A Unifying Approach to the Denotational Semantics of Communicating Sequential Processes

M.J. Walsteijn

RUU-CS-88-3
February 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

A Unifying Approach to the Denotational Semantics of Communicating Sequential Processes

M.J. Walsteijn

Technical Report RUU-CS-88-3
February 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht
the Netherlands

Abstract

In this report four different denotational semantics are given for a language based on communicating sequential processes. These four semantics are based on a number of approaches that were presented earlier as attempts of giving a formal definition of Hoare's original CSP. Unfortunately, the earlier approaches all used different variants of CSP that were conveniently chosen in order to simplify the presentation of the particular semantics or to avoid certain insurmountable difficulties in the approach.

In this report the four most successful denotational semantics developed for (variants of) CSP are adapted to one, fixed language based on communicating sequential processes. This makes it possible to understand the four semantic approaches as they materialize for one, common, unified base language, and compare the various semantics from a single, unified perspective.

This [illegible] Master's thesis [illegible]
[illegible] degree in Comp[illegible] Science [illegible] University [illegible]
Supervisor: [illegible]

Contents

Chapter 1

Introduction

In the last decade, many researchers have tried to solve the challenging problem of giving a denotational semantics to CSP [Hoare 78]. CSP is a language for describing communicating nondeterministic processes. The basic ingredients of CSP are assignment, selection, iteration, parallel composition, and input and output. CSP received much attention, maybe because of its elegance as a distributed programming language. It was one of the first languages providing an elegant set of communication constructs, which have had a considerable impact on the design of programming languages for multitasking environments (e.g. Ada [ANSI]).

The statement types of CSP are all standard, except parallel composition. A parallel composition is a statement which causes its components to execute concurrently. These components, usually called processes, communicate via input and output instructions (receive and send) that must match as in a handshake. A handshake is a simultaneous activity (like receive and send) of two processes, which can be exploited for both message passing and synchronization. Furthermore, all processes combined in a parallel composition have disjoint variable sets.

Input and output instructions may occur in selection and iteration statements. A selection has the form $[b_1; c_1 \longrightarrow s_1 \Box \ldots \Box b_m; c_m \longrightarrow s_m]$, where b_i are boolean conditions, c_i are input or output instructions, $b_i; c_i$ are called guards and s_i are arbitrary program fragments. If (for a certain i) b_i is true and the communication partner indicated by c_i is ready to execute the corresponding input or output instruction (the guard $b_i; c_i$ evaluates to true in that case), then the communication may take place and the corresponding s_i is executed. If several guards are true then an alternative is chosen nondeterministically. An iteration $*SEL$, where SEL is a selection, consists of the repeated execution of SEL until all guards evaluate to false.

Four formal (denotational) definitions for CSP have been proposed in the literature. In [Broy 86], the semantics of a CSP-program is a function describing its behavior. This function gives a row of reactions (success or rejection) when confronted with an arbitrary row of proposals for communication. The function indicates whether the row of communications is realizable, i.e. executable in that order, by the program. This kind of functions resemble the ones in [Hoare 85], where they

were used to implement (!) the concurrency theory presented there.

Another approach was followed in [FLP] and in [Sound]. Let a state be an assignment of values to all variables and a communication history be a row of communications. The meaning of a CSP-process now is a function mapping a pair consisting of a state and a communication history to a set of pairs, again consisting of a state and a communication history. Such a function reflects the set of possible computations of a process: given an initial state and an initial history (e.g. the empty row), the function yields the set of results of all possible computations. The result of a computation by a process is the final state of the computation combined with the row of communications needed for that computation. Whether such a computation of a process is realizable depends on the other processes of the program. For example, there should be processes providing the counterparts of the send and receive instructions. From the a priori semantics of the individual processes, the semantics of the entire program is constructed. The approaches in [FLP] and [Sound] differ considerably in the way these basic ideas are elaborated.

Another semantics of CSP was presented in [FHLdeR]. Here, the semantics of a process is a tree. Every path in the tree is a possible computation. The domain of trees is defined by a domain equation. In fact, this will be the only domain defined by a domain equation in the entire report. The meaning of a CSP-program is constructed from the trees of the individual processes in the program.

All four approaches used different variants of CSP in order to simplify the presentation of the particular semantics or to avoid insurmountable difficulties in the approach. The variant used in [Broy 86] will be called CSP-B. For example, in CSP-B receive statements have the form $x?$, where x is a local variable. This means that input for x from any other process is expected. Send statements have the form $x!E$, meaning that the value of expression E is to be sent to the variable x (where x is owned by another, unique process). These two corresponding actions are executed simultaneously: the process which is first, waits. No assignment to or reading of x is allowed to non-owning processes. This mechanism is in contrast to CSP, where $P_i?x$ (receive) indicates that input is expected from process P_i. The output statement of CSP is of the form $P_j!E$, and mentions the destination process P_j instead of a variable.

The other variants of CSP will be denoted as CSP-FLP, CSP-S and CSP-FHLdeR. The deviations of the various CSP variants from CSP are shown in table 1.1.

The notions of nested concurrency, mixed guards and the distributed termination convention will now be explained. We speak of *nested concurrency* if a process may contain a parallel composition of several other processes. *The distributed termination convention* defines when a guard, in particular the communication part of a guard, is false. Generally, a guard evaluates to false if its boolean part is false. Additionally, the distributed termination convention states that the communication part of a guard is false if the communication partner indicated by that guard has terminated. This implies that a loop exits if all communication partners indicated by the guards of that loop have terminated. A variant of CSP features *mixed guards*

	nested conc.	mixed guards	distr. term. conv.
CSP	+	+	+
CSP-B	−	+	−
CSP-FLP	+	+	−
CSP-S	−	+	+
CSP-FHLdeR	−	−	+

Table 1.1: Deviations from CSP.

if mixtures of purely boolean guards (guards with empty communication part) and guards containing a communication within one selection or iteration statement are allowed.

In this report, all four semantic approaches to CSP are adapted to one, unified base variant of CSP. This base variant, called CSP-W in the sequel, is roughly the intersection of CSP-FLP, CSP-S and CSP-FHLdeR and hence very similar to CSP, but does not feature nested concurrency, mixed guards and the distributed termination convention. This makes it possible to compare the four semantic models of CSP for one, unified base variant of the language. The topics of the comparison will be:

- Simplicity: of domains, orderings and denotations.
- Abstractness: to what degree do 'equivalent' program fragments have identical denotations, on the condition that 'inequivalent' program fragments have different denotations?
- Generality: is it possible to easily define language constructs of CSP which are not present in CSP-W, in the particular formalism?
- Compositionality: is the semantic definition of the language obtained by induction on the syntax?
- Continuity: are all functions expressing the computation of programs continuous? (This is necessary in order to apply the theory developed for proving properties of programs.)

All definitions used in this report are completely general. This is in contrast to e.g. [FHLdeR], where selections and iterations are defined by means of example cases. Although this difference is not crucial, definition by example sometimes hides the complexity of a method.

It is worth mentioning that there also exists a weakest precondition semantics of CSP, see [EF]. However, Elrad and Francez had to give up the property of compositionality: their definition is not based on induction on the syntax and is therefore not

completely satisfying. Recently, another variant of CSP was proposed (see [ABC]). This variant synthesized from observing the structure of many correct CSP programs. Both approaches will not be discussed in this paper.

People have found it difficult to give a proper mathematical semantics to CSP; some researchers thought that the language is inherently difficult and some of them diverted their attention to a theory of communicating sequential processes [BHR] in which no assignments occur. In this theory the primitive elements are (abstract) atomic actions, and therefore this theory is much less a programming language than CSP: it is an alternative to concurrency theories like CCS [Milner] and process algebra [BK]. Unlike assignments, these atomic actions do not influence the communications that may take place; an assignment may influence the choice of an alternative in a selection. This theory is essentially different from CSP, although it is called CSP in [Hoare 85]. A more appropriate name would be TCSP, with the "T" standing for "Theory".

Throughout this report it is assumed that the reader is familiar with the techniques used in the denotational semantics of sequential programming languages (see e.g. [Stoy], [SM], [MA]).

The rest of this report is organized as follows. In chapter 2 the language which will be used (CSP-W) is defined informally and an example program is given. In chapter 3 a formal definition of CSP-W is given, based on the model in [Broy]. In chapter 4 a semantics is given based on the model in [FLP]. Chapter 5 contains a semantics based on the ideas in [Sound] and chapter 6 contains a semantics based on [FHLdeR]. Each of the four chapters contains a section on domains and definitions, a section on the denotations and a section with a discussion on the method. In chapter 7 some conclusions are listed and the comparison is carried out.

Chapter 2

The language CSP-W

This chapter introduces the language which will be used in this report. Four formal definitions of this language will be given in the subsequent chapters.

2.1 Syntax and informal semantics of CSP-W

The language will be very similar to Hoare's original CSP with some exceptions: no nested concurrency and mixed guards are allowed and no distributed termination convention is enforced. The syntax of CSP-W in B.N.F. is as follows (identifiers, expressions, process names and booleans are left unspecified):

<program>	::=	|[<processes>]|
<processes>	::=	<process> |
		<processes> || <processes>
<process>	::=	<proname> :: <command>
<command>	::=	skip | <id>:=<exp> |
		<iocommand> |
		<command>;<command> |
		<selection> | <iteration>
<iocommand>	::=	<proname>!<exp> | <proname>?<id>
<selection>	::=	[<gcs>]
<gcs>	::=	<pbgcs> | <comgcs>
<pbgcs>	::=	<pbgc> | <pbgcs> □ <pbgcs>
<pbgc>	::=	<pbguard> ⟶ <command>
<comgcs>	::=	<comgc> | <comgcs> □ <comgcs>
<comgc>	::=	<comguard> ⟶ <command>
<pbguard>	::=	<boolean>
<comguard>	::=	<boolean>;<iocommand> | <iocommand>
<iteration>	::=	* <selection>

The informal semantics of CSP-W will indicate the form and intended meaning of a program written in CSP-W.

A program consists of the parallel composition (||) of one or more concurrent processes. The execution of a multiple-process program may be conceived as being an arbitrary interleaving of the actions of the individual processes or as the real concurrent execution of the processes (multiprogramming or real concurrency). In both cases, the only restriction is that corresponding sends and receives must happen simultaneously. The state of a program (process) is defined as an assignment of values to all program (process) variables. For a state σ, an identifier x and data element d, we denote the updated state by $\sigma[d/x]$, i.e.

$\sigma[d/x](x) = d$
$\sigma[d/x](y) = \sigma(y)$ for $y \neq x$

It is required that all process names in a program are distinct. Each process consists of a sequential program labeled by a process name. Variables are strictly local to a process and are not exported across process boundaries (hence there are no shared variables).

In the sequel $P_1,\ldots,P_n$ denote the distinct names of the n concurrent processes of a program. Communication between processes P_i and P_j takes place by process P_i executing a command or guard having the form $P_j!E$ and P_j executing $P_i?x$. This execution has to be simultaneous, and its effect is the assignment to x in P_j of the value of E in P_i at the moment of the communication. In other words, P_i outputs the value E to P_j, which inputs it in a handshake with P_i.

The execution of skip immediately terminates with unaffected state, without performing any communication. The execution of $x{:=}E$ involves the evaluation of E and assignment of the value of E in the current state to x. An instruction of the form command1;command2 is executed as the usual sequential composition: the execution of command1 is followed by the execution of command2.

There are two forms of selections: either all guards are purely boolean or all guards contain a communication. Mixed guards are not permitted. The two types of selection statement have the following meaning:

1. $[b_1 \longrightarrow$ command1 $\Box \ldots \Box\, b_m \longrightarrow$ commandm] is a selection statement which involves the selection of a true boolean b_i and execution of the corresponding commandi. If there is more than one b_i true then any alternative with a true boolean is chosen nondeterministically. If there are no true b_i then the execution of the entire program is aborted.

2. $[b_1; c_1 \longrightarrow$ command1 $\Box \ldots \Box\, b_m; c_m \longrightarrow$ commandm] (where b_i is true by default if it is missing and c_i is an iocommand) is a selection statement which involves the selection of an enabled guard $b_i; c_i$, and the subsequent execution of c_i and then commandi. A guard is enabled if b_i is true and the communication partner indicated by c_i is ready to communicate. The execution of c_i again requires the simultaneous execution of the iocommand of the current process

and its communication partner. If no indicated partner in any guard with a true boolean is ready to communicate, then the whole command is delayed. In case of more than one enabled guard, again any alternative that is enabled is chosen nondeterministically. Furthermore, if there are no true b_i then the entire program is aborted.

Let C be a selection statement of the form $[g_1 \longrightarrow$ command1 $\Box \ldots \Box g_m \longrightarrow$ commandm], either with all guards g_i purely boolean or with each guard g_i containing a communication. The execution of the iterative command $*C$ consists of the repeated execution of C and continues until all b_i's of the guards are false. Note that a loop in which one of the b_i's is always true (in every iteration of the loop this may be a different one) never terminates even in the case of iterations with all guards containing a communication. A special case of this is, that a loop in which all guards are purely communication guards (i.e. , all b_i's missing) never terminates.

The given interpretation of selection and iteration deviates from Hoare's formulation for the case of CSP, which considers the communication part of a guard to be false if its communication partner has terminated (the distributed termination convention). However, it is possible to encode the sensing of termination of other processes by additional explicit communications. Therefore, this deviation is not crucial.

Note that the language does not provide for nested concurrency; the syntax prohibits this.

2.2 An example of a CSP-W program

In this section an example of a program in CSP-W is given. The problem was taken from [Moitra].

Problem: Given two disjoint and nonempty sets of integers S and T, $S \cup T$ has to be partioned into two subsets A and B such that |S|=|A|, |T|=|B| and every element of A is smaller than any element of B.

A simple, sequential solution of the problem swaps the maximum of S and the minimum of T while the maximum of S is greater than the minimum of T. This can be encoded in the following sequential CSP-W program, where max, min, + and - are maximum, minimum, set union and set difference, respectively.

$$
\begin{array}{l}
|[Partition :: maxS := max(S); \\
\qquad\qquad minT := min(T); \\
\qquad\qquad *[maxS > minT \longrightarrow S := S - \{maxS\} + \{minT\}; \\
\qquad\qquad\qquad\qquad\quad T := T - \{minT\} + \{maxS\}; \\
\qquad\qquad\qquad\qquad\quad maxS := max(S); \\
\qquad\qquad\qquad\qquad\quad minT := min(T)] \\
]|
\end{array}
$$

Next, we transform the program into a distributed one. Process P_1 computes the maximum of S (a local set), receives the minimum of T (a remote set for P_1) from P_2 and sends its computed maximum to P_2. Now it is possible to perform the same test as in the sequential program and to adjust S accordingly. After this, the process is repeated. Process P_2 is completely symmetric.

Globally, the following happens. First, the maximum of S and the minimum of T are computed concurrently. Next, these computed values are exchanged and then the test of the sequential program is possible. If all elements of S are smaller than all elements of T then both P_1 and P_2 exit their loops, because P_1 and P_2 perform the same test. Therefore, the entire program terminates. In the other case, S and T are updated concurrently and the entire computation is repeated. Note how the communications synchronize the processes. Furthermore, one could perform the test in only one process and send it to the other, but this breaks the symmetry. Moreover, a communication is (nowadays) more expensive than a test. Therefore, in the following CSP-W program, no leader process is present.

$$
\begin{array}{l}
[\![P_1 :: maxS := max(S); \\
\quad P_2?temp1; \\
\quad P_2!maxS; \\
\quad *[maxS > temp1 \longrightarrow S := S - \{maxS\} + \{temp1\}; \\
\qquad\qquad\qquad\qquad\quad maxS := max(S); \\
\qquad\qquad\qquad\qquad\quad P_2?temp1; \\
\qquad\qquad\qquad\qquad\quad P_2!maxS] \\
\\
\| \; P_2 :: minT := min(T); \\
\quad P_1!minT; \\
\quad P_1?temp2; \\
\quad *[temp2 > minT \longrightarrow T := T - \{minT\} + \{temp2\}; \\
\qquad\qquad\qquad\qquad\quad minT := min(T); \\
\qquad\qquad\qquad\qquad\quad P_1!minT; \\
\qquad\qquad\qquad\qquad\quad P_1?temp2] \\
]\!]
\end{array}
$$

This program is at least as elegant as the one in [Moitra], which uses the distributed termination convention.

Chapter 3

A semantic model of CSP-W using stream processing functions

In this chapter we give a denotational semantics of CSP-W in the spirit of [Broy 86]. Broy uses stream processing functions in his formal definition of CSP-B. The idea of stream processing functions will be applied to CSP-W, which will require several adaptions of the semantic model of Broy.

3.1 The notion of observable behavior

For a better understanding and a motivation of the model of stream processing functions, a particular model of experiment for testing the identity of a process is introduced. The testing machinery consists of a processing unit in which the CSP-W program is loaded, and of a display together with a keyboard on which actions can be typed in. After loading the program, the program variables have an undefined initial value (the initial state). The experiment now proceeds as follows, based on information shown on the display:

1. The display may show 'terminated' and a list of the values of the variables: the final state. Then the experiment is finished.
2. Otherwise, an arbitrary communication action is chosen by the experimenter. The action is typed in via the keyboard. Below, this action will be called an *offer*.
3. The display may show either 'rejected', 'accepted' or the data value that is communicated between two of the processes of the program. It is also possible that the machine does not give an answer at all (divergence).
4. The experimenter may finish the experiment or may start all over again with 1.

If the display shows 'terminated', then further input via the keyboard cannot change this.

In this model, assignments are not regarded as observable atomic actions but as silent steps ([Milner]), which are not observable. Likewise, skip is a silent step. The communications, however, are regarded as observable atomic actions.

In Broy's model, the semantics of a CSP-W program is defined as the behavior of a program in an arbitrary experiment. An experiment is a stream of offers which correspond to communication actions. An offer (a proposal of a communication) is possible if the offer corresponds to the next communication instruction in one of the processes in the program. If the proposed communication is possible then the action is executed and reacted to, and the executing process advances internally to a next observable action, i.e. a communication action. If the offer is not possible, then it is rejected and the program considers the next offer in the stream of offers, never reconsidering the rejected offer.

The stream of reactions (acceptances, rejections, data values) together with the final state as a function of the stream of offers is called the behavior of a CSP-W program. This explains the term stream processing functions: the semantics of a CSP-W program is a stream processing function. In other words, the meaning of a CSP-W program can be determined by exhaustively testing the behavior of the program to various experiments. Obviously, exhaustive testing is not possible in practice. The idea of physical testing machinery is just used to support the intuition. These ideas will be formalized in the next section.

3.2 The domains

$STATE \stackrel{\text{def}}{=} \text{<id>} \rightarrow D$

where D is assumed to be a given set of data.

$STATE^{\perp}$ is the flat domain defined by $STATE$ and $\perp$ (undefined). A state is the association of a value from D to every variable in <id>. Divergence is represented by $\perp$. An abortion is not distinguished from the entering of a nonterminating loop and therefore represented by $\perp$ too.

$$OFFER \stackrel{\text{def}}{=} (\text{<proname>} \times \text{<proname>}) \cup (\text{<proname>} \times \text{<proname>} \times \text{<id>} \times D)$$
$$REACT \stackrel{\text{def}}{=} \{R, A\} \cup D$$

$OFFER$ contains two types of elements. An offer can be an element having the form (*name1*,*name2*), which is a proposal to process *name1* to output to process *name2*. The reaction is R (Rejection) if the next instruction of *name1* is not *name2*!E or if the proposal is not handled by process *name1*. Otherwise, the reaction to this offer is the value of the expression in the output instruction.

Another form of offer is (*name1*,*name2*,x,d), which is a proposal to process *name2* to input from process *name1* the value d. This value should be accepted as

the new value of x. The reaction to this offer is R if the next instruction of *name2* is not *name1*?x or if the proposal is not handled by process *name2*. Otherwise the reaction to this proposal is A (Acceptance). This also explains the definition of $REACT$.

Given a set X, the set $STREAM(X)$ is defined by

$STREAM(X) \stackrel{\text{def}}{=} X^* \cup X^\infty$

Concatenation on streams is denoted by $++$. The empty stream is denoted by ϵ. For $a \in X$ the one element stream is $[a]$. For $a \in X$, $s \in STREAM(X)$, $a : s$ is written for $[a]{+}{+}s$. Furthermore, $\perp : s$ is defined as ϵ (for any s).

Several functions are used on streams:

first: $STREAM(X) \rightarrow X^\perp$
rest: $STREAM(X) \rightarrow STREAM(X)$
$|\ |$: $STREAM(X) \times INT \rightarrow STREAM(X)$
prefix: $STREAM(X) \times STREAM(X) \rightarrow BOOL$
drop: $STREAM(X) \times STREAM(X) \rightarrow STREAM(X)$

where

first$(\epsilon) = \perp$
first$(a : s) = a$
rest$(\epsilon) = \epsilon$
rest$(a : s) = s$
$|s|_0 = \epsilon$
$|s|_{n+1}$= if $s \neq \epsilon$ then first(s): $|\text{rest}(s)|_n$ else ϵ
prefix(ϵ, s) =true
prefix$(x : p', \epsilon)$ =false
prefix$(x : p', y : s') =$ if $x = y$ then prefix(p', s') else false
drop$(\epsilon, s) = s$

and, for $x, y \neq \perp$, $r1$ infinite and r finite:

drop$(x : r, y : s) =$ drop(r, s)
drop$(r1, s) = \epsilon$

The functions first and rest do not need further explanation. By $|s|_n$ the prefix of s of length n is denoted. The function prefix tests whether its first argument is a prefix of its second argument. The function drop throws away a prefix of its second argument. The length of this prefix is determined by the length of the first argument of drop. If the latter length is infinite then the result of drop is the empty

stream. Note that drop is not defined if the second argument is longer than the first argument, except when the first argument is infinite.

Now all ingredients are available for the most important definition of this section. The meaning of a CSP-W program will be modelled by a function in

$$DENOT \stackrel{\text{def}}{=} STATE^{\perp} \rightarrow (STREAM(OFFER) \rightarrow P(STREAM(REACT) \times STATE^{\perp}))$$

In this definition the semantics of a CSP-W program is a function from an initial state to a function from a stream of offers to a set of pairs: streams of reactions and final states. The powerset P in the definition of $DENOT$ reflects the nondeterminism of a CSP-W program: more than one outcome is possible with the same initial state and stream of offers.
$STREAM(X)$ for every X and $STREAM(REACT) \times STATE^{\perp}$ form domains if ordered by

$S1 \sqsubseteq S2$ iff $S1$ is a prefix of $S2$, and
$(S1, \sigma 1) \sqsubseteq (S2, \sigma 2)$ iff $(\sigma 1 = \perp \wedge S1 \sqsubseteq S2) \vee ((S1, \sigma 1) = (S2, \sigma 2))$,

respectively.

3.3 The semantic equations

Let V :<exp>$\rightarrow$ $(STATE^{\perp} \rightarrow D^{\perp})$ be a function, which gives meaning to boolean and arithmetical expressions without side-effects, i.e. , V associates with every expression a function from a state to a value (for every state the expression has a certain value).

We now give the formal definition of F :<process>$\rightarrow$ $DENOT$, together with the informal explanation of each clause in the definition. F gives meaning to individual processes. Given an initial state σ, it assigns a stream processing function to a process. F is defined by induction on the syntax: the meaning of a composite language construct is defined in terms of the meaning of its constituents. Every CSP-W language construct which may occur within a CSP-W process is defined by F in a separate clause. We will write $F_{\sigma}[t]$ as a shorthand for $F(P_i :: t)(\sigma)$. Note that in $F_{\sigma}[t]$ the term t occurs in process P_i by convention. Similar notations are used for other functions. s will denote an arbitrary stream of offers, i.e. $s \in STREAM(OFFER)$.

(1) $F_{\perp}[t](s) = \{(\epsilon, \perp)\}$

In the sequel we let σ be arbitrary, with $\sigma \in STATE$.

(2) skip

$F_\sigma[\text{skip}](s) = \{(\epsilon, \sigma)\}$

Obvious.

(3) assignment

$$F_\sigma[x := E](s) = \begin{cases} \{(\epsilon, \bot)\} & \text{if } V_\sigma[E] = \bot \\ \{(\epsilon, \sigma[V_\sigma[E]/x])\} & \text{otherwise} \end{cases}$$

The meaning of assignment is the appropriate update to the state σ. In cases like division by zero V yields $\bot$. Like skip, assignment is not an observable action. Therefore, the stream of offers is disregarded and no reaction is output.

(4) output

$$F_\sigma[P_j!E](s) = \begin{cases} \{(V_\sigma[E] : \epsilon, \sigma)\} \text{ if } \text{first}(s) = (P_i, P_j) \wedge i \neq j \wedge V_\sigma[E] \neq \bot \\ \{(\epsilon, \bot)\} \text{ if } s = \epsilon \vee V_\sigma[E] = \bot \\ \{(R : s', \sigma') \mid (s', \sigma') \in F_\sigma[P_j!E](\text{rest}(s))\} \text{ otherwise} \end{cases}$$

Output has no side effect. If the presented offer $\text{first}(s)$ is not in correspondence with the output command, then it is rejected and the next offer in the stream is considered. Note that the second of the names in the offer $\text{first}(s)$ has to match the one in the output command and the first name the name of the process in which the output command occurs. See section 3.2 for further explanation.

(5) input

$$F_\sigma[P_j?x](s) = \begin{cases} \{(A : \epsilon, \sigma[d/x])\} \text{ if } \text{first}(s) = (P_j, P_i, x, d) \wedge i \neq j \\ \{(\epsilon, \bot)\} \text{ if } s = \epsilon \vee i = j \\ \{(R : s', \sigma') \mid (s', \sigma') \in F_\sigma[P_j?x](\text{rest}(s))\} \text{ otherwise} \end{cases}$$

The side effect of the input command is the update to the state corresponding to the assignment of d to x. The value d corresponds to the value in the offer $\text{first}(s)$. Remarks, similar to those for (4) can be made here again. In addition, the variable

x in the offer first(s) has to match the one in the input command.

(6) sequential composition

$$F_\sigma[C1;C2](s) = \{(s1{+}{+}s2, \sigma 2) \mid (s1, \sigma 1) \in F_\sigma[C1](s) \wedge (s2, \sigma 2) \in F_{\sigma 1}[C2](\mathrm{drop}(s1, s))\}$$

Note that the number of reactions of $C1$ is equal to the number of offers tested by $C1$. Therefore drop throws away the prefix of the experiment s consisting of the offers tested by $C1$. Consequently, the input to F applied to $C2$ are the modified state $\sigma 1$ and the stream consisting of the rest of the offers (the remainder of the experiment). If the stream of reactions of $C1$ is infinite then the remainder of the experiment, which is meant as input to $C2$, is empty. The stream of reactions produced by $C1;C2$ simply is the concatenation of the streams produced by $C1$ and $C2$.

(7) selection with all guards purely boolean

The semantics of a selection of the form:
$SEL1 = [b_1 \longrightarrow \mathrm{command1} \,\Box \ldots \Box\, b_m \longrightarrow \mathrm{commandm}]$
is as follows:

$$\begin{aligned} F_\sigma[SEL1](s) = &\{p \mid \exists i(p \in F_\sigma[\mathrm{commandi}](s) \wedge V_\sigma[b_i] = \mathit{true})\} \\ &\cup \{(\epsilon, \perp) \mid \exists i(V_\sigma[b_i] = \perp) \vee \forall i(V_\sigma[b_i] = \mathit{false})\} \end{aligned}$$

In a selection with all guards purely boolean, the observable actions can only occur in the commands of the guarded commands. In other words, the selection of an alternative itself is not observable. Furthermore, the choice of an alternative does not affect the state. This explains the first set in the definition of selection $SEL1$. In this set all possible behaviors are contained, which are due to local nondeterminism.

An abortion takes place if the value of one of the boolean expressions is undefined or if all boolean expressions yield false. This explains the second set in the definition of selection $SEL1$.

(8) selection with all guards containing a communication

Every selection with all guards containing a communication has the form:

$$\begin{aligned} SEL2 = [&b_1; P_{i_1}!E_1 \longrightarrow \mathrm{command1} \,\Box \ldots \Box\, b_r; P_{i_r}!E_r \longrightarrow \mathrm{commandr} \,\Box \\ &b'_1; P_{j_1}?x_1 \longrightarrow \mathrm{command1'} \,\Box \ldots \Box\, b'_s; P_{j_s}?x_s \longrightarrow \mathrm{commands'}] \end{aligned}$$

If there exists i_k such that $i_k = i$ or j_l such that $j_l = i$ then $F_\sigma[SEL2](s) = \{(\epsilon, \perp)\}$. Thus, without loss of generality we may assume that

$i_1, \ldots, i_r, j_1, \ldots, j_s \in \{1, \ldots, i-1, i+1, \ldots, n\}$, where n is the number of processes in the program.

$F_\sigma[SEL2](s) =$

$\{(R : r, \sigma') \mid \exists n1, n2 \in$<proname> $(\text{first}(s) = (n1, n2) \wedge \forall k(V_\sigma[b_k] = true \rightarrow$
$(n2 \neq P_{i_k} \vee n1 \neq P_i))) \wedge (r, \sigma') \in F_\sigma[SEL2](\text{rest}(s))\}$

$\cup \{(a : r, \sigma') \mid \exists k(\text{first}(s) = (P_i, P_{i_k}) \wedge V_\sigma[b_k] = true \wedge V_\sigma[E_k] = a \wedge$
$((a = \perp \wedge \sigma' = \perp \wedge r = \epsilon) \vee (a \neq \perp \wedge (r, \sigma') \in F_\sigma[\text{commandk}](\text{rest}(s)))))\}$

$\cup \{(R : r, \sigma') \mid \exists n1, n2 \in$<proname>$, x \in$<id>$, d \in D(\text{first}(s) = (n1, n2, x, d) \wedge$
$\forall l(V_\sigma[b'_l] = true \rightarrow (n1 \neq P_{j_l} \vee n2 \neq P_i \vee x \neq x_l))) \wedge$
$(r, \sigma') \in F_\sigma[SEL2](\text{rest}(s))\}$

$\cup \{(A : r, \sigma') \mid \exists n1 \in$<proname>$, x \in$<id>$, d \in D(\text{first}(s) = (n1, P_i, x, d) \wedge$
$\exists l(n1 = P_{j_l} \wedge V_\sigma[b'_l] = true \wedge x = x_l \wedge$
$(r, \sigma') \in F_{\sigma[d/x]}[\text{commandl'}](\text{rest}(s))))\}$

$\cup \{(\epsilon, \perp) \mid \exists k(V_\sigma[b_k] = \perp) \vee \exists l(V_\sigma[b'_l] = \perp) \vee s = \epsilon \vee$
$(\forall k(V_\sigma[b_k] = false) \wedge \forall l(V_\sigma[b'_l] = false))\}$

An offer first(s) is rejected if for all guards of the selection statement the following holds: if the boolean expression yields true, then the offer does not correspond to the input or output command of the guard.

An offer, which proposes an input command is accepted if it corresponds to the input command of a guard in the selection and the boolean expression involved yields true. With the corresponding alternative and updated state is continued.

If the offer corresponds to an output guard and the boolean expression involved yields true, then the reaction consists of the value of the expression of that output guard. If this value is defined, then execution is continued with the corresponding command . Otherwise the effect is an abortion, i.e. $(\epsilon, \perp)$. (Recall that $\perp : r$ is defined as ϵ).

An abortion occurs if one of the boolean expressions in the guards is undefined or if all boolean expressions are false or if the input stream is empty.

(9) iteration

The semantics of iteration is given by a fixed point equation. For the two different types of selection, the semantics of iteration is defined in the same way. An arbitrary selection has the form:
$SEL = [g_1 \longrightarrow \text{command1} \,\square \ldots \square\, g_m \longrightarrow \text{commandm}]$

where SEL contains either purely boolean guards or guards with a communication.

$F_\sigma[*SEL] =$ if C then F_σ[skip] else $F_\sigma[ONESTEP]$

where

$$ONESTEP = [g_1 \longrightarrow \text{command1}; *SEL \Box \ldots \Box g_m \longrightarrow \text{commandm}; *SEL]$$

$$C = \begin{cases} \forall i(V_\sigma[b_i] = false) & \text{if } SEL \text{ is } SEL1 \text{ type} \\ \forall k(V_\sigma[b_k] = false) \wedge \forall l(V_\sigma[b'_l] = false) & \text{if } SEL \text{ is } SEL2 \text{ type} \end{cases}$$

A loop with all guards purely boolean is exited if all booleans are false. If all guards contain a communication, then the loop is exited if the boolean parts of all guards are false.

Note that if SEL is a $SEL2$ type selection, then it follows from this definition that the termination of other processes is not sensed, i.e., there is no distributed termination convention.

(10) parallel composition

The parallel composition of processes in CSP-W is handled in two steps. In the first step an auxiliary, binding function B is defined in terms of F. In the second step a function M is defined, which gives meaning to CSP-W programs with the aid of function B of step one.

(10.1) **step 1.**

For the first step an auxiliary function B (of binding): <processes>$\rightarrow DENOT$ is introduced. Here, the case of two processes is given. The case of more than two processes is handled by considering the parallel composition of two processes as a new process. More precisely, we consider the result of the binding of two processes to be a new process. The order of this binding is irrelevant, since it can be proved that B is associative and commutative.

$B_\sigma[P_1 :: \text{command1} \| P_2 :: \text{command2}](s) =$

$$\{p \in \text{join}(f1, f2, s) \mid f1, f2 \in [STREAM(OFFER) \rightarrow STREAM(REACT) \times STATE^\perp] \wedge \forall t \in STREAM(OFFER)(f1(t) \in F(P_1 :: \text{command1})(\sigma)(t) \wedge f2(t) \in F(P_2 :: \text{command2})(\sigma)(t))\}$$

where the function join on two functions (see the definition of B for their type) and

a stream of offers is defined by:

$$\begin{aligned}\text{join}(f1, f2, x : s) = &\{(y : t, \sigma') \mid (t, \sigma') \in \text{join}(f1_x, f2_x, s) \wedge y = \text{first}'(f1(x : s)) \wedge \\ &\quad \neg(y \in \{R, \bot\})\} \\ &\cup \{(y : t, \sigma') \mid (t, \sigma') \in \text{join}(f1_x, f2_x, s) \wedge y = \text{first}'(f2(x : s)) \wedge \\ &\quad \neg(y \in \{R, \bot\})\} \\ &\cup \{(\epsilon, \text{upd}(\sigma, \sigma1, \sigma2)) \mid f1(x : s) = (\epsilon, \sigma1) \wedge f2(x : s) = (\epsilon, \sigma2) \wedge \\ &\quad \sigma1, \sigma2 \neq \bot\} \\ &\cup \{(R : t, \sigma') \mid (t, \sigma') \in \text{join}(f1_x, f2_x, s) \wedge \\ &\quad ((\text{first}'(f1(x : s)) = R \wedge \text{terminal}(f2(x : s))) \vee \\ &\quad (\text{first}'(f2(x : s)) = R \wedge \text{terminal}(f1(x : s))) \vee \\ &\quad (\text{first}'(f1(x : s)) = \text{first}'(f2(x : s)) = R))\} \\ &\cup \{(\epsilon, \bot) \mid f1(x : s) = (\epsilon, \bot) \vee f2(x : s) = (\epsilon, \bot)\}\end{aligned}$$

where upd, terminal, f_x, and first$'$ are defined by

$$\text{upd}(\sigma, \sigma1, \sigma2)(x) = \begin{cases} \sigma1(x) & \text{if } \sigma1(x) \neq \sigma(x) \wedge \sigma2(x) = \sigma(x) \\ \sigma2(x) & \text{if } \sigma2(x) \neq \sigma(x) \wedge \sigma1(x) = \sigma(x) \\ \sigma(x) & \text{if } \sigma(x) = \sigma1(x) = \sigma2(x) \end{cases}$$

$$\text{terminal}((s, \sigma)) = (s = \epsilon \wedge \sigma \in STATE)$$

$$\text{first}'((t, \sigma)) = \text{first}(t)$$

$$f(x : s) = (s1, \sigma) \Rightarrow f_x(s) = (\text{rest}(s1), \sigma)$$

The different f1's and f2's represent the different paths of computation, which command1 and command2 may take due to nondeterminism. The effect of join on $f1$, $f2$ and s could be pictured as follows:

First, the behaviors $(r1, \sigma1)$ and $(r2, \sigma2)$ of command1 and command2 to the experiment s are calculated. The result of the join is a particular kind of 'merge' of $r1$ and $r2$ and an updated state. This state is the original state updated by both the changes of command1 to the original state and the changes of command2 to the original state. Because all variables are local to a process, it is not possible that both command1 and command2 change the same variable (all names of variables could be thought prefixed with the name of the process; with $\sigma[d/x], \sigma[d/P_i x]$ is meant). It is possible that one command yields an undefined state, but this is handled by an extra condition.

The particular kind of merge of $r1$ and $r2$ can be conceived as being calculated as follows: if the first element of $r1$ is A or d_1 (with $d_1 \in D$) then it is the first element of the result. It cannot be the case that at the same time the first element of $r2$ is A or d_2 as well, because an offer is always accepted or responded to by at most one process: an offer is always directed to one particular process. Hence, $r2$ is

empty or its first element is R. The rest of the result of the merge is the merge of the rest of $r1$ and the rest of $r2$.

The case that the first element of $r2$ is A or d is completely analogous. If one of the streams is empty, then the other is copied to yield the rest of the result. Otherwise, an offer is rejected only if it is rejected by both commands. In this case the rest of the result is again the merge of the rest of $r1$ and the rest of $r2$.

Example. Suppose $s = [(P_1, P_2), (P_1, P_2, x, d), (P_1, P_5), (P_5, P_3, y, d'), (P_3, P_4)]$ and suppose $f1, f2$ model P_1, P_2, respectively, where
$f1(s) = ([10, R, 20, R, R], \sigma 1)$ and
$f2(s) = ([R, A, R, R], \sigma 2)$ and
the original state is σ, then
$\text{join}(f1, f2, s) = \{([10, A, 20, R, R], \text{upd}(\sigma, \sigma 1, \sigma 2))\}$

At this stage the behavior of the parallel composition is such that an offer, which can be accepted or responded to by one of the processes, is in fact accepted or responded to by the parallel composition. Cooperation between the processes is not achieved yet. This will be done in step two.

(10.2) **step 2.**

If all n commands in the parallel composition are bound by the binding function B, then the second step in the parallel composition is performed. This second step gives the meaning of a CSP-W program. A function M is introduced, where
M :<program>$\rightarrow$ $DENOT$. This function M is the main semantic function, as it assigns a meaning to a CSP-W program. This is achieved by the use of the auxiliary function B, which in its turn uses the a priori semantics of the individual processes given by F. In this way induction on the syntax is performed. The definition of M is as follows: (parcom is a CSP-W program without begin-end brackets)

$M_{\sigma_0}[\ |[\text{parcom}]|\ ](s) =$

$$\begin{aligned}
&\{(r, \sigma) \mid (r, \sigma) \in B_{\sigma_0}[\text{parcom}](s) \wedge \text{handshake}(s, r)\} \cup \\
&\{(r', \bot) \mid \exists p, r, \sigma\ ((r, \sigma) \in B_{\sigma_0}[\text{parcom}](s) \wedge \text{prefix}(p, s) \wedge \\
&\qquad \text{prefix}(r', r) \wedge \text{handshake}(p, r') \wedge \\
&\qquad \neg\text{handshake}(|\text{drop}(p, s)|_2,\ |\text{drop}(r', r)|_2)\}
\end{aligned}$$

where

$\text{handshake}(s, r) = \forall i(\text{hs}(|s|_{2i}, |r|_{2i}))$

$$\begin{aligned}
&\mathrm{hs}(\epsilon, \epsilon) = \text{true} \\
&\mathrm{hs}(o1 : \epsilon, s) = \mathrm{hs}(s, r1 : \epsilon) = \text{false} \\
&\mathrm{hs}((o1 : o2 : s1, r1 : r2 : s2) = \mathrm{hs}(s1, s2) \wedge \exists n1, n2 \in \text{<proname>}, x \in \text{<id>}, \\
&\qquad\qquad d \in D(o1 = (n1, n2) \wedge \\
&\qquad\qquad o2 = (n1, n2, x, d) \wedge r1 = d \wedge r2 = A)
\end{aligned}$$

The predicate handshake enforces that all offers are done in corresponding pairs, of which the input offer must be accepted and of which the output offer must be reacted to with a value equal to the value in the input offer. If this condition does not hold then the result of a program to an experiment s contains the (possibly empty) largest prefix of the reactions, which, together with the corresponding offers, makes the predicate handshake true. In this case the resulting state is undefined. Note that handshake is defined for both finite and infinite streams.

The intuition behind the correspondence between an experiment and a program can now be stated as follows. A communication action is proposed by means of two offers. If the offers correspond and the reactions are appropriate, then the communication instructions are executed simultaneously in the communication partners involved and a next communication action may be proposed. At the moment the two (or fewer) offers and the reactions do not represent a handshake, the state becomes undefined and the experiment terminates abnormally (further input does not affect this behavior). This is in contrast to the a priori meaning of a process, which may reject an offer and yet respond to offers appearing after it in the experiment.

3.4 Remarks and extensions

In the preceding section we defined F, B and M. The meaning of a recursive definition is given by applying the appropriate fixed point theory. Here, the same fixed point technique as in [Broy 84] has been used. Now, the features of CSP which are absent in CSP-W will be discussed.

The model of stream processing functions is capable of dealing with mixed guards, see [Broy 86]. The model of stream processing functions also allows for the treatment of nested concurrency: the binding operator B combines the semantics of the processes P_i and P_j, which are functions in $DENOT$, to form the semantics of the parallel composition of them, again a function in $DENOT$. In other words, the parallel composition of two processes is a new process, replacing the other two. Thus binding may be performed on any level. Moreover, it can be proved that $P_1 \parallel \ldots \parallel P_n$ may be combined in pairs in any association order.

Whether the distributed termination convention can be introduced in the model or not is an open problem.

A serious drawback of stream processing functions seems to be that the semantics of a program is not a function from an initial state to a set of final states, like one would possibly hope. Nevertheless it is possible to abstract from communications (offers) in the following way. Consider the following alternative M' to M,

M' :<program>$\rightarrow STATE \rightarrow P(STATE^{\perp})$

$$M'_{\sigma_0}[\,|[\text{parcom}]|\,] = \{\sigma \in STATE \mid \exists s1 \in OFFER^*, s2 \in REACT^*((s2,\sigma) \in B_{\sigma_0}[\text{parcom}](s1) \wedge \text{hs}(s1,s2))\}$$

$$\cup \begin{cases} \{\perp\} & \text{if deadlock}(B_{\sigma_0}[\text{parcom}]) \vee \text{diverge}(B_{\sigma_0}[\text{parcom}]) \\ \emptyset & \text{otherwise} \end{cases}$$

where deadlock is a predicate on stream processing functions which is true if the possibility exists that after some offers and reactions a point is reached where all pairs of corresponding offers can be rejected, and diverge is a predicate on stream processing functions which is true if infinite chattering (two or more processes co-operate in an infinite sequence of communications) is possible or if after some offers and reactions an infinite (internal) loop is entered. In the latter case no reaction occurs on any offer.

Actually, this is the approach followed in [Broy 86]. In it, the definitions of diverge and deadlock can be found. The reason why this approach wasn't followed in this report is that it seemed more instructive, more consequent and more supporting the intuition to define the semantics of a CSP-W program as a stream processing function. The entire model was explained by the notion of a certain kind of experiment and the behavior of a program to an experiment. If one abstracts from offers and reactions, the principle of observability is abstracted away with it.

For instance, observability implies that a deadlock is observed if every pair of corresponding offers typed in by the experimenter is rejected. Hence, a deadlock can be concluded by purely observing reactions to some experiments. Therefore, no special predicate 'deadlock' is necessary. Similar statements can be made for diverge.

Moreover, in the approach in this report the a priori semantics of a process is in the same function space as the semantics of an entire program. Both processes and programs can be observed by an experimenter. However, sometimes it is desirable to abstract from communications and this is why the method to do so was enunciated above.

Finally, we make some remarks about the formal definition of CSP-B in [Broy 86]. From Broy's definition it follows that it is perfectly legal to have the fragment $\ldots x!E; x? \ldots$ in one process. If the corresponding offers are supplied to this process then the effect of this is the assignment of E to x. Also at binding, this situation is not noticed and therefore not forbidden. Hence, also if an entire program is supplied with the usual two offers for proposing a communication involving x, then the effect is still an assignment of E to x within the process containing the fragment. This presents a serious problem in the framework proposed by Broy. Note that with the approach in this report, a similar fragment in CSP-W is not possible, i.e. it is forbidden by the formal definition.

It also follows from the formal definition of CSP-B that in the selection statement of CSP-B no abortion occurs if the boolean parts of all guards are false. This can

easily be repaired by extending the definition of the selection.

A last flaw in the definition of CSP-B concerns the definition of join in [Broy 86]. If this version of join is presented with the example of the preceding section then in the second recursive call of join, $f1_x$ becomes undefined: see its definition. In the join as defined in this report this is not the case.

Chapter 4

A linear history semantics for CSP-W

In this chapter a linear history semantics is given for CSP-W. This semantics uses both linear sequences of communications to record the communications occurring in a computation and special states, called expectation sets, characterizing potential deadlocks. For any well-formed program fragment the semantics is a mapping from an initial state to a set of pairs, consisting of a (final) state and a communication sequence to attain this state. These ideas of Francez, Lehmann and Pnueli [FLP] are applied to CSP-W.

4.1 Definitions and domains

First we begin with several definitions. Let D be the data set. A communication record has the form (P_i, P_j, d), meaning that the value $d \in D$ has been passed from P_i to P_j. Σ_i is the set of communications of process P_i defined by $\Sigma_i = \{(P_i, P_j, d) \mid d \in D, j \neq i\} \cup \{(P_j, P_i, d) \mid d \in D, j \neq i\} \cup \{\delta\}$. The first set contains outputs from P_i to some P_j, the second inputs from some P_j to P_i and the third an empty communication δ, which is associated with each noncommunicating instruction such as skip, a test or an assignment. Σ is defined by $\Sigma = \bigcup_i \Sigma_i$.

The role of δ, the empty communication, is to indicate the progress of a noncommunicating computation. It leads to the property that long computations correspond to long communication sequences, regardless of the amount of communication actually generated. The δ resembles the silent transition τ in [Milner].

A communication history is a finite sequence in $H_i = \Sigma_i^*$. Each computation of P_i reaches a state σ and produces a certain communication sequence h in order to get there. Whether this computation is realizable in a given environment of other processes depends on the ability and readiness of the other processes to cooperate in producing the corresponding communications on their side.

The space of states $\overline{S}_i$ is defined as $\overline{S}_i = S_i \cup \{\perp\} \cup E_i$. S_i is the space of possible proper local states of P_i. Here, for simplicity, $S_i = D^{n_i}$, where n_i is the number of local variables of D_i. Possible type restrictions are thereby ignored. If a

computation converges, then it results in a final state $\sigma \in S_i$. The undefined state $\perp$ denotes an incomplete computation, i.e. , a computation that has not converged yet. E_i is a set of expectation states of the form $e(A)$, where A contains elements of the form $p_j^i!$, which denote that process P_i is ready to execute output commands $P_j!E$, and elements of the form $p_j^i?$, which denote that process P_i is ready to execute input commands $P_j?x$. If a process is in a state $e(A)$ then this means that there is a potential deadlock. Only if an action in A happens then the deadlock is resolved. In other words, if none of them ever happens, i.e. , no matching communication is forthcoming, as may be the case if all potential partners have terminated or are each expecting some other unrelated communication, then the process is deadlocked. If a deadlock is established, then $e(A)$ denotes a necessary deadlock. In particular, $e(\emptyset)$ is to be interpreted as an established deadlock or abortion.

Next, the function Dual is defined (the result of Dual is the matching communication):

$\text{Dual}(p_j^i!) = p_i^j?$
$\text{Dual}(p_j^i?) = p_i^j!$

Furthermore, define Π^i and Π_i as follows:

$\Pi^i = \{p_l^i! \mid l = 1, \ldots, n \wedge l \neq i\} \cup \{p_l^i? \mid l = 1, \ldots, n \wedge l \neq i\}$
$\Pi_i = \text{Dual}(\Pi^i) = \{p_i^l! \mid l = 1, \ldots, n \wedge l \neq i\} \cup \{p_i^l? \mid l = 1, \ldots, n \wedge l \neq i\}.$

Now it is possible to define E_i formally:

$E_i = \{e(A) \mid A \subseteq \Pi^i\}$

Whether a deadlock actually occurs can only be determined at binding, because the necessary information about the other processes will then be available. When P_i is analyzed in isolation, the only information that can be stored is that a situation $e(A)$ is possible, in which at least one of the communications in A is necessary in order to resolve the deadlock. Now the most important definition of this section follows:

$DENOT \stackrel{\text{def}}{=} P(\overline{S}_i \times H_i) \rightarrow P(\overline{S}_i \times H_i)$
and
F :<process>$\rightarrow DENOT$

The intended meaning of this definition is as follows. F maps a process to a function from the powerset of a set of pairs to the powerset of the same set of pairs. This function, which is the meaning of a process, maps an initial state σ_0 and an initial history h_0 to a final state and history. Intuitively, for $\sigma_0 \in S_i$, $\sigma \in \overline{S}_i$, $h_0 \in H_i$ we have

$(\sigma, h_0 + +h_1) \in F[P_i :: \text{commandi}](\{(\sigma_0, h_0)\})$

if the following holds. Starting with an initial state σ_0 and a given initial history h_0, a computation leads to a state σ while communicating h_1. These additional communications of the computation are appended to h_0. The initial history h_0 can be thought of as being empty if P_i is the entire process, but is not necessarily empty if P_i is preceded by another program fragment which produced h_0, e.g. in case of sequential composition. For a state σ, $\sigma \in S_i$ holds if the computation reached a termination point. If the computation did not reach a termination point (in case of an incomplete or partial computation), then $\sigma = \perp$. Furthermore, if the computation led to a situation in which P_i is expecting at least one of the communications $c_1, \ldots, c_k$ then $\sigma = e(\{c_1, \ldots, c_k\})$. In this case $(\perp, h_0 + +h_1) \in F[P_i :: \text{commandi}](\sigma_0, h_0)$ for some h_1. Note that the computation is independent of h_0, the "past". For $\sigma_0 \in \overline{S}_i - S_i$, i.e. , $\sigma_0 \in \{\perp\} \cup E_i$, the following always holds:

$$F[P_i :: \text{commandi}](\{(\sigma_0, h_0)\}) = \{(\sigma_0, h_0)\},$$

independently of P_i. For $V \subseteq \overline{S}_i \times H_i$:

$$F[P_i :: \text{commandi}](V) = \bigcup_{(\sigma,h)\in V} F[P_i :: \text{commandi}](\{(\sigma, h)\}).$$

In words, a function in *DENOT* acts on non-singleton sets by taking the union over all elements of the argument. Hence, $F[P_i :: \text{commandi}]$ is totally distributive.

In addition to all possible convergent computations, the semantics for P_i includes all initial subcomputations of convergent computations. Many of these computations will appear to be unrealizable, because the environment cannot respond with the right communications. Because in the result sets elements of the form $(\perp, h)$ are present (the initial subcomputations), the only way to decide whether there exists a divergent computation is to observe chains of h's of unbounded length.

$P(\overline{S}_i \times H_i)$ is partially ordered by subset inclusion $\subseteq$. The minimal element is $\emptyset$ and not $\perp$, which will be introduced by the semantic equations to denote incomplete computations. This certainly is a complete lattice.

4.2 The semantic equations

As a notational convention we write $F[t](\sigma, h)$ for $F[\ P_i :: t\ ](\{(\sigma, h)\})$. Since for every $\sigma \in \overline{S}_i - S_i$ $F[t](\sigma, h) = \{(\sigma, h)\}$, $F[t]$ applied to $\{(\sigma, h)\}$ will only be defined for $\sigma \in S_i$, $h \in H_i$. $F[t]$ for arbitrary subsets of $\overline{S}_i \times H_i$ is defined by distributivity.

(1) skip

$$F[\text{skip}](\sigma, h) = \{(\perp, h), (\sigma, h + +[\delta])\}$$

The first element in the result set denotes the partial computation up to the point before the instruction. This element will appear in the semantics of all other instructions and commands. The second element in the result set represents the state and history upon completion of the computation by skip: σ is unchanged, and h is extended by δ, thus recording a noncommunicating computation step.

(2) assignment

$$F[x := E](\sigma, h) = \{(\bot, h), (\sigma[E_\sigma/x], h + +[\delta])\}$$

In the second element, $\sigma[E_\sigma/x]$ is defined as in section 2.1, where E_σ denotes the value of expression E when evaluated in state σ. It is assumed that E_σ is well-defined if $\sigma \neq \bot$.

(3) output

$$F[P_j!E](\sigma, h) = \{(\bot, h), (e(\{p_j^i!\}), h), (\sigma, h + +[(P_i, P_j, E_\sigma)])\}$$

Here the set of results includes the possibility $e(\{p_j^i!\})$, which records the possibility of a deadlock at this instruction: a situation in which process P_i (recall that $P_j!E$ occurs in process P_i by convention) is waiting indefinitely for process P_j to accept an output from it. The last element denotes the possibility of a successful communication: (P_i, P_j, E_σ), which means that the value E_σ is communicated from P_i to P_j, is appended to the communication history.

(4) input

$$F[P_j?x](\sigma, h) = \{(\bot, h), (e(\{p_j^i?\}), h)\} \cup \{(\sigma[d/x], h + +[(P_j, P_i, d)]) \mid d \in D\}$$

The second element of the first set is similar to the case as discussed for output. The second set is a set of results corresponding to all input values $d \in D$ which may possibly be communicated from process P_j to process P_i. For each input d, the resulting pair consists of the modified state, in which d is assigned to x, and a new communication history, which is obtained by appending the appropriate communication record to h.

The part of the result set which constitutes the successful communications will be defined by the following. This definition will be convenient in the sequel.

$\mathrm{Suc}[P_j!E](\sigma, h) = \{(\sigma, h + +[(P_i, P_j, E_\sigma)])\}$
$\mathrm{Suc}[P_j?x](\sigma, h) = \{(\sigma[d/x], h + +[(P_j, P_i, d)]]) \mid d \in D\}$,

Suc can be extended to $P(\overline{S}_i \times H_i)$ in the usual way. Suc does not apply to every program fragment. It applies to communication instructions only.

(5) sequential composition

$$F[\text{command1;command2}](\sigma, h) = F[\text{command2}](F[\text{command1}](\sigma, h))$$

Note that any intermediate result (σ', h') with $\sigma' \in \{\perp\} \cup E_i$ and generated in command1 is preserved by F[command2] and is element of the result set of command1;command2.

(6) selection with all guards purely boolean

Every selection statement with all guards purely boolean has the form
$SEL1 = [b_1 \rightarrow \text{command1} \,\square \ldots \square\, b_m \rightarrow \text{commandm}]$. For a given state σ we define L_σ to be the set of all $l \in \{1, \ldots, m\}$ such that $(b_l)_\sigma$ = true. It is assumed that $b_1, \ldots, b_m$ are well defined when $\sigma \neq \perp$.

Then $F[SEL1](\sigma, h) = \{(\perp, h)\}$
$\cup \bigcup_{l \in L_\sigma} F[\text{commandl}](\sigma, h + +[\delta])$
$\cup$ if $L_\sigma = \emptyset$ then $\{(e(\emptyset), h)\}$ else $\emptyset$

The second part of this definition selects all (boolean) guards which are true when evaluated in σ, records this test as a noncommunicating computation step by appending δ to the history, applies the meaning of the corresponding command to the state and updated history and adds the result of this application to the result set of the selection statement. The third part of the definition handles the case of no true guards. In this case $(e(\emptyset), h)$ is generated. Recall that $e(\emptyset)$ is to be interpreted as an established deadlock or an abortion. The latter of these applies, because a selection statement should abort if all guards in it are false.

(7) selection with all guards containing a communication

Every selection with all guards containing a communication has the form

$SEL2 = [b_1; c_1 \rightarrow \text{command1} \,\square \ldots \square\, b_m; c_m \rightarrow \text{commandm}]$. If a b_l is absent, it is taken as true by default.

$$\begin{aligned} F[SEL2](\sigma, h) = \{&(\bot, h)\} \\ &\cup \bigcup_{l \in L_\sigma} F[\text{commandl}](\text{Suc}[c_l](\sigma, h + +[\delta])) \\ &\cup \{(e(\{c_l^i \mid l \in L_\sigma\}), h)\} \end{aligned}$$

where c_l^i is defined by:

$$c_l^i = \begin{cases} p_j^i! & \text{if } c_l \text{ has the form } P_j!E \\ p_j^i? & \text{if } c_l \text{ has the form } P_j?x \end{cases}$$

The second part is similar to the case discussed in (6), except for the effect of the communications. In this part of the definition a successful communication is assumed. Recall that the semantics of P_i includes all possible computations, regardless of which of them will later appear to be unrealizable. The third part denotes the possibility of a deadlock. Because every guard contains a communication, the possibility of a deadlock at this selection statement is present if all processes addressed by communication parts of which the corresponding boolean parts are true, fail to communicate with P_i. Therefore an expecting state containing the communications that have the corresponding boolean parts true, is added to the result. Note that if there is no true boolean then again $(e(\emptyset), h)$ is generated, which should be interpreted as an abortion.

(8) iteration

Let C have the form of either $SEL1$ or $SEL2$. The semantics of iteration is given by the least fixed point of the following equation.

$$\begin{aligned} F[*C](\sigma, h) = \{&(\bot, h)\} \cup \\ &\text{if } \neg(b_1)_\sigma \wedge \ldots \wedge \neg(b_m)_\sigma \\ &\text{then } \{(\sigma, h + +[\delta])\} \\ &\text{else } F[*C](F[C](\sigma, h)) \end{aligned}$$

This definition applies to both types of iteration statement. It states that the computation only halts if all boolean parts of the guards are false. An extra δ for the test is recorded when all guards are false. When some guard is true, the extra δ is recorded within the evaluation of the selection statement.

(9) parallel composition

First we give a number of relevant definitions:

- The set of combinations of proper states of two processes P_i and P_j is defined by $S_i \times S_j = \{(\sigma_i, \sigma_j) \mid \sigma_i \in S_i \wedge \sigma_j \in S_j\}$.
- The entire state space of the combination of two processes P_i and P_j is defined by $\overline{S}_{ij} = (S_i \times S_j) \cup \{\perp\} \cup E_{ij}$, where the combined space E_{ij} of expectation states contains states of the form $e(\ldots p_l^k \ldots)$ where $k = i$ or $k = j$, but excludes elements of the form p_j^i or p_i^j.
- The combined history space is defined by $H_{ij} = \Sigma_{ij}^*$, where $\Sigma_{ij} = (\Sigma_i \Delta \Sigma_j) \cup \{\delta\}$, i.e. a symmetric difference: all communications involving either P_i or P_j, but not communications between P_i and P_j. These will be converted to δ since they are internal to parallel computations of P_i and P_j.

In order to define the semantics of parallel composition a function M :<program>$\rightarrow$ $(P(\overline{S}_{ij} \times H_{ij}) \rightarrow P(\overline{S}_{ij} \times H_{ij}))$ is introduced, which assigns meanings to CSP-W programs. M uses the semantics of the individual processes given by F. In this way, compositionality is achieved. We consider two cases.

1. For $\sigma \in S_i \times S_j$ and $h \in H_{ij}$:

$$M[\ |[P_i :: \text{commandi} \parallel P_j :: \text{commandj}]|\ ](\sigma, h) =$$

$$\{(\sigma_i \times \sigma_j, h + +h') \mid h' \in M_{ij}(h_i, h_j) \wedge (\sigma_i, h_i) \in F[\ |[P_i :: \text{commandi}]|\ ](\pi_i(\sigma), []) \wedge (\sigma_j, h_j) \in F[\ |[P_j :: \text{commandj}]|\ ](\pi_j(\sigma), [])\}$$

where

- $\pi_i(\sigma)$ is the projection of σ on the variables local to process P_i (the set of values assigned to this variables in σ). Thus, $\pi_i(\sigma) \in S_i$, $\pi_j(\sigma) \in S_j$.
- $M_{ij}(h_i, h_j)$ is a particular kind of merge of h_i and h_j defined by:

$$M_{ij}(h_i, h_j) =$$

$$\begin{aligned}
&\text{if } h_i = h_j = [] \text{ then } \{[]\} \text{ else } \emptyset \\
&\cup \text{ if } h_i = [r] + +h_i' \wedge (r = \delta \vee r \in \Sigma - \Sigma_j) \text{ then } r \cdot M_{ij}(h_i', h_j) \text{ else } \emptyset \\
&\cup \text{ if } h_j = [r] + +h_j' \wedge (r = \delta \vee r \in \Sigma - \Sigma_i) \text{ then } r \cdot M_{ij}(h_i, h_j') \text{ else } \emptyset \\
&\cup \text{ if } h_i = [r] + +h_i' \wedge h_j = [r] + +h_j' \text{ then } \delta \cdot M_{ij}(h_i', h_j') \text{ else } \emptyset
\end{aligned}$$

where $x \cdot y$ denotes the set of elements which results from appending x to every element of the set y. $(x \cdot \emptyset = \emptyset)$

- the cross product $\sigma_i \times \sigma_j \in S_{ij}$ of two states $\sigma_i \in \overline{S}_i$, $\sigma_j \in \overline{S}_j$ is defined by:

$\sigma_i \times \sigma_j =$

$$\begin{array}{l}
\text{if } \sigma_i = \bot \vee \sigma_j = \bot \text{ then } \bot \text{ else} \\
\text{if } \sigma_i = e(A) \wedge \sigma_j \in S_j \text{ then } e(A - \Pi_j) \text{ else} \\
\text{if } \sigma_i \in S_i \wedge \sigma_j = e(B) \text{ then } e(B - \Pi_i) \text{ else} \\
\text{if } \sigma_i = e(A) \wedge \sigma_j = e(B) \text{ then} \\
\qquad\qquad \text{if } A \cap \text{Dual}(B) \neq \emptyset \text{ then } \bot \\
\qquad\qquad \text{else } e(A \cup B - \Pi_i - \Pi_j) \\
\text{else if } \sigma_i \in S_i \wedge \sigma_j \in S_j \text{ then } (\sigma_i, \sigma_j)
\end{array}$$

2. For $\sigma \in \overline{S}_{ij} - S_i \times S_j$ the usual rule holds:

$$M[\ |[P_i :: \text{commandi} \parallel P_j :: \text{commandj}]|\ ](\sigma, h) = \{(\sigma, h)\}$$

Before we explain the entire definition of M, we explain the components merge of histories and cross product of states. The merge M_{ij} of the sequences h_i and h_j produced by P_i and P_j, respectively, contains all the communications that P_i and P_j had in h_i and h_j with the external world, but none of the communications between each other. While forming the merge, it is checked that the communications between P_i and P_j match: similar records have to occur in both sequences and are both replaced by one new δ, because inter-communications have become internal to the combination of the processes and are regarded as noncommunicating computation steps.

Note that a merge is successful only if with each (P_i, P_j, d) in h_j there corresponds a (P_i, P_j, d) in h_i and vice versa. If a merge is not successful then it results in an empty set. For instance, $M_{ij}([(P_i, P_j, d)], []) = \emptyset$. In this example one of the processes, say P_i, wishes to communicate with P_j, but P_j does not respond. Finally, all δ's from h_i and h_j occur in the merge.

The cross product $\sigma_i \times \sigma_j$ of two states defines the combined state resulting from P_i reaching state σ_i and P_j reaching σ_j. If either σ_i or σ_j denotes an incomplete computation, then the combined state is undefined. If $\sigma_i = e(A)$, i.e. , P_i is in an expecting state, and $\sigma_j \in S_j$, i.e. , P_j has terminated, then the resulting state is still expecting. From its expectation set all expectations from P_j are deleted, since P_j will never realize them. This may result in an $e(\emptyset)$ state: an established deadlock. The case of $\sigma_j = e(B)$ and $\sigma_i \in S_i$ is symmetric. If $\sigma_i = e(A)$ and $\sigma_j = e(B)$, i.e. , both are expecting, then there are two cases. If $A \cap \text{Dual}(B) \neq \emptyset$ then there exist communications $p_j^i! \in A$ and its dual $p_i^j? \in B$. If this is the case, then both P_i and P_j can resolve the deadlock: both have a ready communication partner in each other. All that we can say is that there exist some incomplete computations which

bring us to these situations in the respective processes. Consequently, the resulting state is $\perp$. Otherwise, P_i and P_j need help in the resolution of the deadlock. For this purpose a unified expecting state is formed consisting of all external expectations of either P_i or P_j: $e(A \cup B - \Pi_i - \Pi_j)$. If both P_i and P_j have converged and reached proper final states $\sigma_i \in S_i$ and $\sigma_j \in S_j$, then the resulting state is the combination (σ_i, σ_j) meaning the combined values of the variable sets corresponding to P_i and P_j, respectively.

The semantics of parallel composition is now defined as follows: the executions of the parallel composition of P_i and P_j are obtained by coupling the matching individual executions of P_i and P_j. Coupling consists of forming the merge of the communication sequences of the computations produced by P_i and P_j and of forming the cross product of the states.

The resulting semantics of the parallel composition can be interpreted as before: for an argument $(\sigma_0, [])$ it contains a set of pairs (σ, h), where σ denotes a final state, an incompleteness state or an expectation state and h a communication history for getting to this state. Note that h may contain either δ steps or communications with the external world of either P_i or P_j .

We are now in a position to consider the semantics of a complete CSP-W program P that has no communication with any process outside the program. The meaning of $P = |[P_1 :: \text{command1}|| \ldots || P_n :: \text{commandn}]|$ can be found by successive application of the binary binding operator. The order in which this happens is irrelevant, because it can be proved that the binary composition is associative and commutative.

Let $\sigma_0 \in S$ be an initial state. The set $M[P](\sigma_0, []) \subseteq S \times \delta^*$ represents all possible computations in P. It can be interpreted as described in the following summary.

The program has a divergent computation iff, for every $n \geq 0$,

$$(\perp, \delta^n) \in M[P](\sigma_0, [])$$

The program has a convergent computation resulting in a state $\sigma \in S$ iff there exists a $n \geq 0$ such that

$$(\sigma, \delta^n) \in M[P](\sigma_0, [])$$

The program can reach a deadlock (no process can proceed but not all processes have terminated) or is aborted iff there exists a $n \geq 0$ such that

$$(e(\emptyset), \delta^n) \in M[P](\sigma_0, [])$$

4.3 Remarks and extensions

One of the advantages of this approach over the three other semantics of CSP-W presented in this report (cf. chapters 3, 5, 6) is the fact that subset ordering is adequate for the semantic analysis and that no power domain construction is necessary.

The proof that the relevant functions (e.g. the one defined by the least fixed point equation for iteration) are continuous is evident since the ordering relation is that of subset inclusion. Furthermore, the proof that the binary binding is associative and commutative is straightforward, but rather tedious because it requires an exhaustive consideration of many cases.

Now consider the features of CSP which are absent in CSP-W. The linear history semantics can model mixed guards, see [FLP]. Nested concurrency can be modeled by the linear history semantics because of the uniformity of the a priori semantics of a process and the semantics of the parallel composition of several processes. A parallel composition could be combined with other processes as though it were a single process. This allows for the treatment of nested concurrency, in which one process contains a parallel composition of several other processes, in a most natural way. Problems arising from naming conventions, made necessary by nested concurrency, are also elegantly solved by linear history semantics, see [FLP].

Francez, Lehmann and Pnueli claim that a "reasonable" extension to the model can be made which will accommodate the distributed termination convention. In other words, the linear history semantics presented for CSP-W must be extended in order to deal with the distributed termination convention.

Chapter 5

An alternative linear history semantics for CSP-W

In this chapter an alternative linear history semantics is given for CSP-W: it is based on the same notions of local states and communication histories as presented in chapter 4, but it differs in the way these ideas are elaborated. The definitions of the domains, the orderings and denotations for several constructs differ considerably and sometimes are completely different. Moreover, the meaning of an entire CSP-W program will be a mapping from the set of combined states of the individual processes to the powerset of the union of these combined states and $\{\perp, d\}$, where $\perp$ denotes an infinite loop, and d the fail state.

Essentially, information that was available in the expectation sets of the previous chapter will now be stored in the communication histories. In this chapter a semantics of CSP-W will be given based on this alternative approach of Soundararajan [Sound].

5.1 Definitions and domains

Let S_i be the set of local states in which the abort state a_i is included. $S_i^{\perp}$ is the usual flat domain. Again, $\perp$ denotes nontermination, which is treated differently from abortion. Σ_i, the set of possible communications of P_i, is defined as follows:

$$\begin{aligned}\Sigma_i \; = & \{(P_i, P_j, d, T) \mid j \neq i \wedge T \subseteq O_j \wedge d \in D\} \cup \\ & \{(P_j, P_i, d, T) \mid j \neq i \wedge T \subseteq O'_j \wedge d \in D\}\end{aligned}$$

where $O_j = \{(P_i, P_{j'}) \mid j' \neq i \wedge j' \neq j\} \cup \{(P_{j'}, P_i) \mid j' \neq i\}$
and $O'_j = \{(P_i, P_{j'}) \mid j' \neq i\} \cup \{(P_{j'}, P_i) \mid j' \neq i \wedge j' \neq j\}$

The two types of records of communication correspond to output and input instructions, respectively. If an output or input instruction does not occur in a guard

of a selection or iteration statement, then $T = \emptyset$ and the records of communication are similar to those of chapter 4.

Now consider the case of an output instruction which occurs within a guard of a selection or iteration statement. Records of communication corresponding to such output instructions have the form (P_i, P_j, d, T) denoting not only that the value d was communicated from P_i to P_j, but also that there were other options available to P_i at this point. T is the set of other options. An element $(P_{j'}, P_i)$ in T indicates that instead of outputting to P_j, P_i could have input from $P_{j'}$. Similarly, an element $(P_i, P_{j'})$ in T indicates that P_i could have output to $P_{j'}$ instead of to P_j. The records of communication corresponding to input instructions within a guard are quite similar to those for output instructions within a guard. This motivates the definition of Σ_i.

H_i is the domain of communication sequences of P_i. This domain is rather complex, because of the complexity of Σ_i: $H_i = \Sigma_i^* \cup \Sigma_i^\infty$, where Σ_i^* is the set of all finite sequences of elements of Σ_i and Σ_i^∞ the set of all infinite sequences of elements of Σ_i. The ordering on H_i is the initial subsequence order: $h_i \sqsubseteq h_i'$ iff h_i is an initial subsequence of h_i'. The empty sequence ϵ (or []) is the least element of H_i.

Now consider the domain $S_i^\perp \times H_i$. Here, the Cartesian product notation is used despite the fact that $S_i^\perp \times H_i$ does not include all elements which have the form (σ_i, h_i). The precise definition of the domain is as follows:

$$S_i^\perp \times H_i = \{(\sigma_i, h_i) \mid \sigma_i \in S_i^\perp \wedge h_i \in \Sigma_i^*\} \cup \{(\perp, h_i) \mid h_i \in \Sigma_i^\infty\}$$

Thus, a proper state in S_i cannot be paired with an infinite communication sequence. The reason for this is that if P_i has an infinitely long communication sequence then its state must be $\perp$, because such a sequence can only be the result of a nonterminating loop. Note that $(\perp, h_i)$, where h_i is a finite sequence, is a perfectly reasonable combination of a state and communication history.

This 'almost' Cartesian product of $S_i^\perp$ and H_i has an ordering different from the usual ordering on Cartesian product domains. The ordering is defined by:

$$(\sigma_i, h_i) \sqsubseteq (\sigma_i', h_i') \text{ iff } (\sigma_i = \sigma_i' \wedge h_i = h_i') \vee (\sigma_i = \perp \wedge h_i \sqsubseteq h_i')$$

Now, all ingredients are available for the following, crucial definition:

$DENOT \stackrel{\text{def}}{=} P_s(S_i^\perp \times H_i) \rightarrow P_s(S_i^\perp \times H_i)$ and
F :<process>$\rightarrow DENOT$,

where P_s denotes the 'almost' powerset (see below).
The order on the domain $P_s(S_i^\perp \times H_i)$ is the Egli-Milner order. For $X, X' \in P_s(S_i^\perp \times H_i)$:

$$X \sqsubseteq X' \text{ iff } (\forall x \in X \exists x' \in X'(x \sqsubseteq x')) \wedge (\forall x' \in X' \exists x \in X(x \sqsubseteq x'))$$

$P_s(S_i^\perp \times H_i)$ denotes the set of elements X of the usual powerset $P(S_i^\perp \times H_i)$ for which the following conditions hold:

(a) X is convex, i.e. $x \sqsubseteq y \sqsubseteq z \wedge x \in X \wedge z \in X \Rightarrow y \in X$

(b) X is complete, i.e. if X is an infinite set and there exists $x_1, x_2, \ldots$ and $x'_1, x'_2, \ldots$ such that for all $j : x_j \sqsubset x_{j+1} \wedge x_j \sqsubseteq x'_j \wedge x'_j \in X$ then lub $\{x_1, x_2, \ldots\} \in X$

The first condition is rather artificial: the only reason for imposing this condition is that it simplifies the theory considerably, e.g. lub's of chains of elements of $P_s(S_i^\perp \times H_i)$ are unique. An unpleasant side effect of this condition is that some individual processes will have slightly unnatural denotations. Fortunately, the denotations of complete CSP-W programs are unaffected by the convexity requirement.

Essentially, the second condition states that if a process can communicate an arbitrary number of times then it also can communicate forever. The reason for imposing this condition is that it ensures the continuity of the functionals needed in the definition of iterations.

Finally, two operators are defined, which will be used in the definition of the selection and iteration statements. The first is the convex closure operator C_1: $P(S_i^\perp \times H_i) \rightarrow P(S_i^\perp \times H_i)$, defined for any $X \subseteq S_i^\perp \times H_i$:

$$C_1[X] = \{y \mid \exists x \in X, z \in X \ (x \sqsubseteq y \sqsubseteq z)\}$$

The completeness closure operator C_2 with the same functionality is defined for any $X \subseteq S_i^\perp \times H_i$:

$$C_2[X] = X \ \cup \{x \mid \exists x_1, x_2, \ldots, x'_1, x'_2, \ldots ((\forall j (x_j \sqsubset x_{j+1} \wedge x_j \sqsubseteq x'_j \wedge x'_j \in X)) \wedge x = \text{lub}\{x_1, x_2, \ldots\}) \}$$

5.2 The semantic equations

All denotations f which are a result of F, will satisfy the following conditions:

(a) $X = \emptyset \Rightarrow f(X) = \emptyset$

(b) $f(C_1[X_1 \cup X_2]) = C_1[f(X_1) \cup f(X_2)]$ for all $X_1, X_2 \in P_s(S_i^\perp \times H_i)$.

(c) $\forall (\sigma, h) \in X \ \exists (\sigma', h') \in f(X)(h \sqsubseteq h')$

(d) $\forall (\sigma', h') \in f(X) \ \exists (\sigma, h) \in X(h \sqsubseteq h')$

where (c) and (d) hold for all $X \in P_s(S_i^\perp \times H_i)$.

As a notational convention we will write $F[t](\sigma, h)$ for $F[P_i :: t](\{(\sigma, h)\})$. Since

$F[t](\bot, h_i) = \{(\bot, h_i)\}$ and $F[t](a_i, h_i) = \{(a_i, h_i)\}$ and $F[t](X) = \bigcup_{(\sigma,h)\in X} F[t](\sigma, h)$ for t being skip, assignment, an input or output instruction, we only need to define $F[t]$ applied to $\{(\sigma, h)\}$ for such t, where $(\sigma, h) \in S_i^{\bot} \times H_i$, $\sigma \neq \bot$ and $\sigma \neq a_i$. For the other constructs t this is not the case.

(1) skip

$$F[\text{skip}](\sigma, h) = \{(\sigma, h)\}$$

(2) assignment

$$F[x := E](\sigma, h) = \{(\sigma[E_\sigma/x], h)\}$$

where E_σ is defined as in section 4.2 .

(3) output

$$F[P_j!E](\sigma, h) = \{(\sigma, h + +[(P_i, P_j, E_\sigma, \emptyset)])\}$$

Recall that the fourth component of a record of communication is the set of other options open to P_i at this point. In this case it is the empty set, since P_i has no other options at this point.

(4) input

$$F[P_j?x](\sigma, h) = \{(\sigma[d/x], h + +[(P_j, P_i, d, \emptyset)]) \mid d \in D\}$$

(5) sequential composition

$$F[\text{command1;command2}](X) = F[\text{command2}](F[\text{command1}](X))$$

(6) selection with all guards purely boolean

Every selection statement with all guards purely boolean has the form $SEL1 = [b_1 \longrightarrow \text{command1} \,\Box \ldots \Box\, b_m \longrightarrow \text{commandm}]$. The semantics of $SEL1$ is given by:

$$F[SEL1](X) = C_1[X^0 \cup f_1(C_2[X^1]) \cup \ldots \cup f_m(C_2[X^m]) \cup C_2[X^{m+1}]]$$

where

- $X^0 = \{(\sigma, h) \mid (\sigma, h) \in X \wedge (\sigma = \perp \ \vee \sigma = a_i)\}$
- $X^j = \{(\sigma, h) \mid (\sigma, h) \in X \wedge \sigma \neq \perp \wedge \sigma \neq a_i \wedge (b_j)_\sigma\}$, $1 \leq j \leq m$
- $X^{m+1} = \{(a_i, h') \mid \exists (\sigma, h) \in X (\sigma \neq \perp \wedge \sigma \neq a_i \wedge h' = h \wedge \neg(b_1)_\sigma \wedge \ldots \wedge \neg(b_m)_\sigma)\}$
- $f_j = F[\text{commandj}]$, $1 \leq j \leq m$.

It is assumed that $b_1, \ldots, b_m$ are well-defined when $\sigma \neq \perp$ and $\sigma \neq a_i$. X^0 contains those pairs from X of which the state is $\perp$ or a_i. X^j contains those pairs from X for which the j'th guard evaluates to true when evaluated in the state component of such a pair. To X^j the denotation of commandj is applied. X^{m+1} contains the pairs from X for which all guards evaluate to false. The state component of such a pair is replaced by a_i, which denotes an abortion.

(7) selection with all guards containing a communication

Every selection statement with all guards containing a communication has the form $SEL2 = [b_1; c_1 \longrightarrow \text{command1} \,\Box \ldots \Box\, b_m; c_m \longrightarrow \text{commandm}]$. If the boolean part of a guard is absent, then it is taken as true by default. The semantics of $SEL2$ is given by:

$$F[SEL2](X) = C_1[X^0 \cup f_1(g_1(C_2[X^1])) \cup \ldots \cup f_m(g_m(C_2[X^m])) \cup C_2[X^{m+1}]]$$

where X^0, X^j $(1 \leq j \leq m)$, X^{m+1}, f_j are defined as in (6).

Note that the definition of X^{m+1} implies that an abortion occurs when the boolean parts of all guards evaluate to false. The only difference with (6) consists of the extra g_j's, which capture the effect of the communications in the guards. The definition of g_j depends on whether the j'th guard has an input or output instruction

as communication part. If the j'th guard has the form $b_j; P_k!E$ then

$$g_j(\sigma, h) = \{(\sigma, h + +[(P_i, P_k, E_\sigma, T)])\}$$

where T is the set of other options (communications in other guards, whose corresponding boolean parts are true):

$$\begin{aligned} T = &\{(P_i, P_{k'}) \mid \exists l \leq m(l \in OG \wedge (b_l)_\sigma \wedge CP(l) = k' \wedge k' \neq k)\} \cup \\ &\{(P_{k'}, P_i) \mid \exists l \leq m(l \in IG \wedge (b_l)_\sigma \wedge CP(l) = k')\} \end{aligned}$$

where OG, IG are the sets of indices of the output and input guards, respectively. $CP(l)$ is the communication partner of P_i in the l'th guard, i.e. $CP(l) = k$ if the l'th guard is $b_l; P_k!E$ or $b_l; P_k?x$. If the j'th guard has the form $b_j; P_k?x$ then

$$g_j(\sigma, h) = \{(\sigma[d/x], h + +[(P_k, P_i, d, T)]) \mid d \in D\}$$

$$\begin{aligned} \text{where } T = &\{(P_i, P_{k'}) \mid \exists l \leq m(l \in OG \wedge (b_l)_\sigma \wedge CP(l) = k')\} \cup \\ &\{(P_{k'}, P_i) \mid \exists l \leq m(l \in IG \wedge (b_l)_\sigma \wedge CP(l) = k' \wedge k' \neq k)\} \end{aligned}$$

(8) iteration

Let C be either $SEL1$ or $SEL2$. The semantics of the iteration statement $*C$ is the least fixed point of the following equation:

$$F[*C](X) = C_1[X^0 \cup F[*C](Y^1) \cup \ldots \cup F[*C](Y^m) \cup Y^{m+1}]$$

where

- X^0 is defined as in (6).
- $Y^j = f_j(g_j(C_2[X^j]))$, $1 \leq j \leq m$ if C is $SEL2$.
- $Y^j = f_j(C_2[X^j])$, $1 \leq j \leq m$ if C is $SEL1$.
- f_j, g_j and X^j are defined as in (6) and (7).
- $\begin{aligned} Y^{m+1} = C_2[\{(\sigma, h) \mid (\sigma, h) \in X \wedge \sigma \neq \perp \wedge \sigma \neq a_i \wedge \\ \neg(b_1)_\sigma \wedge \ldots \wedge \neg(b_m)_\sigma\}] \end{aligned}$

Note that the iteration only halts if the boolean parts of all guards are false. (Recall that $F[*C](\emptyset) = \emptyset$.) This is true for both types of iteration statement.

(9) parallel composition

The function M, defined below, assigns meanings to CSP-W programs. The meaning of an entire CSP-W program is a function from the set of initial states, i.e. , the set of combined states of all processes, to the powerset of the union of these combined states and $\{\perp, d\}$:

M :<program>$\rightarrow (S_1^- \times \ldots \times S_n^- \rightarrow P(S_1^- \times \ldots \times S_n^- \cup \{\perp, d\}))$
where $S_i^- = S_i - \{a_i\}$ (N.B. $\perp \notin S_i$).

For a given initial state $(\sigma_1, \ldots, \sigma_n)$, M yields the set of possible final states of the program. If the program can go into an infinite loop, either because (one or more) individual processes go into an infinite loop or because of infinite chattering, $\perp$ will be one of the elements in the set of final states. If the program can reach a deadlock or if one (or more) of its processes aborts, the fail state d will be one of the elements in the set of final states. M uses the semantics of the individual processes given by F. In this way, induction on the syntax is performed and compositionality is achieved. The meaning of a program is defined by:

$M[\ |[P_1 :: \text{command1}\| \ldots \| P_n :: \text{commandn}]|\ ](\sigma_1, \ldots, \sigma_n) =$
$B(F[P_1 :: \text{command1}](\{(\sigma_1, [])\}), \ldots, F[P_n :: \text{commandn}](\{(\sigma_n, [])\}))$

where B is the n-ary binding operator defined by

$B(X_1, \ldots, X_n) = Y_1 \cup Y_2 \cup Y_3$ (where $X_j \in P_s(S_j^{\perp} \times H_j)$, $1 \leq j \leq n$).

Now the definitions of Y_1, Y_2 and Y_3 follow, together with their intuitions.

$$Y_1 = \{(\sigma_1, \ldots, \sigma_n) \mid \sigma_1 \in S_1^- \wedge \ldots \wedge \sigma_n \in S_n^- \wedge \exists h_1, \ldots, h_n(\forall i(\ (\sigma_i, h_i) \in X_i) \wedge \text{compat}(h_1, \ldots, h_n))\}$$

where $\text{compat}(h_1, \ldots, h_n) = \exists h(h \in C^* \wedge \forall i(h/i = \text{trim}(h_i)))$

where

- $C = \{(P_i, P_j, d) \mid i \neq j \wedge 1 \leq i, j \leq n \wedge d \in D\}$
- h/i is the sequence obtained from h by omitting all elements except those of the form (P_i, P_j, d) and (P_j, P_i, d).
- $\text{trim}(h_i)$ is the sequence obtained by dropping the fourth component of each element of h_i.

Y_1 corresponds to a situation in which each process terminates properly, and the communications as recorded in each of the sequences are compatible with the communications in the other sequences. This is enforced in a subtle way. If the sequence h exists, then it is a merge of $h_1, \ldots, h_n$, such that a record of communication is an element of h iff it is a member of the histories of both communication partners involved (and of course, two records of communication in h_i appear in the same order in h).

$$
\begin{aligned}
Y_2 = \{d \mid\ & \exists \sigma_1, \ldots, \sigma_n, h_1, \ldots, h_n (\forall i \leq n(\ (\sigma_i, h_i) \in X_i) \wedge \\
& \exists i (\sigma_i = a_i \wedge \exists h'_1, \ldots, h'_{i-1}, h'_{i+1}, \ldots, h'_n (\forall j \neq i (h'_j \sqsubseteq h_j) \wedge \\
& \text{compat}(h'_1, \ldots, h'_{i-1}, h_i, h'_{i+1}, \ldots, h'_n))))\} \cup \\
\{d \mid\ & \exists \sigma_1, \ldots, \sigma_n, h_1, \ldots, h_n (\forall i \leq n(\ (\sigma_i, h_i) \in X_i) \wedge \\
& \exists h'_1, \ldots, h'_n (\forall i \leq n (h'_i \sqsubseteq h_i) \wedge \text{compat}(h'_1, \ldots, h'_n) \wedge \\
& \text{incompat}(h''_1, \ldots, h''_n)))\}
\end{aligned}
$$

where

- $h''_i = \text{drop}(h'_i, h_i)$ (where drop is defined as in section 3.2, h''_i is the sequence obtained from h_i by dropping the initial subsequence h'_i from it).
- $$
  \begin{aligned}
  \text{incompat}(h''_1, \ldots, h''_n) = & \exists i \leq n (h''_i \neq [\,]) \wedge \\
  & \forall i, j \leq n(\ (i \neq j \wedge |h''_i| \geq 1 \wedge |h''_j| \geq 1) \rightarrow \\
  & \text{options}(h''_i) \cap \text{options}(h''_j) = \emptyset)
  \end{aligned}
  $$

  where

  - $$
    \text{options}(h''_i) = \begin{cases} \{(P_i, P_j)\} \cup T & \text{if first}(h''_i) = (P_i, P_j, d, T) \\ \{(P_j, P_i)\} \cup T & \text{if first}(h''_i) = (P_j, P_i, d, T) \end{cases}
    $$
  - $|h''_i|$ is the lenght of h''_i.
  - $\text{first}(h''_i)$ is defined as in section 3.2

The first part of Y_2 corresponds to the situation in which one of the processes aborts. The condition for this case is that at the moment the abortion occurred, the communication sequences produced at that moment were compatible. This condition is imposed, because it is possible that one of the processes aborts after an unrealizable computation. Such an abortion is not an abortion of the entire program.

The second part of Y_2 corresponds to the situation in which the program cannot continue, because several processes are attempting mutually incompatible communications. This includes the situation in which a process attempts a communication, but its partner has already terminated and the program cannot make progress elsewhere. In these situations the program reaches a deadlock.

$$
\begin{aligned}
Y_3 = \{\bot \mid\ & \exists \sigma_1, \ldots, \sigma_n, h_1, \ldots, h_n (\forall i \leq n(\ (\sigma_i, h_i) \in X_i) \wedge \\
& \exists j (\sigma_j = \bot \wedge \\
& (h_j \in \Sigma_j^* \rightarrow \exists h'_1, \ldots, h'_{j-1}, h'_{j+1}, \ldots, h'_n (\forall i \neq j (h'_i \sqsubseteq h_i) \wedge \\
& \qquad \text{compat}(h'_1, \ldots, h'_{j-1}, h_j, h'_{j+1}, \ldots, h'_n))) \wedge \\
& h_j \in \Sigma_j^\infty \rightarrow \forall m \exists h'_1, \ldots, h'_{j-1}, h'_{j+1}, \ldots, h'_n (\forall i \neq j (h'_i \sqsubseteq h_i \wedge \\
& \qquad \text{compat}(h'_1, \ldots, h'_{j-1}, |h_j|_m, h'_{j+1}, \ldots, h'_n))) \\
&))\}
\end{aligned}
$$

where $|h_j|_m$ is defined as in section 3.2.

Y_3 corresponds to the case of an infinite loop. The part of Y_3 following $h_j \in \Sigma_j^*$ corresponds to the situation in which process P_j goes into an infinite loop after going through a finite number of communications. The part of Y_3 following $h_j \in \Sigma_j^\infty$ corresponds to the situation in which the infinite loop is due to infinite chattering.

This completes the definition of B (and hence the definition of the meaning of a program).

5.3 Remarks and extensions

Proofs of continuity and monotonicity of the relevant functionals and functions, proofs of properties (a), (b), (c), (d) of section 5.2 and the fixed point theory used here can be found in [Sound]. In his paper, Soundararajan developed the domain theory corresponding to the 'almost' powerset $P_s(S_i^\perp \times H_i)$ from scratch. The result is somewhat like the powerdomains of [Plotkin], except that these powerdomains are much more general and complex; they include "recursive powerdomains". Because such complex domains are not needed here, Soundararajan preferred to develop his own theory instead of using Plotkin's powerdomain constructor.

Now consider the features of CSP which are absent in CSP-W. The model of Soundararajan is perfectly able to deal with mixed guards, see [Sound]. In its present form, the alternative linear history semantics of Soundararajan seems to be unable to deal with nested concurrency. The problem is that the meaning of a parallel composition is essentially different from the meaning of an individual process, in particular other domains are involved.

The alternative linear history semantics is able of modeling the distributed termination convention, see [Sound]. However, in Soundararajan's model of CSP-S, the selection statement is not interpreted as obeying the distributed termination convention: the selection statement does not abort if all communication partners implied by a communication in a guard with true boolean part have terminated (and at the same time there is no true, purely boolean guard), as it should with our definition of the distributed termination convention. The selection of CSP-S only aborts if the boolean parts of all guards are false, which only is correct if the language does not feature the distributed termination convention. Fortunately, it is

rather straightforward to repair this unfelicity in the definition of the selection in CSP-S.

In the model for CSP-W presented in this chapter, the communications between processes are hidden, since they are internal to the entire program. The communications are not internal to the individual processes and therefore it was reasonable to include the communication sequence of a process in the a priori semantics of this process. Note the difference between the internalization in this chapter and the one in chapter 4. In chapter 4, communications were made internal by replacing them by δ. As a result, sequences of empty communications were included in the semantics of a program (the length of these sequences could be interpreted as the number of steps in a computation).

Now, a serious drawback of the alternative linear history semantics is discussed. The elements of $P_s(S_i^{\perp} \times H_i)$ are required to be convex. This results in identical denotations for the following processes:

P_i :: [true $\rightarrow$ skip $\Box$ true $\rightarrow$ $P_j!5; P_j!5$]; $*$[true $\rightarrow$ skip]
P_i :: [true $\rightarrow$ skip $\Box$ true $\rightarrow$ $P_j!5 \Box$ true $\rightarrow$ $P_j!5; P_j!5$]; $*$[true $\rightarrow$ skip]

This is rather undesirable, since it is obvious that these processes are distinct and therefore should have distinct semantics. Soundararajan claims that it is possible to develop a more complex theory without imposing the requirement of convexity. Unfortunately, even such a theory has similar problems in more complex situations; for an example of such a situation, see [Sound]. For such complex situations, Soundararajan believes that major changes have to be made if the processes are to have distinct semantics in such situations.

The result of imposing the requirement of completeness is less severe: it seems to make it impossible to define a "fair" semantics using the present approach. For a discussion of this subject, see [Sound].

Chapter 6

A synchronization tree semantics for CSP-W

In this chapter a denotational semantics for CSP-W is given based on the idea of synchronization trees. In this model the possible computations of a process are recorded in trees: if a process has different possibilities for continuing the computation then these possibilities will be expressed as different branches of a tree. This leads to the construction of a semantic domain of synchronization (or history) trees. The histories in question are histories of communication, i.e. , traces of records of communication. At binding, the a priori meanings of processes will be composed to a combined meaning of the entire program. These ideas of Francez, Hoare, Lehmann and de Roever [FHLdeR] will be applied to CSP-W.

6.1 Definitions and domains

A record of communication (roc) is a triple (P_i, P_j, d), where $d \in D$. The intended meaning of this triple is as usual: a message d passed from P_i to P_j. The set of all possible records of communication with source P_i and destination P_j is defined by $\Sigma_i^j = \{(P_i, P_j, d) \mid d \in D\}$, $1 \leq i, j \leq n$, where $i \neq j$ and $\Sigma_i^i = \emptyset$. The set of indices of processes which are different from P_i is defined by $\Gamma_i = \{1, \ldots, n\} - \{i\}$. The set of all possible rocs with source P_i is defined by $\Sigma_i = \bigcup_{j \in \Gamma_i} \Sigma_i^j$. Similarly, the set of all possible rocs with destination P_j is defined by $\Sigma^j = \bigcup_{i \in \Gamma_j} \Sigma_i^j$. S_i is the set of states of P_i, which includes *fail*: a special state which denotes a failing computation.

Now, the complete partial order (cpo) T_i (the domain of synchronization trees corresponding to P_i) is defined as the least solution (in the category of cpo's) of a domain equation. Readers unfamiliar with the technicalities of this kind of equations could consult [Stoy], [SM].

$$T_i = (S_i \cup (\bigcup_{j \in \Gamma_i} [\Sigma_j^i \to T_i]) \cup (\bigcup_{j \in \Gamma_i} (\Sigma_i^j \times T_i)) \cup (\Gamma_i \times T_i)^+ \cup T_i^+)_\perp \qquad (6.1)$$

where

X^+ is the domain of finite, nonempty sequences over $X - \{\bot\}$, with the following ordering: there is no bottom element, sequences of different length are not comparable and sequences of the same length are ordered coordinatewise by the ordering inherited from X (when defining X^+ formally, partially undefined and infinite objects should be avoided, see [LS]).

$A_\bot$ is obtained from a partially ordered set A by adding a new bottom element to A.

$\cup$ denotes the disjoint union of partially ordered sets (no bottom element is added). This union is associative. An element of $X_1 \cup X_2 \cup X_3$ is either in X_1 or in X_2 or in X_3 and hence corresponds to three cases.

If A is a set and B a cpo then $[A \to B]$ is the cpo of all total functions from A to B with the obvious ordering.

$\times$ denotes Cartesian product.

The intuition corresponding to the rather complex definition of the domain T_i is as follows. By 6.1, a synchronization tree in T_i is either bottom or a member of one of five summands. The solution to equation 6.1 can be thought of as the domain of all finite and infinite trees which have the following nodes:

(i) leaves: these are labeled by elements of $S_i \cup \{\bot\}$ and have no outgoing arcs.

(ii) input nodes: these have a number of outgoing arcs, each labeled by a record of communication from Σ^i. The number of arcs may be infinite, because infinitely many input values may be possible.

(iii) output nodes: these have one outgoing arc, labeled by a roc from Σ_i. This roc corresponds to the output value.

(iv) global nodes: these have a finite, positive number of outgoing arcs, each labeled by an element from Γ_i. A label denotes a target process, because a global node signals willingness to communicate (either to input or to output) in a situation in which a process has a global choice to communicate with a number of target processes. In a global choice, the choice depends on the environment of the process containing the selection or iteration statement as well. The communication partners are inspected with respect to their willingness to communicate. The number of arcs is finite, because in CSP-W the number of target processes involved in a global choice is finite.

(v) local nodes: these have a finite, positive number of unlabeled arcs. Like global nodes correspond to global nondeterminism, local nodes correspond to local nondeterminism. In a local choice, a process can decide on its own which alternative to choose. A process may choose between a number of independent alternatives, each of which will be recorded in one of the subtrees corresponding to the outgoing arcs of the local node. Because in CSP-W the number of alternatives in a local choice is finite, the number of outgoing arcs is finite.

Note that the five different kinds of node described by (i)-(v), correspond to the five summands in 6.1 . Equation 6.1 induces the following ordering on T_i (see [LS], [Stoy]):

tree1 $\sqsubseteq$ tree2 iff tree2 may be obtained from tree1 by replacing some $\perp$-labeled leaf by some tree' $\in T_i$.

Note that the smallest element in this ordering is $\perp$. By i_S, i_I^j, i_O^j, i_G and i_L we denote the injections from the components to their corresponding copies in T_i, cf. [SS]. Thus, $i_S : S_i \to T_i$, $i_I^j : [\Sigma_j^i \to T_i] \to T_i$, etc. In fact, i_I^j denotes the composition of two injections. The first injection maps a function in $[\Sigma_j^i \to T_i]$ onto an element of the disjoint union $\bigcup_j [\Sigma_j^i \to T_i]$. The second injection maps this element onto a synchronization tree. Now, the definition of $DENOT$ follows.

$DENOT \stackrel{\text{def}}{=} S_i \to T_i$, and
F :<process>$\to DENOT$.

Intuitively, the meaning of a process is a mapping from an initial state to a synchronization tree. F is the function which assigns meanings to processes. The meaning of an entire program will be a function in:

$$S_1 \times \ldots \times S_n \to P_{EM}((S_1 \times \ldots \times S_n) \cup \{\perp, fail, deadlock\})$$

Using $\perp$, the occurrence of an infinite computation in any process P_i can be recorded. $\perp$ denotes the 'undefined' n-tuple. In T_i, $\perp$-nodes are used to describe approximations to other elements in T_i and will appear in the approximations to the a priori semantics of loops within P_i. The state *deadlock* records the possibility of a deadlock situation and the state $fail$ records the possibility of an abortion in any process.

$P_{EM}(D \cup \{\perp\})$ denotes the collection C of all nonempty subsets V of $D \cup \{\perp\}$ satisfying: if $V \in C$ and V is infinite then $\perp \in V$. The ordering on P_{EM} is the following: for $V_1, V_2 \in P_{EM}(D \cup \{\perp\})$

$V_1 \sqsubseteq V_2$ iff either $\perp \in V_1$ and $V_1 - \{\perp\} \subseteq V_2$ or $\perp \notin V_1$ and $V_1 = V_2$

Note that $\{\perp\}$ is the bottom element of P_{EM}. With this ordering P_{EM} is a complete partial order, called the Egli-Milner order (see [Milner], [Plotkin]).

6.2 The semantic equations

As a notational convention we will write $F[t](\sigma)$ for $F[P_i :: t](\sigma)$. We now define $F[t]$ for all sequential constructs t of CSP-W.

(1) $F[t](fail) = i_S(fail)$ for all t.

In the sequel we assume that $\sigma \neq fail$.

(2) skip

$F[\text{skip}](\sigma) = i_S(\sigma)$

The meaning of a skip statement, when applied to an input state, is a tree consisting of a leaf, labeled by the input state.

(3) assignment

$$F[x := E](\sigma) = \text{if } V(E,\sigma) = fail \text{ then } i_S(fail) \\ \text{else } i_S(\sigma[V(E,\sigma)/x])$$

$V(E,\sigma)$ is an auxiliary function, which computes the value of an expression E in state σ. $V(E,\sigma)$ is defined to be $fail$ when E is undefined. The meaning of the assignment is a leaf with updated state. Hence, it does not create any new arcs.

(4) output

$F[P_j!E](\sigma) = i_O^j(\ ((P_i, P_j, V[E,\sigma]), i_S(\sigma))\ )$

Here it is assumed that the value of E is well-defined when evaluated in σ. We obtain a tree with one output node, one outgoing arc labeled with the roc and a leaf labeled by the unmodified state, which reflects that output has no side-effect. At binding, the label of the arc will have to match the label of an input arc in the tree corresponding to P_j.

(5) input

$F[P_j?x](\sigma) = i_I^j(\ \lambda r.i_S(\sigma[r \downarrow 3/x])\ )$

The meaning of an input instruction is a tree consisting of an input node with possibly infinite outgoing arcs, each labeled by a possible input roc r. Each arc ends in a leaf, labeled by an updated state. This update corresponds to the value in the roc r of the arc. This records the side-effect of the input instruction. The value component of r is denoted by $r \downarrow 3$.

(6) sequential composition

$F[\text{command1;command2}](\sigma) = R[\ F[\text{command1}](\sigma), F[\text{command2}]\ ]$

where $R : T_i \times [S_i \rightarrow T_i] \rightarrow T_i$ is defined by

$R[tree, f] =$

$\bot$	if $tree = \bot$
$f(\sigma)$	if $tree = i_S(\sigma)$, for certain $\sigma \in S_i$
$i_I^j(\ \lambda r.R[fun(r), f]\ )$	if $tree = i_I^j(fun)$, for certain $fun \in [\Sigma_j^i \to T_i]$
$i_O^j(\ (r, R[subtree, f])\ )$	if $tree = i_O^j((r, subtree))$, for certain $(r, subtree) \in \Sigma_i^j \times T_i$
$i_G([(i_1, R[subtree1, f]), \ldots, (i_k, R[subtreek, f])])$	if $tree = i_G([(i_1, subtree1), \ldots, (i_k, subtreek)])$, for certain $[(i_1, subtree1), \ldots, (i_k, subtreek)] \in (\Gamma_i \times T_i)^+$
$i_L([R[subtree1, f], \ldots, R[subtreek, f]])$	if $tree = i_L([subtree1, \ldots, subtreek])$, for certain $[subtree1, \ldots, subtreek] \in T_i^+$

$R[tree, f]$ is the tree $tree$, in which each leaf labeled by σ is replaced by the tree $f(\sigma)$, where it is assumed that $f(fail) = i_S(fail)$. In the sequential composition all possible computations of command2 can follow a possible, terminating computation of command1. Thus, the meaning of command2, i.e. , F[command2], has to be applied to all leaves of the intermediate tree produced by command1, i.e. , F[command1](σ). This is done by the replacement operator R.

The definition reflects that the operation of command2 depends on the final state of command1, in which it continues. R may be applied with F as an argument, because F[command1]$(fail) = i_S(fail)$ holds by definition (see (1)). Hence, the assumption about the argument f of R is satisfied. If command1 has a nonterminating computation, then F[command1](σ) will have an infinite path which is not affected by R.

(7) selection with all guards purely boolean

Every selection with all guards purely boolean has the form
$SEL1 = [b_1 \ \to$ command1 $\Box \ldots \Box\, b_m \ \to$ commandm].
Let $L_\sigma \stackrel{\text{def}}{=} \{j \mid 1 \leq j \leq m \wedge V(b_j, \sigma) = \text{true}\}$. It is assumed that $b_1, \ldots, b_m$ are well-defined when evaluated in σ. This will also be assumed in (8)-(10). The meaning of $SEL1$ is defined by:

$$F[SEL1](\sigma) = \text{if } L_\sigma = \emptyset \text{ then } i_S(fail) \text{ else } i_L([F[\text{command}i_1](\sigma), \ldots, F[\text{command}i_k](\sigma)])$$

when $L_\sigma = \{i_1, \ldots, i_k\}$ (if $L_\sigma = \emptyset$ then $k = 0$).

If all guards are false, the computation is aborted and we obtain a tree consisting of a leaf labeled by the fail state. Otherwise we obtain a tree with a local node

and k (≥ 1) outgoing, unlabeled arcs. The trees corresponding to the k commands of which the corresponding booleans are true, are attached to these k arcs. In this way, local nondeterminism is recorded.

(8) selection with all guards containing a communication

Every selection with all guards containing a communication has the form $SEL2 = [b_1; c_1 \rightarrow \text{command1} \,\square \ldots \square\, b_m; c_m \rightarrow \text{commandm}]$. Let L_σ be defined as in (7) and let CP be defined as in section 5.2 (7). Let L'_σ be the set of elements j from L_σ, for which c_j has the form $P_k!E$. For elements j of L'_σ, $EXPR(j)$ is defined as the expression E within c_j.

L''_σ is defined as the set of elements j from L_σ, for which c_j has the form $P_k?x$. For elements j from L''_σ, $VAR(j)$ is defined as the variable x within c_j. Furthermore, if a b_j is absent then it is taken as true by default. The meaning of $SEL2$ is defined by:

$F[SEL2](\sigma) =$

if $L_\sigma = \emptyset$ then $i_S(fail)$
else $i_G($ $[(CP(i_1), i_I^{CP(i_1)}($ $\lambda r.F[\text{command}i_1](\sigma[r \downarrow 3/VAR(i_1)])$ $)),$
$\ldots, (CP(i_k), i_I^{CP(i_k)}($ $\lambda r.F[\text{command}i_k](\sigma[r \downarrow 3/VAR(i_k)])$ $)),$
$(CP(j_1), i_O^{CP(j_1)}($ $((P_i, P_{CP(j_1)}, V[EXPR(j_1), \sigma]),$ $F[\text{command}j_1](\sigma))$ $)),$
$\ldots, (CP(j_l), i_O^{CP(j_l)}($ $((P_i, P_{CP(j_l)}, V[EXPR(j_l), \sigma]),$ $F[\text{command}j_l](\sigma))$ $))]$ $)$

when $L'_\sigma = \{j_1, \ldots, j_l\}$ and $L''_\sigma = \{i_1, \ldots, i_k\}$.

Note that in the else-part k and l may not both be zero. If the boolean parts of all guards are false then the computation is aborted and we again obtain a tree consisting of a leaf labeled by the fail state. Otherwise we obtain a tree consisting of a global node with $k+l$ outgoing arcs. The trees corresponding to the k commands, of which the corresponding booleans are true and the corresponding communications are input instructions, are attached to the k arcs. Each of these k trees corresponds to a particular command and has an input node as a root. The subtrees below this input node correspond to the possible input values of the input instruction: every such subtree corresponds to the meaning of the (fixed) particular command applied to the original state, in which the variable which is involved in the communication is updated by a possible input value. The outgoing arcs of the input node are labeled as in (5).

The trees corresponding to the l commands of which the corresponding booleans are true and the corresponding communications are output instructions, are attached to the l arcs. All l trees consist of an output node with one outgoing arc labeled by the roc corresponding to the output instruction, to which the subtree, which results from the application of the meaning of the particular command to the unmodified state, is attached. All $k + l$ arcs are labeled by the indices of the target processes,

implied by communications of which the corresponding booleans are true. In this way, global nondeterminism is recorded.

(9) iteration with all guards purely boolean

Every iteration with all guards purely boolean has the form $*SEL1$. The meaning of this iteration is given by the least fixed point of the following equation:

$$\begin{aligned} F[*SEL1](\sigma) \;=\; & \text{if } L_\sigma = \emptyset \text{ then } i_S(\sigma) \\ & \text{else } i_L([\; R[F[\text{command}i_1](\sigma), F[*SEL1]], \\ & \quad \ldots, R[F[\text{command}i_k](\sigma), F[*SEL1]] \;]) \end{aligned}$$

when $L_\sigma = \{i_1, \ldots, i_k\}$.

When all guards are false, the loop is exited. This is represented by a leaf in the synchronization tree. (This leaf is labeled by a proper state.) Otherwise, a local node is created and the alternatives of which the corresponding booleans are true, are recorded. Every such alternative is followed by an execution of the whole iteration. Recall that sequential composition is done by the replacement operator R.

(10) iteration with all guards containing a communication

Every iteration with all guards containing a communication has the form $*SEL2$. Again, the meaning is given by the least fixed point of the following equation:

$$F[*SEL2](\sigma) \;=$$

$$\begin{aligned} & \text{if } L_\sigma = \emptyset \text{ then } i_S(\sigma) \\ & \text{else } i_G([(CP(i_1), i_I^{CP(i_1)}(\; \lambda r.R[F[\text{command}i_1](\sigma[r \downarrow 3/VAR(i_1)]), F[*SEL2]] \;)), \\ & \qquad \ldots, (CP(i_k), i_I^{CP(i_k)}(\; \lambda r.R[F[\text{command}i_k](\sigma[r \downarrow 3/VAR(i_k)]), F[*SEL2]] \;)), \\ & \qquad (CP(j_1), i_O^{CP(j_1)}(\; ((P_i, P_{CP(j_1)}, V[EXPR(j_1), \sigma]), \\ & \qquad\qquad R[F[\text{command}j_1](\sigma), F[*SEL2]]) \;)), \\ & \qquad \ldots, (CP(j_l), i_O^{CP(j_l)}(\; ((P_i, P_{CP(j_l)}, V[EXPR(j_l), \sigma]), \\ & \qquad\qquad R[F[\text{command}j_l](\sigma), F[*SEL2]]) \;))]) \end{aligned}$$

when $L'_\sigma = \{j_1, \ldots, j_l\}$ and $L''_\sigma = \{i_1, \ldots, i_k\}$.

When the boolean parts of all guards are false, the loop is exited. Otherwise, a global node is created and again the alternatives of which the corresponding booleans are true are recorded together with the effect of the communication on the state. Every alternative is followed by an execution of the entire iteration. The handling of input

and output guards is similar to (8).

(11) parallel composition

The meaning of an entire CSP-W program will be given by a function
M :<program>$\rightarrow (S_1 \times \ldots \times S_n \rightarrow P_{EM}((S_1 \times \ldots \times S_n) \cup \{\perp, fail, deadlock\}))$.
M combines the a priori semantics of the individual processes given by F. For this purpose, M uses an auxiliary binding function B. In this way, compositionality is achieved. The computation of a program starts in state $(\sigma_1, \ldots, \sigma_n)$ and may produce a set of n-tuples as final states. In this way, nondeterminism is represented. The elements of $\{\perp, fail, deadlock\}$ were explained in section 6.1. The meaning of a program is defined by:

$M[\,|[P_1 :: \text{command1} \parallel \ldots \parallel P_n :: \text{commandn}]|\,]((\sigma_1, \ldots, \sigma_n)) =$
$B(F[P_1 :: \text{command1}](\sigma_1), \ldots, F[P_n :: \text{commandn}](\sigma_n))$

where $B : T_1 \times \ldots \times T_n \rightarrow P_{EM}((S_1 \times \ldots \times S_n) \cup \{\perp, fail, deadlock\})$ is defined by:

$B(tree1, \ldots, treen) =$

1. if $\exists i(1 \leq i \leq n \wedge treei = \perp)$ then $\{\perp\}$ else $\emptyset$ $\cup$

2. if $\exists i(1 \leq i \leq n \wedge treei = i_S(fail))$ then $\{fail\}$ else $\emptyset$ $\cup$

3. $\{(\sigma_1, \ldots, \sigma_n) \mid \forall i(1 \leq i \leq n \rightarrow treei = i_S(\sigma_i))\}$ $\cup$

4. if $treei = fun \in i_I^k([\Sigma_k^i \rightarrow T_i])$ and
   $treek = ((P_k, P_i, d), subtree) \in i_O^i(\Sigma_k^i \times T_k)$
   then $B(tree1, \ldots, treei - 1, fun((P_k, P_i, d)), \ldots, treek - 1, subtree, \ldots, treen)$
   else $\emptyset$ $\cup$

5. if $treei = [treei^1, \ldots, treei^m] \in i_L(T_i^+)$
   then $\bigcup_{1 \leq j \leq m} B(treei, \ldots, treei - 1, treei^j, \ldots, treen)$
   else $\emptyset$ $\cup$

6. if $treei = [(k_1, treei^1), \ldots, (k_m, treei^m)]$
   then $\bigcup_{j \in L_1} B(tree1, \ldots, treei - 1, treei^j, \ldots, treej - 1, treej, \ldots, treen)$
   else $\emptyset$ $\cup$

7. if $treei = [(k_1, treei^1), \ldots, (k_m, treei^m)]$ and
   $treej = [(l_1, treej^1), \ldots, (l_r, treej^r)]$
   then $\bigcup_{(p,q) \in L_2} B(tree1, \ldots, treei - 1, treei^p, \ldots, treej - 1, treej^q, \ldots, treen)$
   else $\emptyset$ $\cup$

8. if "none of the other clauses are applicable"
   then $\{deadlock\}$
   else $\emptyset$

where

- for all but the first clause it is assumed that $\forall i(1 \leq i \leq n \wedge treei \neq \perp)$ and for all but the first and the second clause it is assumed that $\forall i(1 \leq i \leq n \wedge treei \neq i_S(fail))$.

- $L_1 \stackrel{\text{def}}{=} \{j \mid 1 \leq j \leq m \wedge$
  $((treei^j = fun \in i_I^{k_j}([\Sigma_{k_j}^i \rightarrow T_i]) \wedge$
  $treek_j = ((P_{k_j}, P_i, d), subtree) \in i_O^i(\Sigma_{k_j}^i \times T_{k_j})\ ) \vee$
  $(treei^j = ((P_i, P_{k_j}, d), subtree') \in i_O^{k_j}(\Sigma_i^{k_j} \times T_i) \wedge$
  $treek_j = fun' \in i_I^i([\Sigma_i^{k_j} \rightarrow T_{k_j}]))) \}$

- $L_2 \stackrel{\text{def}}{=} \{(p,q) \mid k_p = j \wedge l_q = i \wedge$
  $((treei^p = fun \in i_I^j([\Sigma_j^i \rightarrow T_i]) \wedge$
  $treej^q = ((P_j, P_i, d), subtree) \in i_O^i(\Sigma_j^i \times T_j)\ ) \vee$
  $(treei^p = ((P_i, P_j, d), subtree') \in i_O^j(\Sigma_i^j \times T_i) \wedge$
  $treej^q = fun' \in i_I^i([\Sigma_i^j \rightarrow T_j]))\ )\}$

Finally, the intuitive explanation of each clause in the definition of the binding operator B follows:

1. This clause is needed for the continuity of B.

2. Once an abortion occurs in any process, it will be reflected in the value of B. Recall from (1) that fail is preserved under the a priori semantics.

3. This clause corresponds to the case of successful termination of all processes, each process reaches a final state $\sigma_i \in S_i$.

4. Here, the case of successful communication is handled: *treei* contains an input node (corresponding to input from P_k) and *treek* contains an output node (corresponding to output to P_i). Then, *treei* is replaced by the subtree indicated by the input roc. This is done by applying *treei*, which is a function, to the roc in the label of the outgoing arc of the output node of *treek*. Furthermore, *treek* is replaced by its unique subtree. Finally, B is called recursively.

5. In the case of local nondeterminism, any of the subtrees of *treei* can be chosen and bound to the other *treej*'s. Thus, an arbitrary j, with $1 \leq j \leq m$ is chosen and *treei* is replaced by its subtree $treei^j$. Then, B is called recursively. Since the union is taken over all possibilities, each $treei^j$ will be considered.

6. This is a case of resolvable global nondeterminism: *treei* is a global node, reflecting that P_i is willing to communicate with $P_{k_1}, \ldots, P_{k_m}$. B is called recursively with all subtrees $treei^j$ of *treei* for which the following condition holds: $treei^j$ is an input node (corresponding to an input guard), addressing P_{k_j} and $treek_j$ is an output node addressing P_i, or $treei^j$ is an output node (corresponding to an output guard) addressing P_{k_j} and $treek_j$ is an input node addressing P_i.

   Note that this binding step does not reflect the establishment of the corresponding communication. This communication will be detected at the next level of recursion, when $treei^j$ is confronted with $treek_j$.

7. This is another case of resolvable global nondeterminism, this time in both P_i and P_j. B is called recursively for each pair of corresponding i/o guards (i.e. a guard in P_i addressing P_j and vice versa). Again, the actual communication will be detected at the next level of recursion.

8. This case arises when a group of processes is involved in some cyclic communication attempt, while all other processes have terminated. However, the situation in which a process attempts a communication with a terminated process is interpreted as a deadlock as well (if the program cannot make progress elsewhere). This is the case when *treei* is an input or output node and its communication partner, say *treej*, is a final state in S_j.

   Likewise, the situation of unresolvable global nondeterminism is interpreted as a deadlock. This is the case when *treei* is a global node and all processes $P_{k_1}, \ldots, P_{k_m}$ adressed by *treei* have terminated. These situations are recorded as a deadlock state in the value of B. Note that we are able to detect a nondeterministically possible deadlock state.

6.3 Remarks and extensions

The proofs of monotonicity and continuity of the relevant functionals and functions with respect to the corresponding ordering (both F and B are defined recursively) are either evident or can be found in [FHLdeR].

Now we discuss the features of CSP, which are absent in CSP-W. The model of synchronization trees as presented here is not able to model mixed guards. The reason for this is that a distinction is made between local and global nondeterminism which are reflected by local and global nodes, respectively. Mixed guards do not fit in this framework.

Likewise, nested concurrency is a problem in this model. In its present form, the domains involved in the binding operator B and the domains involved in the a priori semantics of a process are essentially different. If one would introduce a binary binding operator, which constructs an intermediate tree from the trees of two processes (such a tree could be considered as corresponding to a new process: the combination of the other two), then such a tree may correspond to a process with mixed guards in its guarded commands. For an example of this, see [FHLdeR]. But mixed guards cannot be modeled by the synchronization trees of this chapter. Hence, nested concurrency cannot be modeled in this framework.

The distributed termination convention can be modeled by the synchronization tree semantics, see [FHLdeR].

Note that after the processes of a program are bound, all the histories of communication are forgotten and only final states are left.

Finally, the following remark: in this report, the approach of Francez, Hoare, Lehmann and de Roever concerning input- and output nodes was not followed as they described it. In particular, we use additional injection functions in order to emphasize that the spaces of input- and output nodes are disjoint unions as well. Francez et. al. omitted one of the two composed injections. Consider their denotation for the input instruction $P_j?x$: it maps the input state σ to the synchronization tree $i_I(\lambda r.i_S(\sigma[r \downarrow 3/x]))$. Furthermore, their injection has functionality $(\bigcup_j[\Sigma_j^i \to T_i]) \to T_i$. Francez et. al. omitted the injection from $[\Sigma_j^i \to T_i]$ to $\bigcup_j[\Sigma_j^i \to T_i]$. However, without this injection the communication partner P_j is *not* visibly recorded! At binding the following test is done in [FHLdeR]: "if $treei \in i_I([\Sigma_j^i \to T_i])\ldots$". This test is only possible at the moment of binding, if the information of the communication partner is available by the extra injection of $\lambda r.i_S(\sigma[r \downarrow 3/x])$ into the disjoint union. In our approach, input instructions with different communication partners are handled differently. In [FHLdeR] this is not visible.

The same argument does not apply to output nodes because the test "if $treei \in i_O(\Sigma_i^j \times T_i)\ldots$" is always possible, since in the second component of the roc in the first component of the pair the communication partner was recorded.

Similar remarks can be made about input nodes in (8) and (10). Although the communication partners are recorded in the global nodes, at binding the actual communication takes place a level of recursion deeper than the resolution of the global nondeterminism and at that later moment the information about the communication partner must be available in another way. Therefore, also in this case the extra injection is crucial.

Chapter 7

Conclusions

In this report, four denotational semantics have been given for CSP-W, a language based on Communicating Sequential Processes. The various semantics are patterned after the models in [Broy 86], [FLP], [Sound] and [FHLdeR], but required several modifications in order to be applicable to the unified formalism of CSP-W. For example, the model of Broy was applied to CSP-W and CSP-W resembles CSP much more than the version of CSP used by Broy. Furthermore, various flaws in the formal definitions of the four original variants of CSP were discovered. Finally, all definitions used here are completely general: no definitions by example cases of language constructs were used.

Because all four models are applied to one language, it is possible to compare these models from a unified perspective. The topics of the comparison are:

1. Simplicity: of domains, orderings and denotations.
2. Abstractness: to which degree do 'equivalent' program fragments have identical denotations, on the condition that 'inequivalent' program fragments have different denotations?
3. Generality: is it possible in the particular formalism to easily define language constructs of CSP which are not present in CSP-W?
4. Compositionality: is the definition of the language obtained by induction on the syntax?
5. Continuity: are all functions expressing the computation of programs continuous? (This is necessary in order to apply the theory developed for proving properties of programs.)

The four models are examined with respect to these five points:

1. The domains, ordering and denotations in the approach of [FLP] are the simplest; sets of states and communication histories are used. Subset inclusion is used as the ordering relation and the denotations are relatively simple as well. In fact no ordering is needed at all, see [FLP].

The domain (the 'almost' powerdomain) and the ordering (the power domain ordering) are more complicated in the approach of [Sound]. This approach supports the intuition equally well as or even better than the approach of [FLP]: no empty communications are used in the denotations of processes and neither in those of entire programs. Although the information of the length of execution of computations is thereby lost, this is an improvement since length of execution of computations is a pure operational notion.

In the approach of [Broy 86] the ordering (power domain ordering) is complicated and the intuitions corresponding to Broy's model are not as simple as the idea of states and communication histories.

The domain in the approach of [FHLdeR] consists of complex trees with possibly infinite branching degree and possibly infinite depth. Moreover, there are two orderings (of which one is the power domain ordering), because of the recursive definitions of F and B, which operate on different domains.

Furthermore, the denotations in the approaches of [Broy 86] and [FHLdeR] sometimes are rather complicated.

2. This criterion is examined by means of an example. Consider the following processes:

   $P_1 :: x := 1;\ [x = 1 \longrightarrow \text{skip} \ \square\ \text{true} \longrightarrow \text{skip}]$
   $P_1 :: x := 1$
   $P_1 :: x := 1;\ \text{skip};\ \text{skip};\ \text{skip}$

   All three processes have the same semantics in the approaches of [Sound] and [Broy 86]. However, as mentioned in chapter 5, in the approach of [Sound] there even are distinct processes, which have identical semantics. In the approach of [FHLdeR] the semantics of the first process is a rather complex tree and is different from the semantics of the other two. The latter two have identical semantics. In [Broy 84] it is argued that the synchronization tree semantics is not very abstract: it just abstracts from an operational semantics in the sense that it abstracts from sequential notation and from the particular way in which processes are defined (such as bound identifiers). It is shown in the above example that this is an oversimplification. Finally, all three processes have distinct semantics in the approach of [FLP].

3. Let's consider the three major language features of CSP, which are absent in CSP-W:

   The distributed termination convention: the formalism of Francez, Hoare, Lehmann and de Roever quite easily handles the distributed termination convention, but several extensions have to be made to Soundararajan's formalism in order to make it work (here, the formalisms are meant as presented in this report). The distributed termination convention is not

handled in [FLP], but it is claimed that a "reasonable" extension will suffice. Whether the distributed termination convention can be modeled by the model of Broy is an open problem.

Nested concurrency: This feature comes for free in the approaches of [Broy 86] and [FLP]. In the approaches of [Sound] and [FHLdeR], nested concurrency is impossible without major changes.

Mixed guards: The approaches of [Broy 86], [FLP] and [Sound] easily handle mixed guards, but it is impossible to fit them in the framework presented in [FHLdeR].

4. All four approaches define the language by induction on the syntax and thus satisfy the criterion of compositionality.

5. In all four approaches, the resulting functions expressing the computations of programs are continuous. However, it should be mentioned that this had to be enforced rather artificially in the approach of [Sound]; with the consequences as discussed earlier.

Now, some open ends are summarized:

- While the approach of [Broy 86] is superior in the important matter of abstractness (in the approach of [Sound], the disadvantage of identical semantics for distinct processes overshadows the abstractness of the approach), it remains to be seen whether the distributed termination convention can be handled within this framework.

- It also remains to be seen whether the major changes, needed to ensure distinct semantics for different processes in the approach of [Sound], yield a new approach, in the sense that it does not coincide with for instance the approach of [FLP].

Finally, a conclusion must be that if a "reasonable" extension suffices for the incorporation of the distributed termination convention in the approach of [Broy 86], then the approaches of [Broy 86] and [FLP] are equally general and the abstractness of the approach of [Broy 86] balances the simplicity of the approach of [FLP].

Acknowledgements. I would like to thank Jan van Leeuwen for his continuous support during the preparation of this paper. He suggested numerous improvements in the presentation as well. Thanks are due to K.R. Apt for a helpful discussion in an early stage.

Bibliography

[ABC] K.R. Apt, L. Bougé and Ph. Clermont. *Two normal form theorems for CSP programs.* Information Processing Letters, 26, pp. 165-171, 1987.

[ANSI] American National Standards Institute, Inc. *The programming language Ada reference manual.* Lecture Notes in Computer Science 155, Springer Verlag, Berlin, 1983.

[BHR] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe. *A theory of communicating sequential processes.* Journal of the ACM, Vol. 31, No. 3, pp. 560-599, 1984.

[BK] J.A. Bergstra and J.W. Klop. *Process algebra: specification and verification in bisimulation semantics.* In: Mathematics and Computer Science II, (M. Hazewinkel, J.K. Lenstra and L.G.L.T. Meertens, eds.), CWI Monograph 4, North Holland, Amsterdam, pp. 61-94, 1986.

[Broy 84] M. Broy. *Semantics of communicating processes.* Information and Control, 61, pp. 202-246, 1984.

[Broy 86] M. Broy. *Denotational semantics of communicating sequential programs.* Information Processing Letters, 23, pp. 253-259, 1986.

[Dijkstra] E.W. Dijkstra. *A discipline of programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[EF] T. Elrad and N. Francez. *A weakest precondition semantics for communicating processes.* Lecture Notes in Computer Science, 137, pp. 78-90, 1982.

[Egli] H. Egli. *A mathematical model for nondeterministic computations.* Technological University, Zurich, 1975.

[FHLdeR] N. Francez, C.A.R. Hoare, D.J. Lehmann, W.P. de Roever. *Semantics of nondeterminism, concurrency and communication.* Journal of Computer and System Sciences, 19, pp. 290-308, 1979.

[FLP] N. Francez, D.J. Lehmann, A. Pnueli. *A linear history semantics for languages for distributed programming.* Theoretical Computer Science, 32, pp. 23-46, 1984.

[Hoare 78] C.A.R. Hoare. *Communicating sequential processes.* Communications of the ACM, Vol. 21, No. 8, pp. 666-677, 1978.

[Hoare 85] C.A.R. Hoare. *Communicating sequential processes.* Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[LS] D.J. Lehmann and M.B. Smyth. *Algebraic specifications of data types: A synthetic approach.* Mathematical Systems Theory. (Summary in "Proceedings, 18th Annual Symposium on F.O.C.S. Providence, R.I., pp. 7-12, 1977").

[MA] E.G. Manes and M.A. Arbib. *Algebraic approaches to program semantics.* Springer Verlag, Berlin, 1986.

[Milner] R. Milner. *A calculus of communicating systems.* Lecture Notes in Computer Science, 92, Springer Verlag, Berlin, 1980.

[Moitra] A. Moitra. *Automatic construction of CSP programs from sequential nondeterministic programs.* Science of Computer Programming, 5, pp. 277-307, 1985.

[Plotkin] G.D. Plotkin. *A power domain construction.* SIAM J. Comput., 5, No. 3, 1976.

[SM] C. Strachey and R. Milne. *A theory of programming language semantics.* Chapman and Hall, London, 1977.

[Sound] N. Soundararajan. *Denotational semantics of CSP.* Theoretical Computer Science, 33, pp. 279-304, 1984.

[SS] D. Scott and C. Strachey. *Towards a mathematical semantics for computer languages.* In "Proceedings, Symposium on Computers and Automata, Microwave Research Institute, 1971".

[Stoy] J. Stoy. *Denotational semantics of programming languages: The Scott-Strachey approach.* MIT Press, Cambridge, 1977.