ (the local variables),
- $BV^\ell(s)$, $\ell \in \{I,O\}$, the bound variables of $s$ occurring in constraints labeled by $\ell$. So the local variables introduced by the process itself are given by $BV^O(s)$ and those introduced by the environment by $BV^I(s)$,
- $\text{var}(s)$, the variables of $s$.

We can now define the transition system. Table 1 describes the rules for $T$ relative to the "successful" computation steps. We call them computation rules. The first rule models a transition by the process, whilst the second rule models a transition by the environment. The condition $FV(c) \cap BV^O(s) = \emptyset$ formalizes the requirement that the local variables of a process are hidden from the environment. Note that we allow $A$ to be the empty goal $\emptyset$. This models the possibility that the process has terminated whilst the environment still continues to produce constraints. The last rule describes the behaviour of a goal as the interleaving of its subgoals.

Table 2 and table 3 illustrate the rules for failure and suspension respectively. We need to introduce in our configurations the symbols $\text{fail}$ and $\text{susp}$, with the obvious meaning.

Note that if we drop from $T$ the rule $C2$, we obtain a transition system that essentially formalizes the informal model of the previous section.

The reason why we represent a store in $T$ as a sequence of (existentially quantified) constraints is that this allows to express in an elegant way the appropriate closure conditions for full abstraction (see section 4). If we were only interested in compositionality, then sequences of simple constraints would have been sufficient.

The operational semantics $O$ based on this transition system $T$ delivers sets of sequences $s$ of input/output constraints, ended by a termination mode. We denote the set of these sequences as $\text{Seq} = \text{Con}_{IO}\{\text{ss}, \text{ff}, \text{dd}, \bot\}$. The set $\text{Con}_{IO}^*$ denotes the sequences of constraints generated during the computation, whilst the symbols $\text{ss}$, $\text{ff}$, and $\text{dd}$ represent the possible ways in which a process can terminate: success, failure and deadlock, respectively. Sequences ending in $\bot$ denote
Table 1: The Transition System $T$. Computation Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>$(A; s) \to (B; s, (\exists X \vartheta_1 \land \vartheta_2)^O)$ if $\exists A \leftarrow \exists \text{ask}(\vartheta_1) : \text{tell}(\vartheta_2) \mid B \in \text{Inst}(W)$ such that $X \cap \var{var}(s) = \emptyset$ and $A1 : \models \text{Store}(s) \Rightarrow \exists X \vartheta_1$, $A2 : \models \exists \text{Store}(s) \land \vartheta_1 \land \vartheta_2$</td>
</tr>
<tr>
<td>C2</td>
<td>$(\bar{A}; s) \to (\bar{A}; s, c^I)$ if $\models \exists \text{Store}(s, c^I), BV(c) \cap (\var{var}(\bar{A}) \cup \var{var}(s)) = \emptyset$ and $FV(c) \cap BV^O(s) = \emptyset$</td>
</tr>
<tr>
<td>C3</td>
<td>$(\bar{A}; s) \to (\bar{A}; s, c^O)$ if $\var{var}(\bar{B}) \cap BV(c) = \emptyset$</td>
</tr>
</tbody>
</table>

Table 2: The Transition System $T$. Failure Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>$(A; s) \to \langle \text{fail}; s \rangle$ if there exists no clause with head $A$ in $\text{Inst}(W)$ such that $A2$ holds.</td>
</tr>
<tr>
<td>F2</td>
<td>$(\bar{A}; s) \to \langle \text{fail}; s \rangle$, $(\bar{A}, B; s) \to \langle \text{fail}; s \rangle$</td>
</tr>
</tbody>
</table>

Unfinished computations. Such sequences are introduced in order to obtain a non-empty semantics for non-terminating programs. This is necessary to describe failure compositionally. The symbol $\alpha$ will denote an element ranging over the set $\{ss, ff, dd, \bot\}$.

We can now define the operational semantics. In the sequel, $\mathcal{P}$ will denote the powerset operation.

**Definition 3.5 (The operational semantics)** The operational semantics $\mathcal{O} : \text{Prog} \times \text{Goal} \to \mathcal{P}($Seq$)$ is given by

$$\mathcal{O}[W; A] = \{ ss : (\bar{A}, s) \to (\emptyset; s) \}$$

$$\cup \{ ff : (\bar{A}, s) \to (\text{fail}; s) \}$$

$$\cup \{ dd : (\bar{B}, s) \to (\text{susp}; s) \}$$

$$\cup \{ \bot : (\bar{A}, s) \to (\bar{B}; s) \}$$

Table 3: The Transition System $T$. Suspension Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>$(A; s) \to (\text{susp}; s)$ if for all clauses with head $A$ in $\text{Inst}(W)$ $A1$ or $A2$ does not hold and there exists such a clause for which $A1$ does not hold and $A2$ does</td>
</tr>
<tr>
<td>S2</td>
<td>$(\bar{A}; s) \to (\text{susp}; s)$, $(\bar{B}; s) \to (\text{susp}; s)$, $(\bar{A}, B; s) \to (\text{susp}; s)$</td>
</tr>
</tbody>
</table>
The transition relation \( \rightarrow \) is assumed with respect to the program \( W \).

From this operational semantics we obtain our observation criterium as follows:

**Definition 3.6 (The observables)** The observables \( \text{Obs} : \text{Prog} \times \text{Goal} \rightarrow \mathcal{P}(\text{Con} \times \{\text{ss, ff, dd}\}) \) are defined as

\[
\text{Obs}[W; A] = \text{Result}(\mathcal{O}[W; A]) / \sim
\]

where \( \text{Result}(S) = \{(\text{Estore}(s), \alpha) : s.\alpha \in S \text{ contains only output constraints and } \alpha \neq \bot\} \), and, given a set \( C \) of constraints, \( C / \sim \) denotes the closure of \( C \) under logical equivalence.

Note that we select sequences containing only output constraints thus modeling a computation that the initial goal is able to carry out on its own.

To show the compositionality of the operational semantics we define the parallel composition \( || \). This operator, first introduced in [9], allows to combine sequences of input/output constraints that are equal at each point, apart from the labels, so modeling the interaction of a process with its environment.

**Definition 3.7 (The parallel composition operator)** The partial operator \( || : \text{Seq} \times \text{Seq} \rightarrow \text{Seq} \) is defined by

\[
\begin{align*}
\bullet \ s_1.\alpha_1 || s_2.\alpha_2 &= s_2.\alpha_2 || s_1.\alpha_1 \\
\bullet \ c'.s_1.\alpha_1 || c'.s_2.\alpha_2 &= c'.(s_1.\alpha_1 || s_2.\alpha_2) \\
\bullet \ \alpha || \text{ss} &= \alpha \\
\bullet \ \alpha || \text{ff} &= \text{ff} \\
\bullet \ \text{dd} || \text{dd} &= \text{dd} \\
\bullet \ \bot || \bot &= \bot
\end{align*}
\]

Sometimes we will use the parallel composition on sequences of input/output constraints, without the termination mode (notation \( s_1 || s_2 \)). The extension of \( || \) to sets is defined in the obvious way. The following result shows the compositionality of our operational semantics with respect to goal conjunction.

**Theorem 3.8 (Compositionality of \( \mathcal{O} \))**

\[\mathcal{O}[W; A, B] = \mathcal{O}[W; A] || \mathcal{O}[W; B] \]

**Proof (Sketch)** The inclusion \( \mathcal{O}[W; A, B] \subseteq \mathcal{O}[W; A] || \mathcal{O}[W; B] \) can be proved by showing that for every computation \( (A, B, \lambda) \rightarrow^* (A', s) \) there exist computations \( (A, \lambda) \rightarrow^* (A'_1, s_1) \), \( (B, \lambda) \rightarrow^* (A'_2, s_2) \), where \( A'_1, A'_2 = A' \), and \( s_1 \parallel s_2 = s \). The proof proceeds by induction on the length of \( s \).

The other inclusion \( \mathcal{O}[W; A] || \mathcal{O}[W; B] \subseteq \mathcal{O}[W; A, B] \) is proved by showing that computations \( (A, \lambda) \rightarrow^* (A'_1, s_1), (B, \lambda) \rightarrow^* (A'_2, s_2) \), such that \( s_1 \parallel s_2 \) is defined, can be composed into a computation \( (A, B, \lambda) \rightarrow^* (A'_5, A'_1, s_0 \parallel s_1) \). The proof proceeds by induction on the length of \( s_1 \parallel s_2 \).

The compositionality result can be easily generalized to the union of neatly intersecting programs. Two programs \( W_1 \) and \( W_2 \) are neatly intersecting [9] iff every predicate \( p \in \text{Pred} \) that occurs in both programs is defined in the same way. Namely, the clauses with head \( p \) are exactly the same both in \( W_1 \) and \( W_2 \).

More in general, two pairs \( W_1; A_1 \) and \( W_2; A_2 \) are neatly intersecting iff
• $W_1$ and $W_2$ are neatly intersecting, with shared predicates $p_1, \ldots, p_k$,
• $A_1$ does not share predicates with $W_2$, apart from $p_1, \ldots, p_k$, and
• $A_2$ does not share predicates with $W_1$, apart from $p_1, \ldots, p_k$.

**Corollary 3.9** Let $W_1; A_1$ and $W_2; A_2$ be neatly intersecting. Then

$$\mathcal{O}[W_1; A_1] \cup \mathcal{O}[W_2; A_2] = \mathcal{O}[W_1; A_1] \parallel \mathcal{O}[W_2; A_2]$$

**Proof** By theorem 3.8 we have

$$\mathcal{O}[W_1; A_1] \cup \mathcal{O}[W_2; A_2] = \mathcal{O}[W_1; A_1] \parallel \mathcal{O}[W_2; A_2]$$

By the restriction upon the predicates of $A_1$ and $A_2$ we have

$$\mathcal{O}[W_1; A_1] = \mathcal{O}[W_1; A_1] \text{ and } \mathcal{O}[W_2; A_2] = \mathcal{O}[W_2; A_2]$$

\[Q.E.D.\]

In the following examples, we assume the constraint system to support the usual equality theory on the Herbrand universe.

**Example 3.10** Consider the following program

$$W_1 = \{ p(x) \leftarrow \text{ask}(x = a) \mid . \}$$

We have

$$\mathcal{O}[W_1; p(x)] = \{ \text{dd}, (x = a) \text{. ss}, (x = b) \text{. ff}, \ldots \}$$

$$\text{Obs}[W_1; p(x)] = \{ (\text{true}, \text{dd}) \}.$$ 

Consider now the program

$$W_2 = \{ q(x) \leftarrow \text{tell}(x = a) \mid . \}$$

We have

$$\mathcal{O}[W_2; q(x)] = \{ (x = a) \text{. ss}, (x = a) \text{. ss}, (x = b) \text{. ff}, \ldots \}$$

$$\text{Obs}[W_2; q(x)] = \{ (x = a, \text{ss}) \}.$$ 

We consider now the union of the two programs and the goal $\leftarrow p(x), q(x)$. Since $W_1; p(x)$ and $W_2; q(x)$ are neatly intersecting, we get

$$\mathcal{O}[W_1 \cup W_2; p(x), q(x)] = \mathcal{O}[W_1; p(x)] \parallel \mathcal{O}[W_2; q(x)]$$

$$= \{ (x = a) \text{. ss}, (x = a) \text{. ss}, (x = b) \text{. ff}, \ldots \}$$

$$\text{Obs}[W_1 \cup W_2; p(x), q(x)] = \{ (x = a, \text{ss}) \}.$$ 

The following example corresponds to example 1.3 in the introduction.

**Example 3.11** Consider the two programs

$$W_1 = \{ p_1(x, y, z) \leftarrow \text{ask}(x = a) \mid q_1(y, z).$$

$$q_1(y, z) \leftarrow \text{ask}(y = b) \mid .$$

$$q_1(y, z) \leftarrow \text{ask}(z = c) \mid . \}$$

9
\( W_2 = \{ \begin{align*}
p_2(x, y, z) &\leftarrow \text{ask}(x = a) \mid q_2(y). \\
p_2(x, y, z) &\leftarrow \text{ask}(x = a) \mid r_2(z). \\
q_2(y) &\leftarrow \text{ask}(y = b) \mid . \\
r_2(z) &\leftarrow \text{ask}(z = c) \mid .
\end{align*} \} \)

We have:

\[
O[W_1; p_1(x, y, z)] = \{ (x = a)^I, (x = a)^O, (y = b)^I, (y = b)^O, ss, \\
(x = a)^I, (x = a)^O, (z = c)^I, (z = c)^O, ss, \\
\ldots \}
\]

\[
O[W_2; p_2(x, y, z)] = \{ (x = a)^I, (x = a)^O, (y = b)^I, (y = b)^O, ss, \\
(x = a)^I, (x = a)^O, (z = c)^I, (z = c)^O, ss, \\
(x = a)^I, (x = a)^O, (y = b)^I, dd, \\
(x = a)^I, (x = a)^O, (z = c)^I, dd, \\
\ldots \}
\]

Consider now the program

\[
W = \{ p(x, y) \leftarrow \text{tell}(x = a) \mid \text{tell}(y = b). \}
\]

we have

\[
O[W; p(x, y)] = \{ (x = a)^O, (x = a)^I, (y = b)^O, ss, \\
(x = a)^O, (x = a)^I, (y = b)^I, ss, \\
\ldots \}
\]

Therefore, since \( W_1; p_1(x, y, z), W_2; p_2(x, y, z), \) and \( W; p(x, y) \) are neatly intersecting, we have

\[
O[W \cup W_1; p(x, y), p_1(x, y, z)] = O[W; p(x, y)] \parallel O[W_1; p_1(x, y, z)]
\]

\[
= \{ (x = a)^O, (x = a)^I, (y = b)^O, ss, \ldots \}
\]

\[
\text{Obs}[W \cup W_1; p(x, y), p_1(x, y, z)] = \{ (x = a \land y = b, ss) \}
\]

whilst

\[
O[W \cup W_2; p(x, y), p_2(x, y, z)] = O[W; p(x, y)] \parallel O[W_2; p_2(x, y, z)]
\]

\[
= \{ (x = a)^O, (x = a)^I, (y = b)^O, ss, \\
(x = a)^O, (x = a)^I, (y = b)^I, dd, \\
\ldots \}
\]

\[
\text{Obs}[W \cup W_2; p(x, y), p_2(x, y, z)] = \{ (x = a \land y = b, ss), (x = a \land y = b, dd) \}
\]

4 The fully abstract semantics \( \mathcal{D} \)

The operational semantics \( \mathcal{O} \) defined in the previous section is not fully abstract. The reason is that, due to the monotonic nature of communication, the order and the granularity in which constraints are produced cannot be sensed by any context. Indeed, after any (logically) equivalent sequence of constraints produced by the process, the reaction of the context will be the same.

The fully abstract semantics is obtained by applying to the operational semantics some closure conditions that eliminate at the set level the distinctions due to different order and granularity in the sequences.

For a concise description of the closure conditions, we modularize the sequences of \( \mathcal{O} \). The notion of modular sequence has been introduced in [11]. Intuitively, a sequence is modular if \( E_{\text{store}}(s) \) is equivalent to the conjunction of the constraints in \( s \):
Definition 4.1 A sequence of constraints is called modular if and only if for arbitrary two distinct constraints $c$ and $c'$ occurring in $s$ we have $FV(c) \cap BV(c') = \emptyset$.

Example 4.2 The sequence $(\exists y \ x = f(y))', (x = a)''$ is not modular, whilst $(\exists y \ x = f(y))', (x = f(a))''$ is.

In order to transform non modular sequences into modular ones without changing the meaning and the structure, we define a notion of equivalence $\equiv$.

Definition 4.3 We define $s_1 \sim s_2$ iff

- $s_1 = s.c', s'$ and $s_2 = s.c', s''$
- $\models Estore(s.c') \iff Estore(s.c'')$
- $\models Estore(s_1) \iff Estore(s_2)$

Let $\equiv$ be the reflexive, transitive closure of $\sim$.

It is easy to see that, for every sequence $s$, there exists a modular sequence $s'$ such that $s \equiv s'$.

Example 4.4 Consider the following sequences

- $s_1 = (\exists y \ x = f(y))', (x = f(y))', (x = f(y))', (y = g(a))''$
- $s_2 = (\exists y \ x = f(y))', (\exists y, z)(x = f(y) \land y = g(z))', (x = f(a))''$
- $s_3 = (\exists y \ x = f(y))', (\exists y, z)(x = f(y) \land y = g(z))', (x = f(g(a)))''$
- $s_4 = (\exists y \ x = f(y))', (\exists x = f(g(z)))', (x = f(g(a)))''$

We have $s_1 \sim s_2 \sim s_3 \sim s_4$ (and therefore $s_1 \equiv s_4$). Note that $s_1$ and $s_2$ are not modular whilst $s_3$ and $s_4$ are.

In order to structure the presentation of the fully abstract semantics $\mathcal{D}$, we first introduce an intermediate semantics $\mathcal{O}'$ obtained by modularizing the sequences of $\mathcal{O}$.

Definition 4.5 (The operational semantics $\mathcal{O}'$)

$\mathcal{O}'[W; \vec{A}] = \{ s.\alpha : s$ is modular and there exists $s'.\alpha \in \mathcal{O}[W; \vec{A}]$ such that $s \equiv s'$ $\}$

Obviously we have

Proposition 4.6 (Correctness of $\mathcal{O}'$)

$\mathcal{O}[W; \vec{A}] = \mathcal{O}'[W; \vec{A}]$ from $\mathcal{O}'[W; \vec{A}]$ to $\mathcal{O}[W; \vec{A}]$

The semantics $\mathcal{O}'$ is more abstract than $\mathcal{O}$, but still compositional:

Proposition 4.7 (Compositionality of $\mathcal{O}'$)

$\mathcal{O}'[W, \vec{A}, \vec{B}] = \mathcal{O}'[W, \vec{A}] \parallel \mathcal{O}'[W, \vec{B}]$

Proof See appendix A. $\square$

The following corollary extends the previous result to neatly intersecting programs. It follows from proposition 4.7 in the same way as corollary 3.9 follows from theorem 3.8.

Corollary 4.8 Let $W_1; \vec{A}_1$ and $W_2; \vec{A}_2$ be neatly intersecting. Then

$\mathcal{O}'[W_1 \cup W_2; \vec{A}_1, \vec{A}_2] = \mathcal{O}'[W_1; \vec{A}_1] \parallel \mathcal{O}'[W_2; \vec{A}_2]$
We define now the closure conditions that will induce some additional identifications necessary for full abstraction.

**Definition 4.9** Given a set \( S \subseteq \text{Seq} \), we define \( \text{Closure}(S) \) to be the minimal set containing \( S \) and satisfying the conditions \( P_1 \) and \( P_2 \) of table 4. In this table we assume all sequences to be modular, and \( R \) to be an arbitrary set of modular sequences.

Sometimes we will use the notation \( \text{Closure}(S) \) also for a set \( S \) of sequences of input/output constraints (without the termination mode), with the obvious meaning.

The condition \( P_1 \) represents the abstraction with respect to order and granularity. Two sequences that only differ for some logically equivalent subsequences of constraints produced by the same agent (either the process or the environment) must be identified. The condition \( P_2 \) completes this identification at the set level. The arbitrary input constraints (given automatically by the transition system in the original sequences of \( O' \)) must also be added to the new sequences generated by \( P_1 \), in order to eliminate the remaining distinctions.

**Remark 4.10** The closure conditions preserve the meaning of a sequence. Namely, if \( s.\alpha \in \text{Closure}([s'.\alpha']) \) (abbrev. \( \text{Closure}(s'.\alpha') \)), then \( \exists \text{Estore}(s) \Leftrightarrow \text{Estore}(s') \) holds.

**Remark 4.11** The closure operator is idempotent, namely

\[
\text{Closure}(\text{Closure}(S)) = \text{Closure}(S)
\]

We define the fully abstract semantics \( \mathcal{D} \) in a denotational (i.e. compositional) style, and the basic cases (empty and unit goals) we obtain by applying the closure operator to \( O' \).

**Definition 4.12 (The fully abstract semantics \( \mathcal{D} \))** We define \( \mathcal{D} : \text{Prog} \times \text{Goal} \to \mathcal{P}(\text{Seq}) \) as follows

\[
\begin{align*}
\mathcal{D}[W;\emptyset] & = \text{Closure}(O'[W;\emptyset]) \\
\mathcal{D}[W;A] & = \text{Closure}(O'[W;A]) \\
\mathcal{D}[W;A,B] & = \text{Closure}(\mathcal{D}[W;A] \parallel \mathcal{D}[W;B])
\end{align*}
\]

The semantics \( \mathcal{D} \) is strictly more abstract than \( \mathcal{O} \) and \( O' \).

**Example 4.13** Consider the two programs

\[
W_1 = \{ \text{p}_1(x) \leftarrow \exists y \text{tell}(x = f(y)) \mid q(y). \\
\text{q}(y) \leftarrow \text{tell}(y = a) \mid . \}
\]

\[
W_2 = \{ \text{p}_2(x) \leftarrow \text{tell}(x = f(a)) \mid . \\
\text{p}_2(x) \leftarrow \exists y \text{tell}(x = f(y)) \mid q(y). \\
\text{q}(y) \leftarrow \text{tell}(y = a) \mid . \}
\]
We have that \( W_1; p_1(x) \) and \( W_2; p_2(x) \) behave in the same way under every context (i.e. they are observationally equivalent), namely, for every \( W; A \), neatly intersecting with \( W_1; p_1(x) \) and \( W_2; p_2(x) \):

\[
\text{Obs}[W \cup W_1; A, p_1(x)] = \text{Obs}[W \cup W_2; A, p_2(x)]
\]

This is because \( W_1 \subseteq W_2 \) and the behaviour of the first clause of \( W_2 \) can be simulated by the two clauses of \( W_1 \). The operational semantics of these programs is however different (i.e. \( O \) is not fully abstract). In fact

\[
(x = f(a))^O ss \in O[W_2; p_2(x)]
\]

whilst

\[
(x = f(a))^O ss \notin O[W_1; p_1(x)]
\]

The same applies to \( O' \). On the other side, this difference in the operational semantics disappears in the denotational one. In fact, we have

\[
(\exists y \ x = f(y))^O \cdot (y = a)^O ss \in O[W_1; p_1(x)], \quad \text{and}
\]

\[
(\exists y \ x = f(y))^O \cdot (x = f(a))^O ss \in O'[W_1; p_1(x)]
\]

therefore, by an application of \( P_1 \), we obtain

\[
(x = f(a))^O ss \in D[W_1; p_1(x)].
\]

This example shows the use of \( P_1 \) to abstract from granularity of constraint production. In this case, \( P_1 \) has been used to group a sequence of output constraints (to enlarge the granularity). Examples in which \( P_1 \) is needed to split an output constraint (to reduce the granularity) are a bit more complicated, but not difficult to imagine.

Concerning \( P_2 \), its use can be understood to complete the abstraction made by \( P_1 \). Indeed, when we split a constraint into a sequence, we have also to allow additional interleaving points within this sequence. This is modeled by inserting arbitrary (consistent) input constraints \( (P_2) \).

By definition, \( D \) is compositional. As usual, compositionality can be extended to neatly intersecting programs:

**Proposition 4.14** Let \( W_1; A_1 \) and \( W_2; A_2 \) be neatly intersecting. Then

\[
D[W_1 \cup W_2; A_1, A_2] = \text{Closure}(D[W_1; A_1] \parallel D[W_2; A_2])
\]

The next sections will show that \( D \) is correct with respect to the observables and that it is fully abstract.

## 5 The correctness of \( D \)

In this section we prove the correctness of \( D \) with respect to the observables. The structure of the proof is the following: first we prove that \( D \) is the closure of \( O' \) for arbitrary goals (not only for the empty and unit ones), then we use the correctness of \( O' \).

**Lemma 5.1** For arbitrary sets \( S_1 \) and \( S_2 \) of modular sequences which are closed under \( P_2 \) and the following version of \( P_1 \)

\[
P_1-1 \quad s_1, c_1^1 \ldots c_n^1, s_2, \alpha \in R \Rightarrow s_1, c_1^1 \ldots c_m^l, s_2, \alpha \in R
\]

if \( \models Estore(s_1, c_1^1 \ldots c_n^1) \Leftrightarrow Estore(s_1, c_1^l \ldots c_m^l) \)

we have

\[
\text{Closure(Closure}(S_1) \parallel \text{Closure}(S_2)) = \text{Closure}(S_1 \parallel S_2)
\]
Proof See appendix B.

Proposition 5.2 \[ D[W; \tilde{A}] = \text{Closure}(O'[W; \tilde{A}]) \]

Proof Straightforward induction on the length of the goal, using lemma 5.1 (it is easy to verify that \( O' \) is closed under \( P1-I \) and \( P2 \)) and the compositionality of \( O' \).

We can now prove the correctness of \( D \):

Theorem 5.3 (Correctness of \( D \)) \[ \text{Result}(D[W; \tilde{A}]) = \text{Obs}[W; \tilde{A}] \]

Proof

\[
\begin{align*}
\text{Result}(D[W; \tilde{A}]) &= (\text{by proposition 5.2}) \\
\text{Result}(\text{Closure}(O'[W; \tilde{A}])) &= (\text{by remark 4.10}) \\
\text{Result}(O'[W; \tilde{A}]) &= (\text{by proposition 4.6}) \\
\text{Obs}[W; \tilde{A}] &= (\text{by proposition 5.2}) \\
&= (\text{by remark 4.10}) \\
&= (\text{by proposition 4.6})
\end{align*}
\]

6 The full abstraction of \( D \)

In this section we prove the full abstraction of \( D \) with respect to our observation criterium. The basic lines of the proof are the following. Given two goals \( \rightarrow A_1, \rightarrow A_2 \) with a different semantics \( D \), we build a context that is able to "detect" this difference at the observational level. The definition of this context is uniform, in the following sense: given a modular sequence \( s.a \) we define a context \( C(s.a) \) as a pair \( \text{program;goal} \) which "recognizes" \( s.a \). Next we prove that every other sequence \( s'.a \) recognized by \( C(s.a) \) that gives the same result must generate \( s.a \) by application of the closure operator. Then we reason by contradiction: given a sequence \( s.a \) in the semantic difference \( \cdot a \in D[W; \tilde{A}_1] \setminus D[W; \tilde{A}_2] \), if the context \( C(s.a) \) doesn't induce a difference in the observables, then there exists an other sequence \( s'.a \) recognized by the context that produces the same result. But, since \( D \) is closed, the presence of \( s'.a \) implies the presence of \( s.a \), and this contradicts the assumption.

Definition 6.1 Let \( s.a \) be a modular sequence, and \( \bar{z} \) be the free variables of \( s \). We define the context \( C(s.a) \) by induction on the length \( n \) of \( s \). We assume given a set of new predicate symbols \( \{p_0, \ldots, p_n\} \) disjoint from \( \text{Pred} \).

- \( C(ss) = C(ff) = C(dd) = \{p_0(\bar{z}) \leftarrow \cdot \}; p_0(\bar{z}) \), and \( C(\perp) = \{}; p_0(\bar{z}) \).
- \( C((\exists Y \theta)^f \cdot s.a) = \{p_n(\bar{z}) \leftarrow \exists Y \text{tell}(\theta)\}|p_{n-1}(\bar{z})\}; W; p_n(\bar{z}) \), where \( W; p_{n-1}(\bar{z}) = C(s.a) \).
- \( C((\exists Y \theta)^d \cdot s.a) = \{p_n(\bar{z}) \leftarrow \exists Y \text{ask}(\theta)\}|p_{n-1}(\bar{z})\}; W; p_n(\bar{z}) \), where \( W; p_{n-1}(\bar{z}) = C(s.a) \).

Note that the goal \( \rightarrow p_0(\bar{z}) \) gives rise to failure in \( C(\perp) \), since the predicate \( p_0 \) is undefined. In the other cases, it succeeds.

The following proposition states that a context \( C(s.a) \) recognizes the sequence \( s.a \). Namely, \( C(s.a) \) generates \( \tilde{s}.\tilde{a} \), where \( \tilde{s} \) denotes the "mirror" of \( s \), i.e. \( \tilde{c^f}.s = c^d.\tilde{s} \) and \( \tilde{c^d}.s = c^f.\tilde{s} \), and

\[
\tilde{a} = \begin{cases} 
ss & \text{if } \alpha \in \{ss, ff, dd\} \\
ff & \text{otherwise.}
\end{cases}
\]

Proposition 6.2 For any modular sequence \( s.a \) we have \( \tilde{s}.\tilde{a} \in D[C(s.a)] \).
Proof Let \( s' \) be obtained by \( s \) by adding an output constraint \( c^O \) after any input constraint \( c' \). It is not difficult to see that \( s'.a \in \mathcal{O}[C(s,a)] \). Furthermore, since \( s \) is modular, also \( s \) and \( s' \) are modular. Hence

\[
\tilde{s}.\tilde{a} \in \mathcal{O}'[C(s,a)]
\]

We conclude

\[
\begin{align*}
\tilde{s}.\tilde{a} & \in \mathcal{O}[C(s,a)] \quad \text{(by P1)} \\
\text{Closure}(\tilde{s}.\tilde{a}) & \subseteq \mathcal{O}[C(s,a)] \quad \text{(by proposition 5.2)} \\
\mathcal{D}[C(s,a)] & = \mathcal{O}'[C(s,a)] \quad \text{(by remarks 4.10 and 4.11)}
\end{align*}
\]

The next two lemmas together imply that the context \( C(s,a) \) recognizes only \( s.a \), in the sense that if \( C(s,a) \) interacts with a sequence \( s' \) (i.e., for some \( a' \) we have \( \tilde{s'.a' \in \mathcal{D}[C(s,a)]} \)), which gives the same result as \( s \), i.e., \( \models \text{Estore}(s) \iff \text{Estore}(s') \), then \( s \) can be obtained from \( s' \) by applying the closure operator.

The next lemma actually shows that \( \tilde{s} \) is in the closure of \( \tilde{s} \). The final step is then made by the mirroring lemma (see page 17).

Lemma 6.3 Given a modular sequence \( s.a \), for every \( s'.a' \in \mathcal{D}[C(s,a)] \) such that \( \models \text{Estore}(s') \iff \text{Estore}(\tilde{s}) \) we have \( s' \in \text{Closure}(\tilde{s}) \).

Proof By proposition 5.2, \( \mathcal{D}[C(s,a)] = \text{Closure}(\mathcal{O}'[C(s,a)]) \) holds. Therefore, by remarks 4.10 and 4.11, it is sufficient to prove that, for \( s'.a' \in \mathcal{O}'[C(s,a)] \) with \( \models \text{Estore}(s') \iff \text{Estore}(\tilde{s}) \), we have \( s' \in \text{Closure}(\tilde{s}) \).

Let \( s'.a' \in \mathcal{O}'[C(s,a)] \) such that \( \models \text{Estore}(s') \iff \text{Estore}(\tilde{s}) \). Let \( n \) be the length of \( s \). It is not so difficult to see that there exists a computation

\[
\langle p_0(z), \lambda \rangle \rightarrow^* \cdots \rightarrow^* \langle p_{n-1}(z), s'_i \rangle \rightarrow^* \cdots \rightarrow^* \langle p_{n}(z), s'_n \rangle
\]

such that \( s'_i \equiv s' \).

Let \( s'_0 \) denote the prefix of \( s' \) such that \( s'_0 \equiv s'_i \), and let \( \tilde{s}^{(i)} \) denote the suffix of \( \tilde{s} \) starting from its \((i+1)\)th element. We prove that \( s'_i, \tilde{s}^{(i)} \in \text{Closure}(\tilde{s}) \) for \( 0 \leq i \leq n \). We proceed by induction on \( i \).

\( i = 0 \) Obvious, since \( s'_0 = \lambda \) and \( \tilde{s}^{(0)} = \tilde{s} \).

\( i + 1 \) By the induction hypothesis we have

\[
s'_i, \tilde{s}^{(i)} \in \text{Closure}(\tilde{s}).
\]

There are two cases

Case 1: \( \tilde{s}^{(i)} = c'.c^{(i+1)} \) In this case \( p_{n-i} \) is defined in \( C(s.a) \) as

\[
p_{n-i}(z) \leftarrow \exists y \cdot \text{ask}(\tilde{y}) \cdot p_{n-i-1}(z),
\]

where \( c = \exists y \cdot \tilde{y} \). So we have

\[
\langle p_{n-i}(z), s'_i \rangle \rightarrow^* \langle p_{n-i-1}(z), s'_{i+1} \rangle
\]

where

\[
s'_{i+1} = c'_i c'_1 \cdots c'_e . c^{O}
\]
with
\[ \models \text{Store}(e, c_1, \ldots, c_k) \Rightarrow c. \] (3)

Since \( s' \equiv s_n \), we have
\[ s_{i+1}' = s_{i}' \cdot c_1 \ldots c_k \cdot c^O. \]

We note that
\[ \models \text{Store}(s') \iff \text{Store}(\hat{s}) \] (by hypothesis)
\[ \text{Store}(s'_i, s^{(i)}) \] (by 2 and remark 4.10)

from which we derive
\[ \models \text{Store}(s'_i, s^{(i)}) \Rightarrow \text{Store}(s'_i, s^{(i)}) \Rightarrow \text{Store}(s'_i, \hat{s}^{(i)}) \] (4)

Therefore, since \( s'_i \cdot c_1 \ldots c_k \cdot \hat{s}^{(i+1)} \in \text{Closure}(s'_i, \hat{s}^{(i)}) \) (by 4 and P2), and \( \text{Closure}(s'_i, \hat{s}^{(i)}) \subseteq \text{Closure}(\hat{s}) \) (by 2 and remark 4.11), we obtain
\[ s'_i \cdot c_1 \ldots c_k \cdot \hat{s}^{(i+1)} \in \text{Closure}(\hat{s}) \] (5)

Now we have to “transform” \( c^I \) into \( c^O \). We do so by two applications of P1. The first deletes \( c^I \), the second adds \( c^O \). For the first application we need the following:

\[ \models \text{Store}(s'_i, c_1 \ldots c_k) \iff (\text{since } s' \equiv s_n) \]
\[ \text{Store}(s'_i, c_1 \ldots c_k) \iff (3) \]
\[ \text{Store}(s'_i, c_1 \ldots c_k) \iff (\text{since } FV(c) \subseteq FV(s), \text{as } s \text{ is modular}) \]
\[ \text{Store}(s'_i, c_1 \ldots c_k) \land c \iff (\text{since } s' \equiv s_n) \]
\[ \text{Store}(s'_i, c_1 \ldots c_k \land c) \iff (\text{since } s' \equiv s_n) \]
\[ \text{Store}(s'_i, c_1 \ldots c_k). \] (6)

The last equation holds under the assumption \( BV(s') \cap FV(s) = \emptyset \). We can assume this without loss of generality, since \( \models \text{Store}(s') \iff \text{Store}(\hat{s}) \iff \text{Store}(s) \).

Now we can apply P1, thus obtaining
\[ s'_i \cdot c_1 \ldots c_k \cdot \hat{s}^{(i+1)} \in \text{Closure}(s'_i, c_1 \ldots c_k \cdot \hat{s}^{(i+1)}). \] (6)

For the second application we need

\[ \models \text{Store}(s'_i, c_1 \ldots c_k) \iff (\text{since } s' \equiv s_n) \]
\[ \text{Store}(s'_i, c_1 \ldots c_k) \iff (3) \]
\[ \text{Store}(s'_i, c_1 \ldots c_k) \iff (\text{since } s' \equiv s_n) \]
\[ \text{Store}(s'_i, c_1 \ldots c_k). \] (6)

Then, we can apply P1 again, thus obtaining
\[ s'_i \cdot c_1 \ldots c_k \cdot \hat{s}^{(i+1)} \in \text{Closure}(s'_i, c_1 \ldots c_k \cdot \hat{s}^{(i+1)}). \] (7)

We conclude
\[ s'_i \cdot \hat{s}^{(i+1)} \]
\[ s'_i \cdot c_1 \ldots c_k \cdot \hat{s}^{(i+1)} \in (by 7) \]
\[ \text{Closure}(s'_i, c_1 \ldots c_k \cdot \hat{s}^{(i+1)}) \subseteq (by 6) \]
\[ \text{Closure}(s'_i, c_1 \ldots c_k \cdot \hat{s}^{(i+1)}) \subseteq (by 5) \]
\[ \text{Closure}(\hat{s}) \]
Case 2: $\tilde{s}^{(i)} = c^O \tilde{s}^{(i+1)}$ In this case $p_{n-i}$ is defined in $C(s, \alpha)$ as

$$p_{n-i}(\tilde{x}) \leftarrow \exists y \text{tell}(\tilde{y}) p_{n-i-1}(\tilde{x}),$$

where $c = \exists y \tilde{y}$. So we have

$$\langle p_{n-i}(\tilde{x}), s'_t \rangle \longrightarrow^* \langle p_{n-i-1}(\tilde{x}), s'_{t+1} \rangle$$

where

$$s'_{t+1} = s'_t c'_1 \ldots c'_k c^O$$

Since $s' \equiv s'_n$, we have

$$s'_{(i+1)} = s'_i c'_1 \ldots c'_k c^O.$$

Similarly to the previous case, we have

$$\models \text{Estore}(s'_{(i)} c'_1 \ldots c'_k c^O \tilde{s}^{(i+1)}) \iff \text{Estore}(s'_{(i)} c'_1 \ldots c'_k \tilde{s}^{(i)}) \iff \text{Estore}(s'_{(i)} \tilde{s}^{(i)}) \tag{8}$$

therefore, since $s'_{(i)} c'_1 \ldots c'_k c^O \tilde{s}^{(i+1)} \in \text{Closure}(s'_{(i)} \tilde{s}^{(i)})$ (by 8 and P2), and $\text{Closure}(s'_{(i)} \tilde{s}^{(i)}) \subseteq \text{Closure}(\tilde{s})$ (by 2 and remark 4.11), we obtain

$$s'_{(i)} c'_1 \ldots c'_k c^O \tilde{s}^{(i+1)} \in \text{Closure}(\tilde{s}) \tag{9}$$

We have now to “transform” $c^O$ into $c'$. We need the following

$$\models \text{Estore}(s'_{(i)} c'_1 \ldots c'_k c^O) \iff (\text{since } \text{FV}(c) \subseteq \text{FV}(\tilde{s}))$$

$$\text{Estore}(s'_{(i)} c'_1 \ldots c'_k c^O) \land c \iff (\text{since } s' \equiv s'_n)$$

$$\text{Estore}(s'_{(i)} c'_1 \ldots c'_k c^O) \therefore (\text{since } \text{FV}(c) \subseteq \text{FV}(s))$$

$$\text{Estore}(s'_{(i)} c'_1 \ldots c'_k c^O) \iff (\text{since } s' \equiv s'_n)$$

Thus an application of P1 yields

$$s'_{(i)} c'_1 \ldots c'_k c^O \tilde{s}^{(i+1)} \in \text{Closure}(s'_{(i)} c'_1 \ldots c'_k c^O \tilde{s}^{(i+1)}). \tag{10}$$

Therefore we can conclude

$$s'_{(i+1)} \tilde{s}^{(i+1)} = s'_{(i)} c'_1 \ldots c'_k c^O \tilde{s}^{(i+1)} \in \text{Closure}(s'_{(i)} c'_1 \ldots c'_k c^O \tilde{s}^{(i+1)}) \in \text{Closure}(\tilde{s}) \tag{10}$$

Lemma 6.4 (Mirroring lemma) If $s' \in \text{Closure}(s)$, then $\tilde{s} \in \text{Closure}(\tilde{s}')$. (As usual, $\tilde{s}$ and $\tilde{s}'$ denote the mirror of $s$, $s'$, respectively.)

**Proof** It is sufficient to show that for any set of sequences $S$, if $\text{Closure}(S) = S$, then $S$ satisfies the following property:

**P3** $s_1 c^O s_2, \alpha \in S \Rightarrow s_1 s_2, \alpha \in S$

if $\models \text{Estore}(s_1, s_2) \iff \text{Estore}(s_1 c^O s_2)$.

17
We then can proceed by induction on the number of applications of the closure conditions \( P_1 \) and \( P_2 \), making use of the fact that \( P_1 \) mirrors itself, in the following sense: if \( s' \) is derived from \( s \) by one application of \( P_1 \) then \( \tilde{s} \) can be derived from \( s' \) using \( P_1 \) again. In same sense an application of \( P_2 \) can be mirrored by \( P_3 \).

We prove \( P_3 \) by induction on the length of \( s \).

\( s_2 = \lambda \) In this case we just apply \( P_1 \).

\( s_2 = c^{\ell} \cdot s_2' \) We consider the cases \( \ell = 1 \) and \( \ell = O \) separately.

\( \ell = O \) By \( P_1 \) we have

\[ s_1 \cdot c \cdot s_2', a \in S \Rightarrow s_1 \cdot c \cdot s_2', a \in S \]

The induction hypothesis then gives us

\[ s_1 \cdot c \cdot s_2', a \in S \]

\( \ell = 1 \) By \( P_2 \) we have

\[ s_1 \cdot c \cdot s_2', a \in S \Rightarrow s_1 \cdot c \cdot s_2', a \in S \]

An application of \( P_1 \) then gives us

\[ s_1 \cdot c \cdot s_2', a \in S \]

By induction hypothesis we obtain

\[ s_1 \cdot c \cdot s_2', a \in S. \]

Now we are ready to prove the main theorem

**Theorem 6.5 (Full abstraction of \( D \)**) For arbitrary \( W_1; \tilde{A}_1, W_2; \tilde{A}_2 \) such that \( D[W_1; \tilde{A}_1] \neq D[W_2; \tilde{A}_2] \), there exists \( W; \tilde{A} \), neatly intersecting with \( W_1; \tilde{A}_1 \) and \( W_2; \tilde{A}_2 \), such that \( \text{Obs}[W \cup W_1; \tilde{A}, \tilde{A}_1] \neq \text{Obs}[W \cup W_2; \tilde{A}, \tilde{A}_2] \).

**Proof** Assume \( s.a \in D[W_1; \tilde{A}_1] \setminus D[W_2; \tilde{A}_2] \). Let \( W; \tilde{A} = C(s.a) \), and let

\[ \tilde{a} = \begin{cases} \text{ss} & \text{if } a \in \{ss, ff, dd\} \\ \text{ff} & \text{otherwise.} \end{cases} \]

By proposition 6.2 we have \( \tilde{s}.\tilde{a} \in D[W; \tilde{A}] \), therefore

\[ \text{Result}(s.a \parallel \tilde{s}.\tilde{a}) \in \text{Result}(D[W \cup W_1; \tilde{A}, \tilde{A}_1]) \setminus \infty = \text{Obs}[W \cup W_1; \tilde{A}, \tilde{A}_1]. \]

Assume now that

\[ \text{Result}(s.a \parallel \tilde{s}.\tilde{a}) \in \text{Obs}[W \cup W_2; \tilde{A}, \tilde{A}_2] = \text{Result}(D[W \cup W_2; \tilde{A}, \tilde{A}_2]) \setminus \infty \]

By the compositionality of \( D \) and remark 4.10 it follows that there exist \( s'.a' \in D[W_2; \tilde{A}_2] \) and \( \tilde{s}'.\tilde{a}' \in D[W; \tilde{A}] \) such that

\[ \text{Result}(s'.a' \parallel \tilde{s}'.\tilde{a}') = \text{Result}(s.a \parallel \tilde{s}.\tilde{a}). \] (11)

Thus we have

\[ \vdash \text{Estore}(\tilde{s}') \iff \text{Estore}(s' \parallel \tilde{s}') \iff \text{Estore}(s \parallel \tilde{s}) \iff \text{Estore}(\tilde{s}), \]

so by lemma 6.3 we have \( \tilde{s}' \in \text{Closure}(\tilde{s}) \). An application of lemma 6.4 then yields \( s \in \text{Closure}(s') \). Therefore \( s.a \in D[W_2; \tilde{A}_2] \) holds. By definition of \( O \) and \( O' \), and proposition 5.2, it follows \( s \perp \in D[W_2; \tilde{A}_2] \). Furthermore we observe that, by definition of \( \tilde{a} \), of the context \( C(s.a) \), and by 11, we have \( a' = a \) if \( a \neq \perp \). So we conclude \( s.a \in D[W_2; \tilde{A}_2] \), and this contradicts our initial assumption. \( \square \)
7 Conclusions and future work

We have studied in this paper the asynchronous nature of the communication in concurrent logic languages. We have shown that the fully abstract semantics for these languages requires an approach quite different from the standard ones for languages like CCS. One of the main differences consists in the description of the deadlock behaviour. In CCS the deadlock of a process depends upon the current state of the system as described by the failure sets, whereas in concurrent logic languages the deadlock essentially depends upon the result of the past behaviour of the system.

A future research topic is the investigation of the non-monotonic case, namely when non-monotonic test predicates (like the *var* of Prolog) occur in the guards.

Another promising subject is the extension of our approach to more general concurrent logic paradigms, like concurrent constraint programming [18], which include an explicit choice operator.

An even more ambitious project is to show that the construction of the fully abstract model we have presented can be extended to asynchronous languages in general. This conjecture asks for the definition of a general paradigm which subsumes all possible asynchronous communication mechanisms. For such a paradigm we think it is possible to define a compositional semantics based upon linear sequences with input assumptions, thus providing an uniform deadlock analysis for asynchronous communication. The fully abstract model for a particular asynchronous language then will consist of the application of some appropriate closure conditions on the general compositional model. These closure conditions will model the specific features of the asynchronous communication of that particular language. Actually, part of this project is already under development in [6].

Acknowledgements Part of the motivations for this work originated from the joint work with Joost Kok and Jan Rutten on the semantics of PARLOG and GHC. We thank the members of the C.W.I. concurrency group, J.W. de Bakker, F. Breugel, A de Bruin, E. Horita, P. Knijnenburg, J. Kok, J. Rutten, E. de Vink and J. Warmerdam for their comments on preliminary versions of this paper. We also thank Seif Haridi and Vijay Saraswat for helpful and stimulating discussions.

References


Appendix

A  The compositionality of \( \mathcal{O} \)

In this appendix we prove:

**Proposition 4.7**

\[
\mathcal{O}'[W; A, B] = \mathcal{O}'[W; A] \parallel \mathcal{O}'[W; B]
\]

**Proof** We prove \( \mathcal{O}'[W; A] \parallel \mathcal{O}'[W; B] \subseteq \mathcal{O}'[W; A, B] \), the other inclusion follows immediately from the compositionality of \( \mathcal{O} \). Let \( r_1, \alpha_1 \parallel r_2, \alpha_2 \in \mathcal{O}'[W; A] \parallel \mathcal{O}'[W; B] \), say, \( r_1 = e_1^{t_1} \cdots e_n^{t_n} \), and \( r_2 = e_1^{t'_1} \cdots e_n^{t'_n} \). So there exist

- \( s_1 = e_1^{t_1} \cdots e_n^{t_n} \), \( s_1, \alpha_1 \in \mathcal{O}[W; A] \)
- \( s_2 = e_1^{t'_1} \cdots e_n^{t'_n} \), \( s_2, \alpha_2 \in \mathcal{O}[W; B] \),

such that \( r_1 \equiv s_1 \), and \( r_2 \equiv s_2 \).

We may assume without loss of generality that the sets \( BV(r_1) \), \( BV(s_1) \), and \( BV(s_2) \) have no variables in common. Define \( s'_1 = f_1^{t_1} \cdots f_n^{t_n} \) and \( s'_2 = g_1^{t'_1} \cdots g_n^{t'_n} \), where, for \( 1 \leq i \leq n \),

\[
f_i = g_i = \begin{cases} e_i & \text{if } \ell_i = O \\ d_i & \text{if } \ell_i' = O \\ c_i & \text{otherwise} \end{cases}
\]

Note that \( s'_1 \parallel s'_2 \) is defined.

It remains to be shown that \( s'_1, \alpha_1 \in \mathcal{O}[W; A] \), \( s'_2, \alpha_2 \in \mathcal{O}[W; B] \), that is, \( s'_1, \alpha_1 \parallel s'_2, \alpha_2 \in \mathcal{O}[W; A, B] \), and \( s'_1 \parallel s'_2 \equiv s_1 \parallel s_2 \). For this purpose we introduce the following

**Lemma A.1** For \( 1 \leq i \leq n \) we have

\[\models \text{Estore}(f_1^{t_1} \cdots f_i^{t_i}) \leftrightarrow \text{Estore}(c_1^{t_1} \cdots c_i^{t_i})\]

and

\[\models \text{Estore}(g_1^{t'_1} \cdots g_i^{t'_i}) \leftrightarrow \text{Estore}(d_1^{t'_1} \cdots d_i^{t'_i}).\]

**Proof of Lemma A.1** We prove the first equivalence by induction on \( i \) (the second is treated similarly): suppose the proposition holds for \( i \). We consider the cases \( \ell_{i+1} = I \), \( \ell_{i+1} = O \) separately.

Let \( \ell_{i+1} = O \), so \( f_{i+1} = c_{i+1} \). Furthermore, let \( \text{out}(s) [\text{in}(s)] \) be the subsequence of \( s \) consisting of all the output [input] constraints. We have

\[\models \text{Estore}(f_1^{t_1} \cdots f_i^{t_i+1}) \leftrightarrow (\text{since } BV(s_1) \cap BV(r_1) = BV(s_1) \cap BV(s_2) = \emptyset) \]

\[\text{Estore(out}(f_1^{t_1} \cdots f_i^{t_i+1})) \land \text{Estore}(f_1^{t_1} \cdots f_i^{t_i}) \leftrightarrow \text{Estore(out}(c_1^{t_1} \cdots c_i^{t_i+1})) \land \text{Estore}(c_1^{t_1} \cdots c_i^{t_i}) \leftrightarrow (\text{since } BV^f(s_1) \cap BV^O(s_1) = \emptyset) \]

Now let \( \ell_{i+1} = I \), so \( f_{i+1} = c_{i+1} \). We have

\[\models \text{Estore}(f_1^{t_1} \cdots f_i^{t_i+1}) \leftrightarrow (\text{since } BV(r_1) \cap BV(s_1) = BV(r_1) \cap BV(s_2) = \emptyset) \]

\[\text{Estore(in}(e_1^{t_1} \cdots e_i^{t_i})) \land \text{Estore}(f_1^{t_1} \cdots f_i^{t_i}) \leftrightarrow (\text{since } r_1 \equiv s_1) \]

\[\text{Estore(in}(e_1^{t_1} \cdots e_i^{t_i})) \land \text{Estore}(c_1^{t_1} \cdots c_i^{t_i}) \leftrightarrow (\text{since } r_1 \equiv s_1) \]

\[\text{Estore}(e_1^{t_1} \cdots e_i^{t_i+1}) \leftrightarrow (\text{since } r_1 \equiv s_1) \]

\[\text{Estore}(e_1^{t_1} \cdots e_i^{t_i+1}) \leftrightarrow (\text{since } r_1 \equiv s_1) \]

\[\text{Estore}(e_1^{t_1} \cdots e_i^{t_i+1}) \leftrightarrow (\text{since } r_1 \equiv s_1) \]
Finally, let \( \ell_{i+1} = 1 \) and \( \ell'_{i+1} = 0 \). We have

\[
\models Estore(f_1^{i_1} \ldots f_{i+1}^{i_1}) \iff (\text{since } BV(r_1) \cap BV(s_2) = BV(s_1) \cap BV(s_2) = \emptyset)
\]

\[
Estore(out(d_1^{i_1} \ldots d_{i+1}^{i_1})) \land Estore(f_1^{i_1} \ldots f_{i+1}^{i_1}) \iff (\text{since } r_1 \equiv s_1 \text{ and } r_2 \equiv s_2)
\]

\[
Estore(out(d_1^{i_1} \ldots d_{i+1}^{i_1})) \land Estore(c_1^{i_1} \ldots c_{i+1}^{i_1}) \iff (\text{since } BV^I(s_2) \cap BV^O(s_2) = \emptyset)
\]

\[
Estore(d_1^{i_1} \ldots d_{i+1}^{i_1}) \land Estore(d_1^{i_1} \ldots d_{i+1}^{i_1}) \iff (\text{since } r_2 \equiv s_2)
\]

\[
Estore(c_1^{i_1} \ldots c_{i+1}^{i_1})
\]

From lemma A.1 it follows that \( s'_1 \parallel s_2' \equiv s_1 \parallel r_2 \).

Furthermore, \( s'_1,\alpha_1 \in O[W;A] \) can easily be seen to follow from the observation that if \( \models Store(c_1^{i_1} \ldots c_{i+1}^{i_1}) \Rightarrow c \), with \( FV(c) \cap BV^I(c_1^{i_1} \ldots c_{i+1}^{i_1}) = \emptyset \), then \( \models Store(f_1^{i_1} \ldots f_{i+1}^{i_1}) \Rightarrow c \). In fact, given some first-order model, let \( \sigma \) be an assignment of values of that model to the variables. Then we have, by lemma A.1, that

\[
\sigma \models Store(f_1^{i_1} \ldots f_{i+1}^{i_1})
\]

implies

\[
\sigma \models Estore(c_1^{i_1} \ldots c_{i+1}^{i_1})
\]

Since \( BV^I(s_1) \cap BV^O(s_1) = \emptyset \), we have

\[
\models Estore(c_1^{i_1} \ldots c_{i+1}^{i_1}) \iff Estore(out(c_1^{i_1} \ldots c_{i+1}^{i_1})) \land Estore(in(c_1^{i_1} \ldots c_{i+1}^{i_1}))
\]

Furthermore, by definition,

\[
out(f_1^{i_1} \ldots f_{i+1}^{i_1}) = out(c_1^{i_1} \ldots c_{i+1}^{i_1}),
\]

therefore we have

\[
\sigma \models Store(out(c_1^{i_1} \ldots c_{i+1}^{i_1})) \land Estore(in(c_1^{i_1} \ldots c_{i+1}^{i_1}))
\]

From which we conclude, by the validity of \( Store(c_1^{i_1} \ldots c_{i+1}^{i_1}) \Rightarrow c \) and the restriction \( FV(c) \cap BV^I(c_1^{i_1} \ldots c_{i+1}^{i_1}) = \emptyset \), that \( \sigma \models c \). In a similar way it follows that \( s'_2,\alpha_2 \in O[W;B] \).
B  A property of the closure conditions

In this appendix we prove:

**Lemma 5.1** For arbitrary sets $S_1$ and $S_2$ of modular sequences which are closed under $P_2$ and the following version of $P_1$:

1. $s_1, c_1^{l_1}, \ldots, c_n^{l_n}, s_2, \alpha \in R \Rightarrow s_1, c_1^{l_1}, \ldots, c_m^{l_m}, s_2, \alpha \in R$ if $Estore(s_1, c_1^{l_1}, \ldots, c_n^{l_n}) \Leftrightarrow Estore(s_1, c_1^{l_1}, \ldots, c_m^{l_m})$

we have

$$\text{Closure}(\text{Closure}(S_1) \ || \ \text{Closure}(S_2)) = \text{Closure}(S_1 \ || \ S_2)$$

**Proof** It suffices to prove that $\text{Closure}(S_1) \ || \ \text{Closure}(S_2) \subseteq \text{Closure}(S_1 \ || \ S_2)$. Let $s_1, \alpha_1 \in S_1$ and $s_2, \alpha_2 \in S_2$. Furthermore let $s_1', \alpha_1 \in \text{Closure}(s_1, \alpha_1)$ and $s_2', \alpha_2 \in \text{Closure}(s_2, \alpha_2)$ such that $s_1', \alpha_1 \parallel s_2', \alpha_2$ is defined. We prove by induction on the number of applications of $P_1-O$, the closure condition $P_1$ for $\ell = 0$, that $s_1', \alpha_1 \parallel s_2', \alpha_2 \in \text{Closure}(S_1 \ || \ S_2)$.

The case that the number of applications of $P_1-O$ equals zero follows immediately from the closedness of $S_1$ and $S_2$ under $P_1-I$ and $P_2$.

For the sake of a smooth presentation let's introduce the notation $s \Rightarrow s'$ for the derivability of $s'$ from $s$ by one application of an arbitrary closure condition.

Now let

$$s_1 \Rightarrow s_{11}, c_1^{O}, \ldots, c_n^{O}, s_{12} \Rightarrow s_{11}, c_1^{O}, \ldots, c_m^{O}, s_{12} \Rightarrow s_1'$$

where $s_{11}, c_1^{O}, \ldots, c_m^{O}, s_{12} \Rightarrow s_1'$ consists only of applications of $P_1-I$ and $P_2$. We have

$$s_1' = s_{11}, c_1^{O}, d_1^{m-1}, c_2^{O}, \ldots, d_{m-1}, c_m^{O}, s_{12}$$

where the input constraints are introduced by $P_2$, $s_{11} \Rightarrow s_{11}'$ and $s_{12} \Rightarrow s_{12}'$, both derivations using only $P_1-I$ and $P_2$. (The slightly more general case that some of the $d_i$'s introduced are actually sequences of input constraints can be treated in the same way, but requires a more elaborate notation which might obscure the underlying idea.)

So we have

$$s_2' = s_{21}, c_1^{l_1}, d_1^{m-1}, c_2^{l_2}, \ldots, d_{m-1}, c_m^{l_m}, s_{22}$$

such that $s_1' \parallel s_2'$ and $s_{12}' \parallel s_{22}'$ are defined.

Define

$$r_1 = s_{11}, c_1^{O}, \ldots, c_n^{O}, d_1^{m-1}, s_{12}$$

It is not difficult to see that $s_1 \Rightarrow r_1$, where the number of applications of $P_1-O$ is one less than that in $s_1 \Rightarrow s_1'$. Similarly, $s_{12} \Rightarrow s_{12}'$.

Furthermore let

$$r_2 = s_{21}, c_1^{l_1}, \ldots, d_1^{m-1}, s_{22}$$

Again it is not difficult to check that $s_2 \Rightarrow r_2$, with the same number of applications of $P_1-O$ as in the derivation $s_2 \Rightarrow s_2'$.

So we are now in a position which allows us to apply the induction hypothesis: $r_1, \alpha_1 \parallel r_2, \alpha_2 \in \text{Closure}(S_1 \ || \ S_2)$.

Finally, we have $r_1, \alpha_1 \parallel r_2, \alpha_2 \Rightarrow s_1', \alpha_1 \parallel s_2', \alpha_2$, which we leave the reader to verify. \(\square\)
<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The formal specification and derivation of CMOS-circuits.</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes.</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a computer for evaluation of flow films.</td>
</tr>
<tr>
<td>85/04</td>
<td>T. Verhoeoff, H.M.L.J.Schols</td>
<td>Delay insensitive directed trace structures satisfy the foam</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the foam rubber wrapper postulate.</td>
</tr>
<tr>
<td>86/01</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems.</td>
</tr>
<tr>
<td>86/02</td>
<td>G.A. Bussing, K.M. van Hee, M. Voorhoeve</td>
<td>ELISA, A language for formal specification of information systems.</td>
</tr>
<tr>
<td>86/03</td>
<td>Rob Hoogerwoord</td>
<td>Some reflections on the implementation of trace structures.</td>
</tr>
<tr>
<td>86/04</td>
<td>G.J. Houben, J. Paredaens, K.M. van Hee</td>
<td>The partition of an information system in several systems.</td>
</tr>
<tr>
<td>86/05</td>
<td>J.L.G. Dietz, K.M. van Hee</td>
<td>A framework for the conceptual modeling of discrete dynamic systems.</td>
</tr>
<tr>
<td>86/06</td>
<td>Tom Verhoeoff</td>
<td>Nondeterminism and divergence created by concealment in CSP.</td>
</tr>
<tr>
<td>86/07</td>
<td>R. Gerth, L. Shira</td>
<td>On proving communication closedness of distributed layers.</td>
</tr>
<tr>
<td>86/09</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>Full abstraction of a real-time denotational semantics for an OCCAM-like language.</td>
</tr>
<tr>
<td>86/10</td>
<td>J. Hooman</td>
<td>A compositional proof theory for real-time distributed message passing.</td>
</tr>
<tr>
<td>86/11</td>
<td>W.P. de Roever</td>
<td>Questions to Robin Milner - A responder's commentary (IFIP86).</td>
</tr>
<tr>
<td>86/12</td>
<td>A. Boucher, R. Gerth</td>
<td>A timed failures model for extended communicating processes.</td>
</tr>
<tr>
<td>86/13</td>
<td>R. Gerth, W.P. de Roever</td>
<td>Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>86/14</td>
<td>R. Koymans</td>
<td>Specifying passing systems requires extending temporal logic.</td>
</tr>
<tr>
<td>87/01</td>
<td>R. Gerth</td>
<td>On the existence of sound and complete axiomatizations of the monitor concept.</td>
</tr>
<tr>
<td>87/02</td>
<td>Simon J. Klaver, Chris F.M. Verberne</td>
<td>Federatieve Databases.</td>
</tr>
<tr>
<td>87/03</td>
<td>G.J. Houben, J.Paredaens</td>
<td>A formal approach to distributed information systems.</td>
</tr>
<tr>
<td>87/04</td>
<td>T. Verhoeff</td>
<td>Delay-insensitive codes - An overview.</td>
</tr>
<tr>
<td>87/05</td>
<td>R. Kuiper</td>
<td>Enforcing non-determinism via linear time temporal logic specification.</td>
</tr>
<tr>
<td>87/06</td>
<td>R. Koymans</td>
<td>Temporele logica specificatie van message passing en real-time systemen (in Dutch).</td>
</tr>
<tr>
<td>87/07</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems with real-time temporal logic.</td>
</tr>
<tr>
<td>87/08</td>
<td>H.M.J.L. Schols</td>
<td>The maximum number of states after projection.</td>
</tr>
<tr>
<td>87/10</td>
<td>T. Verhoeff</td>
<td>Three families of maximally nondeterministic automata.</td>
</tr>
<tr>
<td>87/11</td>
<td>P. Lemmens</td>
<td>Eldorado ins and outs. Specifications of a database management toolkit according to the functional model.</td>
</tr>
<tr>
<td>87/12</td>
<td>K.M. van Hee and A. Lapinski</td>
<td>OR and AI approaches to decision support systems.</td>
</tr>
<tr>
<td>87/13</td>
<td>J.C.S.P. van der Woude</td>
<td>Playing with patterns - searching for strings.</td>
</tr>
<tr>
<td>87/14</td>
<td>J. Hooman</td>
<td>A compositional proof system for an occam-like real-time language.</td>
</tr>
<tr>
<td>87/16</td>
<td>H.M.M. ten Eikelder, J.C.F. Wilmont</td>
<td>Normal forms for a class of formulas.</td>
</tr>
<tr>
<td>87/17</td>
<td>K.M. van Hee, G.-J. Houben, J.L.G. Dietz</td>
<td>Modelling of discrete dynamic systems framework and examples.</td>
</tr>
<tr>
<td>Year/Volume</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>87/18</td>
<td>C.W.A.M. van Overveld</td>
<td>An integer algorithm for rendering curved surfaces.</td>
</tr>
<tr>
<td>87/19</td>
<td>A.J. Seebregts</td>
<td>Optimalisering van file allocatie in gedistribueerde database systemen.</td>
</tr>
<tr>
<td>87/20</td>
<td>G.J. Houben, J. Paredaens</td>
<td>The $R^2$-Algebra: An extension of an algebra for nested relations.</td>
</tr>
<tr>
<td>87/21</td>
<td>R. Gerth, M. Codish, Y. Lichtenstein, E. Shapiro</td>
<td>Fully abstract denotational semantics for concurrent PROLOG.</td>
</tr>
<tr>
<td>88/01</td>
<td>T. Verhoeff</td>
<td>A Parallel Program That Generates the Möbius Sequence.</td>
</tr>
<tr>
<td>88/03</td>
<td>T. Verhoeff</td>
<td>Settling a Question about Pythagorean Triples.</td>
</tr>
<tr>
<td>88/04</td>
<td>G.J. Houben, J. Paredaens, D. Tahon</td>
<td>The Nested Relational Algebra: A Tool to Handle Structured Information.</td>
</tr>
<tr>
<td>88/06</td>
<td>H.M.J.L. Schols</td>
<td>Notes on Delay-Insensitive Communication.</td>
</tr>
<tr>
<td>88/09</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>A Tutorial for Data Modelling.</td>
</tr>
<tr>
<td>88/10</td>
<td>J.C. Ebergen</td>
<td>A Formal Approach to Designing Delay Insensitive Circuits.</td>
</tr>
<tr>
<td>88/12</td>
<td>A.E. Eiben</td>
<td>Abstract theory of planning.</td>
</tr>
<tr>
<td>88/13</td>
<td>A. Bijlsma</td>
<td>A unified approach to sequences, bags, and trees.</td>
</tr>
<tr>
<td>88/14</td>
<td>H.M.M. ten Eikelder, R.H. Mak</td>
<td>Language theory of a lambda-calculus with recursive types.</td>
</tr>
</tbody>
</table>
An introduction to the category theoretic solution of recursive domain equations.

Bottom-up tree acceptors.

Executable specifications for discrete event systems.

Discrete event systems: concepts and basic results.

Fasering en documentatie in software engineering.

EXSPECT, the functional part.

Reconstruction of a 3-D surface from its normal vectors.

A systolic design for dynamic programming.

Some category theoretical properties related to a model for a polymorphic lambda-calculus.

Compositionality and modularity in process specification and design: A trace-state based approach.

Networks of Communicating Processes and their (De-)Composition.

Characterizations of Delay-Insensitive Communication Protocols.

A systematic design of a parallel program for Dirichlet convolution.

A general theory of genetic algorithms.

Discrete event systems: Dynamic versus static topology.

A new efficient implementation of CSP with output guards.

Algebraic specification and implementation of infinite processes.

A concise formal framework for data modeling.
<table>
<thead>
<tr>
<th>No.</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>89/14</td>
<td>H.C. Haesen</td>
<td>ELDA, data manipulatie taal.</td>
</tr>
<tr>
<td>89/15</td>
<td>J.S.C.P. van der Woude</td>
<td>Optimal segmentations.</td>
</tr>
<tr>
<td>89/16</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>Towards a framework for comparing data models.</td>
</tr>
<tr>
<td>89/17</td>
<td>M.J. van Diepen, K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra.</td>
</tr>
<tr>
<td>90/2</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Dynamic process creation in high-level Petri nets, pp. 19.</td>
</tr>
<tr>
<td>90/4</td>
<td>A. Peeters</td>
<td>Decomposition of delay-insensitive circuits, p. 25.</td>
</tr>
<tr>
<td>90/8</td>
<td>M.B. Josephs</td>
<td>Receptive Process Theory, p. 16.</td>
</tr>
<tr>
<td>90/9</td>
<td>A.T.M. Aerts, P.M.E. De Bra, K.M. van Hee</td>
<td>Combining the functional and the relational model, p. 15.</td>
</tr>
<tr>
<td>90/10</td>
<td>M.J. van Diepen, K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).</td>
</tr>
<tr>
<td>90/11</td>
<td>P. America, F.S. de Boer</td>
<td>A proof system for process creation, p. 84.</td>
</tr>
<tr>
<td>90/12</td>
<td>P. America, F.S. de Boer</td>
<td>A proof theory for a sequential version of POOL, p. 110.</td>
</tr>
<tr>
<td>90/14</td>
<td>F.S. de Boer</td>
<td>A proof system for the language POOL, p. 70.</td>
</tr>
<tr>
<td>90/15</td>
<td>F.S. de Boer</td>
<td>Compositionality in the temporal logic of concurrent systems, p. 17.</td>
</tr>
</tbody>
</table>
A fully abstract model for concurrent logic languages, p. 23.

On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences, p. 29.