THE UNIVERSITY *of* EDINBURGH

AN OPERATIONAL APPROACH TO SEMANTICS AND TRANSLAT FOR CONCURRENT PROGRAMMING LANGUAGES

by

Wei Li

Doctor of Philosophy

University of Edinburgh

1982

Abstract

The problems of semantics and translation for concurrent programming languages are studied in this thesis.

A structural operational approach is introduced to specify the semantics of parallelism and communication. Using this approach, semantics for the concurrent programming languages CSP (Hoare's Communicating Sequential Processes), multitasking and exception handling in Ada, Brinch-Hansen's Edison and CCS (Milner's Calculus of Communicating Systems) are defined and some of their properties are studied.

An operational translation theory for concurrent programming languages is given. The concept of the correctness of a translation is formalised, the problem of composing transitions is studied and a composition theorem is proved. A set of sufficient conditions for proving the correctness of a translation is given.

A syntax-directed translation from CSP to CCS is given and proved correct. Through this example the proof techniques of this approach is demonstrated. Finally, as an application of operational semantics and translation, a proposal for implementing multitasking in Ada is given via a two-step syntax-directed translation.

Acknowledgements

I would like most of all to express my gratitude to my supervisor, Gordon Plotkin. His guidance, encouragement, detailed suggestions and patient reading of this thesis have had a fundamental influence on the development of this research. I am also very grateful to Matthew Hennessy and Robin Milner for acting as supervisors whilst Gordon was away.

It is hard to say sufficiently how much I have benefitted from the help of my colleagues at Edinburgh. I would like to thank all those involved in the theory of computation, especially Don Sannella and Mark Millington for helpful discussions, useful suggestions and patiently reading my thesis. Special thanks to my wife Hua and my daughter Geng. Without their sympathetic understanding, moral encouragement and self-sacrifice the thesis could not have been written.

The work here was supported in part by a studentship from the Education Ministry of China, and in part by a studentship from the University of Edinburgh.

Declaration

This thesis was composed by myself. Chapter 2 is essentially an improved version of a paper written by Gordon Plotkin (see [Plotkin 82]). Early versions of parts of Chapter 3 are published in [Hennessy and Li 82] and [Li 82]. An early version of a part of Chapter 6 is published in [Hennessy, Li and Plotkin 81]. Otherwise the work is my own, under the guidance of my supervisor Gordon Plotkin.

CONTENTS

0. Introduction

A number of programming languages intended to describe concurrent computations have been proposed in the last decade. These languages are called concurrent programming languages. Their number is not as great as that of strictly sequential languages but the number is increasing yearly. Among them, Communicating Sequential Processes ([Hoare 78]), Ada ([DoD 80]), Edison ([Brinch-Hansen 81]) and Calculus of Communication Systems ([Milner 80]) are the most influential and typical representatives. The first three are imperative languages and the last is an applicative language.

Lively research has grown up rapidly around these languages. Most of this research can be categorised into the following three areas:

A. Formalising the semantics of parallelism and communication.

B. Implementing concurrent programming languages and proving the correctness of the implementation.

C. Construction and verification of concurrent programs.

This thesis attacks the first two problems.

For the first problem, we know that in sequential languages there are four basic approaches: denotational semantics, algebraic semantics, axiomatic semantics and operational semantics. Each of these four approaches is also being applied to describing the semantics of concurrency.

We will study the semantics problem using an operational approach. Roughly speaking, the operational approach is to formally describe the execution of programs, i.e. to formalise the "operational nature" of programs. In general this purpose is

achieved by specifying some convenient abstract machine and modelling the execution of programs on that machine. This can give a hint of the way the language can be implemented.

One merit of the operational approach is that since the essential feature is to formalise the "operational nature" of programs, if a language can be implemented then its operational semantics, in principle, should be definable. In general an operational semantics differs from other approaches in that it does not require a lot of heavy mathematical machinery and is easy to understand.

The weakness of operational semantics is that because the semantics is based on an abstract machine it usually specifies some irrelevant details. This tends to make the semantics of any nontrivial language very obscure and detailed from the mathematical point of view.

To overcome this weakness or at least to reduce it to a minimum, in this thesis we introduce a new operational approach --- the structural operational approach or axiomatic operational approach developed by Plotkin and his colleagues. The basic ideas of this approach are:

a. To abstract away from the irrelevant details of the abstract machines we adopt some of the successful features of the denotational approach such as the use of abstract syntax to replace concrete syntax, and the viewing of states (stores) and environments as functions. Thus a simple configuration of an abstract machine can be written as

$$\langle S, s \rangle \quad \text{or} \quad \langle S, \rho, s \rangle$$

where S denotes the current statement to be executed, and ρ and s

denotes the current environment and state. Some other possible configurations are s (denoting normal termination resulting in the state s) and abortion (denoting abnormal termination). We use Γ to denote the set of all possible configurations.

Furthermore, to distinguish the successful executions from other computations (deadlocked and infinite computations) we introduce the set T of terminal configurations which is a subset of Γ. For example, we can take

$$T = States \cup \{abortion\} \subseteq \Gamma$$

b. We use labelled transition relations to model computation; thus a transition:

$$r \xrightarrow{\lambda} r'$$

models one elementary execution step. This transition is interpreted as the configuration r "may perform action λ to become r' " or r "is transformed to r' via the action λ". Here the action λ denotes an internal action or interactive communication with some super system or the outside world. Thus communication between concurrent "processes" can easily be captured and formalized by labelled transitions. Let Λ be the set of possible transition actions. Then the labelled transition relation

$$\longrightarrow \subseteq \Gamma \times \Lambda \times \Gamma$$

describes the possible executions of programs. Execution of a program can be viewed as a transition sequence:

$$r_0 \xrightarrow{\lambda 1} r_1 \xrightarrow{\lambda 2} r_2 \xrightarrow{\lambda 3} \ldots$$

which is either infinite or finite.

The crux of the matter lies in how to define the labelled transition relation which describe the semantics of a language. Let us consider how we could deal with two typical sequential programming language constructs in this approach:

a. Assignment statement

The semantics of the assignment statement is defined by the following axiom:

$$\langle x:=e,s\rangle \xrightarrow{\varepsilon} \langle \underline{skip}, s[v/x]\rangle \quad \text{where } v=[\![e]\!]_s$$

($[\![e]\!]_s$ is the value of the expression e in the state s.) This transition can be interpreted as saying that the execution of the statement x:=e in the state s results in a new configuration where the new statement is skip and the new state is the same as before except at x where it takes the value of e. The transition action ε means that the execution is performed without interaction with the outside world.

b. Compound statement $S_1;S_2$

The semantics of the compound statement $S_1;S_2$ is defined by the following three rules:

1. if $\langle S_1,s\rangle \xrightarrow{\lambda} \langle S_1',s'\rangle$ then $\langle S_1;S_2,s\rangle \xrightarrow{\lambda} \langle S_1';S_2,s'\rangle$.
2. if $\langle S_1,s\rangle \xrightarrow{\lambda} s'$ then $\langle S_1;S_2,s\rangle \xrightarrow{\lambda} \langle S_2,s'\rangle$.
3. if $\langle S_1,s\rangle \xrightarrow{\lambda} \underline{abortion}$, then $\langle S_1;S_2,s\rangle \xrightarrow{\lambda} \underline{abortion}$.

As usual, these rules signify that if the hypotheses of the rules define transitions then the conclusions define transitions. How can these rules be interpreted? They tell us that the execution completely depends on the execution of the first statement. The

configuration $\langle S_1, s\rangle$ can be transformed via transition action λ in three ways; the result is either a normal configuration $\langle S_1', s'\rangle$ or a normal terminal configuration s' or the abnormal terminal configuration abortion. Rule 1 says that in the first case $\langle S_1; S_2, s\rangle$ is transformed to $\langle S_1'; S_2, s'\rangle$ via the same action λ. Rule 2 says that in the second case $\langle S_1; S_2, s\rangle$ is transformed to $\langle S_2, s'\rangle$. Rule 3 deals with the third case and says that if the first statement aborts then so does the composition. To summarize we see that the above three rules formalize the description given in the Pascal report "The compound statement specifies that its component statements be executed in the same sequence as they are written" ([Jensen and Wirth 78]).

These two examples reflect the typical character of the structural approach. Two points are worth noting:

a. As with formal deductive systems of the kind employed in mathematical logic this approach defines transitions using axioms and rules. Axioms (which have no hypotheses) define the transitions directly, and rules define the transitions indirectly; that is if all hypotheses define transitions then the conclusion of the rule defines a transition. A definition of this type is called a generalized inductive definition. This feature makes the approach rigorously mathematical, allows a semantics to be set up with few preconceptions, and also determines the inductive features of the proof techniques.

b. The definition of the transition relation is based on syntactic transformations of programs and simple operations on the discrete data (state and environment). So methods of proofs rely heavily on structural induction which might easily be automated; and since programmers and language designers are already familiar with

"symbol pushing" this form of the semantics should be more acceptable to them.

These two characteristics will become more pronounced as we progress through the thesis.

Finally, in brief, a labelled transition system can be defined as a quadruple $T=\langle \Gamma, T, \Lambda, \rightarrow \rangle$ and the operational semantics of a language can be given by labelled transition systems.

The original idea of using labelled transition relations to model concurrent computations is from Keller ([Keller 75]); the use of the labelled transition systems to define operational semantics for concurrent programming languages is due to Plotkin ([Plotkin 81]). Milner gave an operational semantics for his Calculus of Communicating Systems (from now on we use the abbreviated name CCS, see [Milner 80]), and Plotkin gave an operational semantics for Hoare's Communicating Sequential Processes (from now on we use its abbreviated name CSP, see [Plotkin 82]). In this thesis we study two other concurrent programming languages (Ada and Edison) using this approach, giving an operational semantics for multitasking and exception handling in Ada and an operational semantics for Edison.

We study the second problem in a rather general way, by examining translations. Between a high level programming language and the "bare" machine on which it runs there are normally several layers represented by intermediate languages. Between each pair of consecutive levels there is a translation of high-level objects into lower-level objects. In this sense the general subject of translation is a very pervasive and important part of computer science today. The examples are too numerous to mention, but it is worth noting that, recently, several proposals for implementing tasking in Ada use this approach. In [Luckham et al 81], the

implementation of multitasking facilities in Ada is by translation into a lower level intermediate language called Adam. In another ambitious project [Bjørner and Oest 80], a semantics (or perhaps an implementation) is given for Ada by translating it into the language META+CSP. And a similar approach is taken in [Belz et al 80] where preliminary Ada has been translated into SEMANOL+Semaphores+Forking.

Although translators (of various sorts) abound, the theory of translation has received relatively little attention in the literature. If one asks whether a translation is correct, then the answer is rather vague and unsatisfactory. In practice, translators are accepted because they work well and seem to agree with the description of the language manual intuitively.

The translation problem may be formalised in the following way. A semantics for a language L can be given by

1. a semantic domain SD(L)

2. a semantic mapping M_L from the objects L (usually programs) to SD(L)

Then a translation can be viewed as a mapping $[\![\ ]\!]:L_1 \rightarrow L_2$, and its correctness can be investigated by considering the diagram

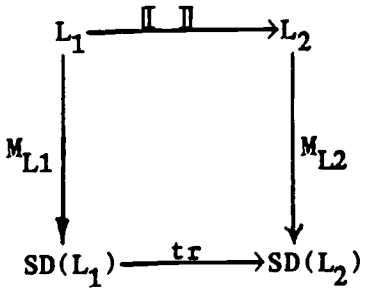


Figure 0.1

where tr is some predefined mapping between the semantic domains.

Thus by the correctness of a translation we mean that the diagram commutes, i.e

$$[\![\ ]\!] \circ M_{L2} = M_{L1} \circ tr$$

Here $[\![\ ]\!]$ is a mapping between syntactic categories and tr is a mapping between semantic domains.

This kind of approach was announced first by McCarthy and Painter ([McCarthy and Painter 67]) and then Burstall and Landin ([Burstall and Landin 70]) with the goal of making compilers for high-level programming languages completely trustworthy by proving their correctness. Morris stated his belief that the compiler correctness problem is much less general and better structured than the unrestricted program correctness problem and gave the above diagram ([Morris 73]) treating $[\![\ ]\!]$ as a compiler. Using a denotational approach he proved the correctness of a compiler for a small sequential language. Later ADJ studied this problem using an algebraic approach [ADJ 79].

It should be mentioned that all these authors are concerned only with sequential programming languages and use either denotational semantics or algebraic semantics. A problem arises when the languages include parallelism and communication as in Ada, CCS, CSP and Edison. The reason is that no satisfactory formal semantics of such a language in the denotational or algebraic style has so far been produced, though research is progressing in this area (see [Hennessy and Plotkin 80], [Plotkin 82]).

We study this problem using the structural operational approach. Roughly speaking, the basic idea of our approach is that:

1. Any syntactic translation between language L_1 and L_2 induces a

semantic translation between the transition systems which define the operational semantics of L_1 and L_2.

2. The execution of a program can only be represented by a finite transition sequence ending in a terminal configuration (successful computation), by a finite transition sequence ending in a nonterminal configuration (deadlock computation), or by an infinite transition sequence. Saying that a translation is correct amounts to saying that all these three kinds of computations for a program in the object system and in the target system correspond to each other. In particular, the possible final configurations of the translation of a program should be just the translations of the final configurations of the possible computations of the program.

3. Once we have formalised a notion of the correctness of translation, the next problem is to set up some sufficient conditions which guarantee the correctness and which can be used to prove a particular translation is correct. We call this the adequacy problem. A first attempt at a sufficient condition for the correctness of a translation might be to require

$$r \xrightarrow{\lambda} r' \quad \text{iff} \quad tr(r) \xrightarrow{tr(\lambda)} tr(r')$$

where $tr(\lambda)$ may be a sequence. This means that if a translation is adequate then any program and its translation should have the "same behaviour" in the corresponding transition systems. Intuitively, the phrase "same behaviour" includes at least that any transition of a program in the object transition system can be simulated by its translation in the target system, and any finite transition sequence from a translation of a program must be a simulation of a transition of that program. We will see later in chapter 5 that this requirement is not enough. In fact, when a transition system

describes a language with nondeterminism, parallelism and communication, the conditions which a correct translation must satisfy are very complicated. Investigating these properties is one of the main goals of this thesis.

In [Hennessy, Li and Plotkin 81] the correctness problem in the operational approach was first studied in a concrete manner, i.e. the correctness of a translation from a simple CSP (without nested parallel structures) to CCS was studied and proved. In [Hennessy and Li 82] the adequacy problem was studied in a more general setting but the conditions found were not sufficient to prove correctness. It should be mentioned that Jensen and Priese studied the simulation problem in a similar way but where only concerned with binary transition relations, without transition labels (see [Jensen 80] and [Priese 80]). In this thesis we formalise the correctness problem, study the composition of several translations, investigate the adequacy problem and give a set of sufficient conditions for correctness of a translation. To demonstrate the proof techniques used in our approach we give a translation from CSP (with nested parallel structures) to CCS and prove its correctness. Finally, as an application of operational semantics and translation theory we give a proposal for implementing multitasking in Ada by a two-step syntax-directed translation algorithm.

We now summarize the work in this thesis.

In chapter 1 the basic concepts and notations concerning labelled transition systems are introduced and two simple examples are studied. These are the evaluation of arithmetic expressions and an operational semantics for CCS. With the help of these examples it is shown how a structural operational semantics can be given using labelled transition systems and the general proof techniques used in

this approach are demonstrated.

Chapter 2 deals with the language CSP. An operational semantics for CSP is given which is an improved version of [Plotkin 82]. Both static and dynamic semantics are improved, and the interaction between static semantics and dynamic semantics is given and proved.

In chapter 3 the semantics of multitasking and exception handling in Ada is studied. The operational semantics for multitasking and exception handling are first given separately, and then these semantics are combined and the interaction between exceptions and parallelism is studied.

In chapter 4 we investigate another concurrent programming language, Edison. Unlike CCS, CSP, and Ada, communication in Edison is achieved by managing mutual exclusive access to shared variables. We define a structural operational semantics for Edison and through some examples show how this semantics works.

Chapter 5 deals with the translation problem. We introduce the concept of the correctness of a translation and prove the composition theorem. The adequacy problem is studied and a set of sufficient conditions for correctness of translation is given. Some examples are studied which show why these conditions were chosen.

In chapter 6 a syntax-directed translation from CSP (with nested parallel structure) to CCS is given which is composed of two translations. Both translations are proved correct. Through the proof we demonstrate the general strategy and techniques used in our translation theory.

Finally, in chapter 7 as an application of the operational approach to semantics and translation for concurrent programming languages we give a proposal for implementing multitasking in Ada.

The implementation is composed of two syntax-directed translations. We first translate multitasking in Ada to an Edison-like language. More precisely, the entries of the tasks are implemented by modules which contain message buffers and the communication statements of Ada are implemented by calls to the corresponding procedures of the module which manipulate the buffer. The second stage of the translation is to implement the when statement of Edison using sequential Ada plus some primitive constructs for the scheduling.

1. Labelled transition relations and operational semantics

Labelled transition relations and labelled transition systems have appeared very often in the literature of computer science in the last decade under various guises, and have been widely used as a powerful tool to define formal semantics, prove compilers correct and verify programs involving parallelism and communication (see [Keller 75], [Milner 80], [Hennessy and Plotkin 80], [Plotkin 81, 82], [Hennessy and Li 82], [Li 82], [Apt 81] etc). The purpose of this chapter is to introduce the basic concepts and notations concerning labelled transition relations and systems.

In section 1.1, the basic concept of a labelled transition relation is given. In fact, a labelled transition relation is just a binary relation with transition labels (or transition actions) on the "arrows". The concept of generalised commutative relations is given and some of its properties are studied. In section 1.2 the formal definition of a labelled transition system is introduced. The rest of the chapter is devoted to two simple but important examples --- evaluating arithmetic expressions and Milner's Calculus of Communicating Systems. With the help of these examples we demonstrate how a structural operational semantics for a language can be defined using labelled transition systems and the techniques which are available for proving the properties of such a semantics.

1.1 Labelled transition relations and their abstract properties

The concept of a labelled transition relation is a generalisation of the notion of a binary relation where a transition label (or transition action) is associated with each pair in the relation. In this section, we introduce its formal definition and study some of its properties. First we need the following standard notation:

Notation 1.1 Sequences

Let A be an arbitrary set. The sets of sequences A^+ and A^* are defined by

$$A^+=\{(a_1,a_2,\ldots,a_n)\mid n>0,\ a_i\varepsilon A,\ i=1,..,n\}$$
$$A^*=A^+ \cup \{\emptyset\}$$

We use w to denote a sequence in A^+ or A^*, and assume the functions $el_i(w)$ (the i^{th} element in w), hd(w) (the first element in w) and tl(w) (the tail of w) have their standard meanings (see [Gordon 79]). Finally, length(w) is the length of w defined by

$$\text{length}(w)=\begin{cases}0 & \text{if } w=\emptyset\\ n & \text{if } w=(a_1,a_2,..,a_n)\end{cases}$$

Labelled transition relations are defined as follows:

Definition 1.1 Labelled transition relations

Let Γ, Λ be given arbitrary sets. The elements of Γ are ranged over by r and called <u>configurations</u>, the elements of Λ are ranged over by λ and called <u>actions</u>. Then a relation $\rightarrow\subseteq\Gamma\times\Lambda\times\Gamma$ is called a Λ-<u>labelled transition relation over</u> Γ. The triple $\langle r,\lambda,r'\rangle\varepsilon\rightarrow$ is called a <u>transition</u> in $\rightarrow$ and is interpreted as "the configuration r moves to the configuration r' via the action λ", and is written as $r\xrightarrow{\lambda}r'$.

A binary relation $\rightarrow\subseteq\Gamma\times\Gamma$ can be considered as a labelled transition relation $\rightarrow\Gamma\times\Lambda\times\Gamma$ where $\Lambda=\{\tau\}$, i.e, Λ only contains one label τ; we call such a relation a <u>transition relation</u> and may write $r\rightarrow r'$ instead of $r\xrightarrow{\tau}r'$. □

From now on we will often use the following notation:

Notation 1.2

Consider a Λ-labelled transition relation over Γ, suppose $r, r' \in \Gamma$, $w \in \Lambda^*$, then:

1. $r \xrightarrow{\emptyset} r$ denotes the identity relation, for any r, $\langle r, \emptyset, r \rangle$ is in $\rightarrow$.

2. $r \xrightarrow{w} r'$ is called a transition sequence. It denotes either an identity relation if $w=\emptyset$ or, inductively, that there exists an r'' such that $r \xrightarrow{hd(w)} r''$ and $r'' \xrightarrow{tl(w)} r'$.

3. $r \xrightarrow{+} r'$ is called the <u>transitive closure</u> of $\rightarrow$. It is defined by $r \xrightarrow{+} r'$ iff there exists a $w \in \Lambda^+$ such that $r \xrightarrow{w} r'$, i.e, it denotes a non-empty transition sequence.

4. $r \xrightarrow{*} r'$ is called the <u>transitive reflective closure</u> of $\rightarrow$ defined by $r \xrightarrow{*} r'$ iff there exists $w \in \Lambda^*$ such that $r \xrightarrow{w} r'$, i.e, it denotes $r \xrightarrow{\emptyset} r'$ or $r \xrightarrow{+} r'$.

5. $\Delta(r)=\{r' \mid r \xrightarrow{\lambda} r' \ \lambda \in \Lambda;\ r' \in \Gamma\}$ is the set of <u>immediate</u> successors of r and r is called <u>active</u> iff $\Delta(r) \neq \emptyset$.

6. $\Delta^+(r)=\{r' \mid r \xrightarrow{+} r',\ r' \in \Gamma\}$ is the set of r's successors reachable in >0 transition steps.

7. $\Delta^*(r)=\{r' \mid r \xrightarrow{*} r',\ r' \in \Gamma\}$ is the set of r's successors reachable in ≥ 0 transition steps.

8. $r \downarrow r'$ says that there exists $r'' \in \Gamma$ such that $r \xrightarrow{*} r''$ and $r' \xrightarrow{*} r''$. This means that some transition relation sequences from r and r' converge.

9. $r\uparrow$ means that there is an infinite transition sequence:

$$r=r_0 \xrightarrow{\lambda 1} r_1 \xrightarrow{\lambda 2} \ldots \xrightarrow{\lambda n} r_n \rightarrow \ldots$$

If there exist no $r \varepsilon \Gamma$ such that $r\uparrow$, then the transition relation is called noetherian.

Most of the confluence properties of binary relations (see [Huet 80]) can be easily generalised to labelled transition relations. In this section we only study those generalised commutative properties which will be used in the later work concerned with translation theory (see chapter 5). For convenience we assume fixed sets Γ, Λ of configurations and labels; we use the term labelled transition relation or even transition relation to replace Λ-transition relation when Λ can be understood from the context.

Definition 1.2 Generalised commutativity

Given $M \subseteq \Lambda$, a transition relation $\rightarrow$ is said to be M commutative in Γ iff for any r, r_1, $r_2 \varepsilon \Gamma$, $\lambda \varepsilon \Lambda$, $\mu \varepsilon M$, whenever $r \xrightarrow{\lambda} r_1$, $r \xrightarrow{\mu} r_2$ then either $r_1=r_2$ and $\lambda=\mu$ or there exists $r_3 \varepsilon \Gamma$ such that $r_1 \xrightarrow{\mu} r_3$ and $r_2 \xrightarrow{\lambda} r_3$, i.e, the relations in the following diagram hold:

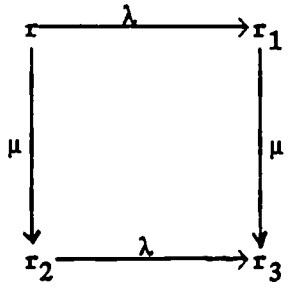


Figure 1.1

If $\rightarrow$ is M commutative then it will have the following useful property. Firstly, let us define the filtration of a sequence.

Definition 1.3 filtration

Given $M \subseteq \Lambda$. Let $fil_M: \Lambda \to \Lambda \backslash M$ be the function defined recursively by

$$fil_M(w) = \begin{cases} \emptyset & \text{if } w=\emptyset \\ hd(w).fil_M(tl(w)) & \text{if } hd(w) \notin M \\ fil_M(tl(w)) & \text{otherwise} \end{cases}$$

Thus $fil_M(w)$ is the subsequence of w obtained by taking out all w's elements contained in M. We sometimes use the abbreviated form fil(w) to replace the form $fil_M(w)$, if M can be understood from the context.

Lemma 1.1

Let $\to$ be M commutative. Then for any r, r_1, r_2 $\varepsilon \Gamma$, $w_{11} \varepsilon \Lambda^*$ and $\mu \varepsilon M$, with $r \xrightarrow{w_{11}} r_1$ and $r \xrightarrow{\mu} r_2$ there exist $r_3 \varepsilon \Gamma$, $w_{12} \varepsilon \Lambda^*$ and $w_{22} \varepsilon M^*$ such that:

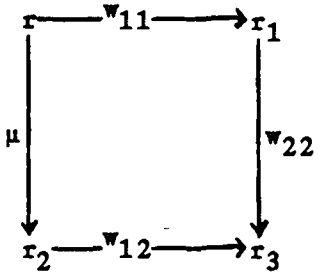


Figure 1.2

and $fil(w_{11})=fil(w_{12})$ and w_{22} is either μ or $\emptyset$.

Proof. Prove by induction on $length(w_{11})$.

If $length(w_{11})=0$ then $fil(w_{11})=\emptyset$, choose $r_3=r_2$, $w_{22}=\mu$, and $w_{12}=\emptyset$; the result is immediate.

Suppose the case where $length(w_{11})=k$ is proved and consider the

the the case where length(w_{11})=k+1. Since $r \xrightarrow{hd(w_{11})} r_{11}$ for some r_{11} and $\rightarrow$ is M commutative there are two possibilities:

1. μ=hd(w_{11}), r_2=r_{11}, then choose r_3=r_1, w_{22}=$\emptyset$, w_{12}=tl(w_{11}) and the case is proved.

2. Otherwise we can construct the following diagram:

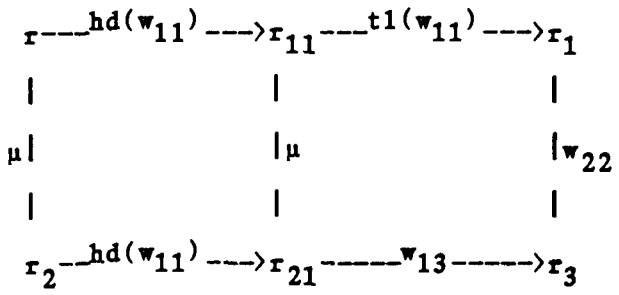


Figure 1.3

The left part of the diagram is constructed using M commutativity. And by the induction hypothesis we find r_3, w_{13} and w_{22} to construct the right part of the diagram with fil(w_{13})=fil(tl(w_{11})) and w_{22} being either μ or $\emptyset$. Thus taking w_{12}=hd(w_{11}).w_{13} the lemma is proved. □

Using this lemma we can prove the following theorem:

Theorem 1.1

If $\rightarrow$ is M commutative, then for any r, r_1, $r_2 \varepsilon \Gamma$, $w_{11} \varepsilon \Lambda^*$ and $w_{21} \varepsilon M^*$, if $r \xrightarrow{w_{11}} r_1$ and $r \xrightarrow{w_{21}} r_2$ then there exist $r_3 \varepsilon \Gamma$, $w_{12} \varepsilon \Lambda^*$ and $w_{22} \varepsilon M^*$ such that

$$fil(w_{11})=fil(w_{12}) \quad \text{and} \quad w_{22} \text{ is a subsequence of } w_{21}$$

and

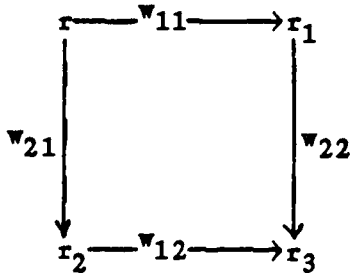


Figure 1.4

Proof. The proof can be obtained by induction on length(w_{21}) (using the above lemma). □

This theorem means that if $\rightarrow$ is M commutative then for the different transition sequences $r \xrightarrow{w_{11}} r_1$ and $r \xrightarrow{w_{21}} r_2$ where $w_{21} \varepsilon M^*$, we get Figure 1.4 which is "almost" commutative. By almost commutative we mean that w_{11} and w_{12} are the same if we filter out their elements which are in M and w_{22} is a subsequence of w_{21}.

1.2 Labelled transition systems

The concept of a labelled transition system has evolved from general automata theory. The original idea of using labelled transition systems to model parallel computation was introduced by Keller ([Keller 75]). The form used here is due to Plotkin ([Plotkin 81]). In fact, a labelled transition system can be viewed as a labelled directed graph (= labelled transition relation) with a distinguished set of terminal nodes.

Definition 1.4 Labelled transition system

A <u>labelled transition system</u> $\mathbb{T}$ is a quadruple $\langle \Gamma, T, \Lambda, \rightarrow \rangle$ where $\rightarrow \subseteq \Gamma \times \Lambda \times \Gamma$ is a labelled transition relation and $T \subseteq \Gamma$ is a set of <u>terminal configurations</u> such that:

$$\forall r \varepsilon T.\ \Lambda(r) = \emptyset$$

The set $D=\{r \mid r\in\Gamma\setminus T,\ \Delta(r)=\emptyset\}$ is called the set of deadlock configurations. In particular, if $\rightarrow$ is a binary relation (see definition 1.1) the corresponding transition system is written as $\langle\Gamma,T,\rightarrow\rangle$. □

Having defined labelled transition systems we can now introduce the notion of a computation:

Definition 1.5 Computation

Let $\Pi:\langle\Gamma,T,\Lambda,\rightarrow\rangle$ be a labelled transition system. A computation from r is either a finite sequence of transitions of the form

$$r=r_0 \xrightarrow{\lambda 1} r_1 \xrightarrow{\lambda 2} \ldots \xrightarrow{\lambda n} r_n$$

or an infinite sequence of transitions of the form

$$r=r_0 \xrightarrow{\lambda 1} r_1 \xrightarrow{\lambda 2} \ldots \xrightarrow{\lambda n} r_n \xrightarrow{\lambda n+1} \ldots$$

It is complete if it is infinite or if it is finite and $\Delta(r_n)=\emptyset$. It is stuck if it is finite and the final configuration $r_n\in D$; it is terminated if it is finite and $r_n\in T$. □

In fact any finite automaton, context free grammar, Turing machine and Petri net can be viewed as a labelled transition system. As examples, let us look at finite automata (this example is due to Plotkin) and Turing machines.

Example 1.1 Finite automata

A finite automaton M is a quintuple $\langle Q,\Sigma,\delta,q_0,F\rangle$ where:

Q is a finite set of states.

Σ is a finite set called the input alphabet.

$\delta: Q\times\Sigma\rightarrow Q$ is the state transition relation.

$q_0 \varepsilon Q$ is the initial state.

$F \subseteq Q$ is the set of final states.

To obtain a transition system we set:

$\Gamma = Q \times \Sigma^*$ and $T = \{\langle q, \emptyset \rangle \mid q \varepsilon F\}$

So any configuration $r = \langle q, w \rangle$ has a control component, q, and a state component w (for the data).

The transition relation is defined by:

$\langle q, w \rangle \longrightarrow \langle q', tl(w) \rangle$ if $w \neq \emptyset$ and $\delta(q, hd(w)) = q'$

The behaviour of the finite automaton is just the set L(M) of strings it accepts:

$L(M) = \{w \mid w \varepsilon \Sigma^* \text{ and } \exists q \varepsilon F \ \langle q_0, w \rangle \xrightarrow{*} \langle q, \emptyset \rangle \}$.

Example 1.2 Turing machines

A Turing machine is defined by $(Q, A, \Sigma, \delta, q_0, B, F)$ where:

Q is a finite set of states.

A is a finite set of allowable tape symbols.

$B \varepsilon A$ is the blank symbol.

$\Sigma \subseteq A - \{B\}$ is the set of input symbols.

$\delta : Q \times A \longrightarrow Q \times A \times \{L, R\}$ is the next move function (δ may be undefined for some arguments).

$q_0 \varepsilon Q$ is the initial state.

$F \subseteq Q$ is a set of final states.

To obtain a labelled transition system we define:

$\Gamma = Q \times N \times A^*$ and $T = F \times N \times A^*$

where N is the set of natural numbers. Thus any configuration $r=\langle q,n,w\rangle$ has a control component q, a tape w and a pointer n which gives the position on the tape scanned by the tape head.

The transition relation is defined as follows

$$\langle q,n,w\rangle \longrightarrow \langle q',n-1,w[a'/n]\rangle \quad \text{if } \delta(q,el_n(w))=(q',a',L) \text{ and } n>1$$

$$\langle q,n,w\rangle \longrightarrow \langle q',n+1,w'\rangle \quad \text{if } \delta(q,el_n(w))=(q',a',R)$$

where

$$w' = \begin{cases} w[a'/n] & \text{if } n<length(w) \\ w[a'/n].B & \text{if } n=length(w) \end{cases}$$

The notation $w[a/n]:A^* \longrightarrow A^*$ where $0 \leq n \leq length(w)$ is defined by

$$el_i(w[a/n]) = \begin{cases} a & \text{if } i=n \\ el_i(w) & \text{otherwise} \end{cases}$$

i.e, w[a/n] denotes the tape which is the same as w except that the n^{th} element is a.

The behaviour of the Turing machine is just the set L(M) of strings it accepts:

$$L(M)=\{w \mid \exists q \in F,\ n \in N,\ w' \in A^*\ (\langle q_0,0,w\rangle \xrightarrow{*} \langle q,n,w'\rangle)\}\ \square$$

The above examples tell us that the behaviour of a Turing machine (or a finite automaton) can be describe by labelled transition systems. In fact, we will see that the operational semantics of

languages concerned with parallelism and communication can also be given by labelled transition systems. In the rest of the chapter we will show, by means of two interesting examples, how they can be used to model parallel computation.

1.3 Evaluating arithmetic expressions

As a useful example of defining an operational semantics using a labelled transition system, we consider the problem of evaluating arithmetic expressions.

1.3.1 Abstract syntax

The abstract syntax of arithmetic expressions is defined using the following disjoint syntactic sets:

N - the set of natural numbers, ranged over by m, n.

Var - a given countably infinite set of variables, ranged over by (the meta variable) x.

Bop={+,-,*,div} - the set of binary operations, ranged over by o.

Now we can define:

Exp - the set of arithmetic expressions, ranged over by e, and defined using a BNF like notation

$$e ::= m \mid x \mid e_1+e_2 \mid e_1-e_2 \mid e_1*e_2 \mid e_1 \underline{\text{div}}\ e_2$$

It should be mentioned here that we sometimes add subscripts and superscripts to a metavariable to generate another metavariable over the same class such as x', x_{ij}, e_1, e', e_{ij} and so on.

In the title of this subsection we used the term abstract syntax

to distinguish it from concrete (or "normal") syntax (see [McCarthy 63]). In general, by abstract syntax we mean a collection of sets and rules (operations) showing how to construct all phrases of the language in the form of trees rather than character strings (as a concrete syntax does). Abstract syntax does not worry about ambiguity, operator precedence etc. These issues must be treated by the concrete syntax in order for it to be useful for parsing. For example, the concrete syntax of arithmetic expessions might be as follows (see [Tennent 81]):

⟨expression⟩ ::= ⟨term⟩ | ⟨expression⟩⟨o⟩⟨term⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩⟨o⟩⟨factor⟩
⟨factor⟩ ::= ⟨variable⟩ | ⟨literal⟩ | (⟨expression⟩)
⟨o⟩ ::= * | + | - | div
⟨variable⟩ ::= a | b | ... | z
⟨literal⟩ ::= 0 | 1 | ... | 9

A concrete syntax like this is necessary for a parser to recognize character strings describing expressions, terms, and factors, but these do not make any difference from the semantic point of view. Roughly speaking, we may view the abstract syntax as describing the syntax trees produced by a parser which utilises the concrete syntax, but ignores semantically irrelevant details like those between expressions, terms and factors. Therefore using abstract syntax we can stress the "deep structure" of languages and avoid getting involved with irrelevant details of the parsing process.

Remark: From now on the syntax of all languages which we will study in this thesis are defined by an abstract syntax.

1.3.2 Operational semantics

The idea of defining an operational semantics for evaluating arithmetic expressions is that:

1. To evaluate an expression e we need to start with an initial state s and putting these together we obtain a configuration $\langle e,s\rangle$.

2. The evaluation of e in a state s should result in either a number associated or an error (when runtime errors occur during evaluation such as 1 div 0). Thus $\langle n,s\rangle$ (n is an integer) and error are terminal configurations.

3. Since we are interested in digital computation the evaluation (execution) will move through discrete stages. We may use transitions $r \rightarrow r'$ to model one step of evaluation (execution) and use the transition sequences to model the working processes of expression evaluation. To define transitions we introduce axioms and rules. For example, consider the expression $e_1 \circ e_2$ and initial state s. Then one step of evaluation should be $\langle e_1 \circ e_2, s\rangle \rightarrow r$ where r is a configuration. Since we can choose either e_1 or e_2 to evaluate there are two possibilities:

a. $\langle e_1, s\rangle \rightarrow r_1$

b. $\langle e_2, s\rangle \rightarrow r_2$

Let us consider case (a). One step of evaluation of $\langle e_1, s\rangle$ may result in a proper successor or a runtime error, that is, r_1 may be $\langle e_1', s\rangle$ or error. Thus the result of one step of evaluation of $\langle e_1 \circ e_2, s\rangle$ will naturally be $\langle e_1' \circ e_2, s\rangle$ or error and we obtain the following two rules:

1. if $\langle e_1, s\rangle \rightarrow \langle e_1', s\rangle$ then $\langle e_1 \circ e_2, s\rangle \rightarrow \langle e_1' \circ e_2, s\rangle$

2. if $\langle e_1, s\rangle \rightarrow$ error then $\langle e_1 \circ e_2, s\rangle \rightarrow$ error

Similarly, for case (b) we have:

3. if $\langle e_2, s\rangle \rightarrow \langle e_2', s\rangle$ then $\langle e_1 \circ e_2, s\rangle \rightarrow \langle e_1 \circ e_2', s\rangle$

4. if $\langle e_2, s\rangle \rightarrow \underline{error}$ then $\langle e_1 \circ e_2, s\rangle \rightarrow \underline{error}$

In fact these four rules exactly model the way in which we evaluate arithmetic expressions by hand. To formalise the above explanation we introduce the following transition system:

Firstly, let a state $s:Var \rightarrow N$ be a partial assignment of values to variables and States be the class of states. The transition system $T_e = \langle \Gamma_e, T_e, \xrightarrow{e} \rangle$ is defined by:

$$\Gamma_e = \{\langle e, s\rangle \mid e \in Exp,\ s \in States\} \cup \{\underline{error}\}$$

$$T_e = \{\langle m, s\rangle \mid m \in N,\ s \in States\} \cup \{\underline{error}\}$$

The general forms of the transition relation $\xrightarrow{e}$ are

$$\langle e, s\rangle \xrightarrow{e} \langle e', s'\rangle$$
$$\langle e, s\rangle \xrightarrow{e} \underline{error}$$

These mean that one step in the evaluation of e (with state s) results in the expression e' (with state s') or an error. For the sake of simplicity we will omit the e under the arrow since there is only one transition relation. We will make use of the following notation:

Notation 1.3
The form $\frac{A, B}{C, D}$ denotes that A and B implies C and D.

The transition relation is defined by the following informal system of <u>axioms</u> and <u>rules</u>:

Identifiers

1. $\langle x,s\rangle \rightarrow \langle s(x),s\rangle$ if s(x) is defined

2. $\langle x,s\rangle \rightarrow \underline{\text{error}}$ if s(x) is undefined

Binary operations

1. $$\frac{\langle e_1,s\rangle \rightarrow \langle e_1',s\rangle}{\langle e_1 \circ e_2,s\rangle \rightarrow \langle e_1' \circ e_2,s\rangle}$$

2. $$\frac{\langle e_1,s\rangle \rightarrow \underline{\text{error}}}{\langle e_1 \circ e_2,s\rangle \rightarrow \underline{\text{error}}}$$

3. $$\frac{\langle e_2,s\rangle \rightarrow \langle e_2',s\rangle}{\langle e_1 \circ e_2,s\rangle \rightarrow \langle e_1 \circ e_2',s\rangle}$$

4. $$\frac{\langle e_2,s\rangle \rightarrow \underline{\text{error}}}{\langle e_1 \circ e_2,s\rangle \rightarrow \underline{\text{error}}}$$

5. $\langle m+m',s\rangle \rightarrow \langle n,s\rangle$ where $n=m+m'$

6. $\langle m-m',s\rangle \rightarrow \langle n,s\rangle$ if $m \geq m'$ and $n=m-m'$

7. $\langle m-m',s\rangle \rightarrow \underline{\text{error}}$ if $m<m'$

8. $\langle m*m',s\rangle \rightarrow \langle n,s\rangle$ where $n=m*m'$

9. $\langle m \underline{\text{div}}\ m',s\rangle \rightarrow \langle n,s\rangle$ if $m' \neq 0$, $m=n*m'+r$, $0 \leq r < m'$.

10. $\langle m \underline{\text{div}}\ m',s\rangle \rightarrow \underline{\text{error}}$ if $m'=0$

Let us explain what is meant by this system of axioms and rules. There are two axioms for identifiers and six axioms for binary operations (5 to 10); there are also four rules for binary

operations. Axioms define transitions directly and rules enable us to derive transitions from axioms inductively. The set of transitions in $\mathbb{T}_e$ satisfies the following three laws:

1. The axioms define transitions.

2. If all the hypotheses of a rule (the numerator) define transitions the conclusions (the denominator) of the rule define transitions.

3. There are no other transitions in $\mathbb{T}_e$.

This kind of approach is called a generalised inductive definition (see [Shoenfield 1967]). For example, the first axiom for identifiers says that for any identifier x and state s, if s(x) is defined then $\langle x,s\rangle \rightarrow \langle s(x),s\rangle$ is a transition in $\mathbb{T}_e$. The second axiom for identifiers says that if s(x) is undefined then $\langle x,s\rangle \rightarrow$ error is a transition. Axioms for binary operations about "+", "-", "*" and "div" are similar. The first rule for binary operations says that for any expressions e_1, e_2, e_1' and state s, if there is a transition $\langle e_1,s\rangle \rightarrow \langle e_1',s\rangle$ then there is a transition $\langle e_1 \circ e_2,s\rangle \rightarrow \langle e_1' \circ e_2,s\rangle$. For example, let s be (x=5,y=6) (this means s(x)=5 and s(y)=6; otherwise s is undefined). Since

$$\langle x,(x=5,y=6)\rangle \rightarrow \langle 5,(x=5,y=6)\rangle$$

is a transition (by the first axiom for identifiers) we have by rule 1

$$\langle x*y,(x=5,y=6)\rangle \rightarrow \langle 5*y,(x=5,y=6)\rangle$$

is a transition in $\mathbb{T}_e$.

It should be emphasized that the symbol + appears twice in rule 3, the first "+" in the clause is a syntactic constructor (operator) and the second denotes the addition function. Similarly, for the "-", and "*" in subsequent rules. We will often overload

symbols in this way when their meanings can be understood from the context.

We introduce the value of an expression as follows:

Definition 1.6

The function $[\![\ ]\!]$: Exp $\rightarrow$ States $\rightarrow$ N+{error} is defined by

$$[\![e]\!]_s = \begin{cases} m & \text{if } \langle e,s\rangle \xrightarrow{*} \langle m,s\rangle \\ \underline{\text{error}} & \text{if } \langle e,s\rangle \xrightarrow{*} \underline{\text{error}} \end{cases}$$

We call $[\![e]\!]_s$ the value of the expression e in state s. □

We now prove that the evaluation mechanism defined above, is noetherian, never stuck, and that each expression has a unique value which does not depend on the particular computation sequence (thereby proving that $[\![e]\!]_s$ is well-defined). If we use num(r) to denote the number of operations and identifiers contained in expression e, then all these proofs can be obtained by induction on num(r) and for each r by cases on the structure of r. Now num(r) is recursively defined by the following table:

r	error	$\langle m,s\rangle$	$\langle mon,s\rangle$	$\langle x,s\rangle$	$\langle e_1 o e_2,s\rangle$
num(r)	0	0	1	1	num($\langle e_1,s\rangle$)+num($\langle e_2,s\rangle$)+1

Notice that if $r \rightarrow r'$ then $r \notin T_e$ (because there is no rule for error and $\langle n,s\rangle$). We have:

Lemma 1.2

If $r \rightarrow r'$ then num(r)>num(r').

Proof By induction on num(r). In the case num(r)=0, then $r \in T_e$ and the result is trivial since as we noted it is not then possible that

$r \rightarrow r'$. Now suppose $r=\langle e,s\rangle$. If $num(r)=1$ then e must be an identifier x or have the form mom'. In both cases the result is immediate since r' must be $\langle m,s\rangle$ or error.

Now we assume as our hypothesis that the lemma is true whenever $num(r) \leq k$ $(k \geq 1)$ and prove it then holds for $num(r)=k+1$. By the above discussion we can assume $k \geq 2$, so according to the syntax we need only examine the case of $e=e_1 \circ e_2$ and either $num(\langle e_1,s\rangle)>0$ or $num(\langle e_2,s\rangle)>0$. By the transition rules, we must analyse four subcases:

a. $\langle e_1,s\rangle \rightarrow \langle e_1',s\rangle$ and $\langle e_1 \circ e_2,s\rangle \rightarrow \langle e_1' \circ e_2,s\rangle$ (by rule 1).
We have by the induction hypothesis that $num(\langle e_1',s\rangle) < num(\langle e_1,s\rangle)$, therefore

$$num(\langle e_1' \circ e_2,s\rangle) = num(\langle e_1',s\rangle) + num(\langle e_2,s\rangle) + 1$$
$$< num(\langle e_1,s\rangle) + num(\langle e_2,s\rangle) + 1 = num(\langle e_1 \circ e_2\rangle)$$

b. $\langle e_1,s\rangle \rightarrow$ error and $\langle e_1 \circ e_2,s\rangle \rightarrow$ error and the result is obvious.

c. $\langle e_2,s\rangle \rightarrow \langle e_2',s\rangle$ the proof is similar to subcase (a).

d. $\langle e_2,s\rangle \rightarrow$ error the proof is similar to subcase (b).

Thus the lemma has been proved. □

Corollary 1.1
The transition relation $\xrightarrow{e}$ is noetherian.

Proof It follows directly from the above lemma. □

This corollary says that all computations from a configuration r are finite with length at most num(r).

Lemma 1.3
If $r \notin T_e$ then there must exist $r' \in \lceil_e$ such that $r \rightarrow r'$.

Proof The proof is by structural induction on the expression contained in r. Since r is not in T_e it must be one of the form: $\langle x,s\rangle$, $\langle m_1 \circ m_2, s\rangle$ and $\langle e_1 \circ e_2, s\rangle$ where at least one of e_1 or e_2 is not an integer. Thus:

case 1. r is $\langle x,s\rangle$.

According to the axioms for identifiers, if s(x) is defined then take $r'=\langle s(x),s\rangle$, otherwise take r'=<u>error</u>.

case 2. r is $\langle m_1 \circ m_2, s\rangle$.

According to the axioms for binary operations we take r' defined below:

$$r' = \begin{cases} \langle m,s\rangle \quad m=m_1 \circ m_2 & \text{if } \circ \text{ is } + \text{ or } * \\ & \text{or } \circ \text{ is } - \text{ and } m_1 > m_2 \\ & \text{or } \circ \text{ is } \underline{\text{div}} \text{ and } m_2 \neq 0 \\ \underline{\text{error}} & \text{if } \circ \text{ is } - \text{ and } m_1 < m_2 \\ & \text{or } \circ \text{ is } \underline{\text{div}} \text{ and } m_2 = 0 \end{cases}$$

case 3. r is $\langle e_1 \circ e_2, s\rangle$. Then

a. if e_1 is not an integer, then by the induction hypothesis

either $\langle e_1,s\rangle \rightarrow \langle e_1',s\rangle$ or $\langle e_1,s\rangle \rightarrow$ <u>error</u>

By the binary operation rule 1 and 2 we have:

$\langle e_1 \circ e_2, s\rangle \rightarrow \langle e_1' \circ e_2, s\rangle$ or $\langle e_1 \circ e_2, s\rangle \rightarrow$ <u>error</u>

Thus take $r'=\langle e_1' \circ e_2, s\rangle$ or <u>error</u> the subcase is done.

b. if e_2 is not an integer, the proof is similar to subcase (a).

Thus the lemma has been proved. □

Corollary 1.2

No configuration is deadlocked and no computation is stuck.

proof It follows directly from the above lemma. □

The corollary 1.1 and 1.2 mean that every complete computation is

terminated and every expression has either a value or an error from every state.

Finally, the transition relation $\xrightarrow[e]{}$ has the confluence property.

Lemma 1.4

For any $r \in \Gamma$, if $r \rightarrow r_1$ and $r \rightarrow r_2$ then $r_1 \downarrow r_2$.

Proof By induction on num(r) again.

For the case of num(r)=0 since r can only be $\langle m,s \rangle$ or error the result holds vacuously. For num(r)=1 r must be $\langle x,s \rangle$ or $\langle mem',s \rangle$, only one transition rule applies in each case so $r_1=r_2$ and hence $r_1 \downarrow r_2$.

Now we assume as our hypothesis that the lemma is true whenever $num(r) \leq k$ $(k \geq 1)$ and prove it then holds for num(r)=k+1. Let $r=\langle e_1 \circ e_2, s \rangle$. According to the transition rules defined above we need to examine the following cases:

case 1. r_1=error. Then

$\langle e_1,s \rangle \rightarrow$error or $\langle e_2,s \rangle \rightarrow$error

For example $\langle e_1,s \rangle \rightarrow$error; then if $r_1=r_2$ there is nothing to prove, otherwise there are two subcases:

a. $r_2=\langle e_1 \circ e_2',s \rangle$ and $\langle e_2,s \rangle \rightarrow \langle e_2',s \rangle$.

By the binary operation rule 1 we have:

$r_2=\langle e_1 \circ e_2',s \rangle \rightarrow$error

b. $r_2=\langle e_1' \circ e_2,s \rangle$ and $\langle e_1,s \rangle \rightarrow \langle e_1',s \rangle$.

By the induction hypothesis we have: $\langle e_1',s \rangle \xrightarrow{*}$error. According to the binary operation rule 1 we have:

$r_2=\langle e_1' \circ e_2, s\rangle \xrightarrow{*} \underline{error}$

Thus in both subcases r_1 and r_2 converge to error.

case 2. $r_2=$error. The proof is similar to case 1.

case 3. $\langle e_1 \circ e_2, s\rangle \rightarrow r_1$ and $\langle e_1 \circ e_2, s\rangle \rightarrow r_2$ and neither r_1 nor r_2 is error.

According to the binary operation rules 1 and 3 there are three subcases:

a. $r_1=\langle e_1' \circ e_2, s\rangle$, $r_2=\langle e_1 \circ e_2', s\rangle$ and $\langle e_1, s\rangle \rightarrow \langle e_1', s\rangle$, $\langle e_2, s\rangle \rightarrow \langle e_2', s\rangle$. Take $r_3=\langle e_1' \circ e_2', s\rangle$. By rules 1 and 3 we have

$\langle e_1' \circ e_2, s\rangle \rightarrow \langle e_1' \circ e_2', s\rangle$ and $\langle e_1 \circ e_2', s\rangle \rightarrow \langle e_1' \circ e_2', s\rangle$

b. $r_1=\langle e_1' \circ e_2, s\rangle$, $r_2=\langle e_1'' \circ e_2, s\rangle$ and $\langle e_1, s\rangle \rightarrow \langle e_1', s\rangle$, $\langle e_1, s\rangle \rightarrow \langle e_1'', s\rangle$. Since $num(e_1) \leq k$, by the induction hypothesis, there exists an r_{13} such that

$\langle e_1', s\rangle \xrightarrow{*} r_{13}$ and $\langle e_1'', s\rangle \xrightarrow{*} r_{13}$

If $r_{13}=\langle e_{13}, s\rangle$ then take $r_3=\langle e_{13} \circ e_2, s\rangle$; otherwise r_{13} must be error and take $r_3=$error.

c. $r_1=\langle e_1 \circ e_2', s\rangle$, $r_2=\langle e_1 \circ e_2'', s\rangle$ and $\langle e_2, s\rangle \rightarrow \langle e_2', s\rangle$, $\langle e_2, s\rangle\rangle \rightarrow \langle e_2'', s\rangle$. The proof is similar to subcase (b).

Thus the theorem has been proved. □

In [Huet 80] the property established by the above lemma is called local confluence and the property

if $r \xrightarrow{*} r_1$ and $r \xrightarrow{*} r_2$ then $r_1 \downarrow r_2$

is called the confluence property. It is proved that a noetherian relation is confluent iff it is locally confluent (see lemma 2.4 in[Huet 80]). This result guarantees that all complete computations in $\mathbb{T}_e$ end in the same configuration, i.e, given state s every expression e has unique value; thus as promised above, we have shown that $[\![e]\!]_s$ is indeed well-defined.

In fact using the techniques given in [Plotkin 80] it can be proved that the operational semantics defined here is equivalent to the denotational semantics given in [Gordon 79] or [Stoy 77].

Remarks As we have noticed the main feature of the proofs of the above lemmas is induction on the structure of terms in configurations. The reason is that all transitions in the transition system T_e are given by generalised inductive definition; i.e, a transition in T_e is either an axiom or the conclusion of a rule. Thus in order to prove that every transition in T_e has a property P, it suffices to prove

1. every axiom has property P.

2. if all of hypotheses of a rule have property P, then the conclusion of the rule has property P.

In fact this approach will be used throughout the whole thesis.

1.4 An operational semantics of CCS

In a series of papers(see [Hennessy and Milner 79], [Milner 80 and 82], [Hennessy and Plotkin 80], etc) Milner and his colleagues have studied a model of parallelism in which concurrent systems communicate by sending and receiving values along lines. Communication is synchronized in that the exchange of values takes place only when the sender and receiver are both ready and this exchange is considered as a single event. This kind of communication is also found in a large group of modern languages such as Hoare's CSP, and Ada. In [Milner 80] a notation for expressing systems is introduced which can be considered as an applicative language, called CCS - Calculus of Communicating Systems. More precisely, there is a family of languages incorporating these ideas. In this

section we study one such language, which is close to the style given by Hennessy and Plotkin (see [Hennessy and Plotkin 80]) and is called asynchronous CCS. For the sake of convenience we just use the name CCS.

1.4.1 The syntax of CCS

The abstract syntax of CCS is parameterised on certain disjoint sets and functions:

Var - a given countably infinite set of variables, ranged over by (the metavariable) x.

Exp - a given countably infinite set of expressions, ranged over by e and assumed to contain the set V which is a nonempty set of values (ranged over by v).

Bexp - a given countably infinite set of boolean expressions, ranged over by b and assumed to contain the set {tt,ff} of truth values.

Remarks: From now on the sets Var, Exp and Bexp will be used in every language which we will study. As we have already stated in the introduction, since the goal of this thesis is to investigate the nature of communication and concurrency, in order to focus our attention on these subjects we gloss over the details of those aspects which are nearly standard and quote the results directly. Following this principle, for the expressions we assume that

1. All expressions e or b have finite sets FV(e), FV(b) of free variables, defined in the normal fashion.

2. The substitution of an expression e' for a variable x in an expression e or b is defined as usual, giving expressions e[e'/x],

b[e'/x]. We assume that the following standard facts hold:

$$FV(e[e'/x]) = \begin{cases} (FV(e)\setminus\{x\}) \cup FV(e') & \text{if } x \in FV(e) \\ FV(e) & \text{if } x \notin FV(e) \end{cases}$$

and $e[e'/x][e''/x] = e[e'[e''/x]/x]$

and $e[e'/x][e''/y] = e[e''/y][e'[e''/y]/x]$ if $x \notin FV(e'')$ and $x \neq y$.

This will do if we are not thinking of complicated expressions with bound variables where also a notion of α-conversion should be taken into account (see [Hindley, Lercher, Seldin 72], [Curry, Feys, Craig 68]).

3. All expressions can be evaluated without side-effects to give a result in V (in following chapters we assume that V contains a distinguished value error). The evaluation may be defined using either a denotational or an operational approach which, for example, is referred to the previous example. We use $[\![e]\!]$ to denote the value of a closed expression e for an applicative language and $[\![e]\!]_s$ (or $[\![e]\!]_{\rho s}$) to denote the value of e in state s (or in environment ρ and store s) for an imperative language. We should mention that in general, the evaluation of an expression may have side-effects(see [Gordon 79], [Tennent 81]) but here we just choose the simple case and assume evaluate expressions without side-effects in order to concentrate on communication.

The following sets are also needed to define the syntax of CCS:

Δ – a given countably infinite set of line names, ranged over by α, β, γ.

Proc – a given countably infinite set of procedure names, ranged over by P.

Given these five sets the two main syntactic categories of CCS can be specified as follows:

Ren - the set of renamings, ranged over by ϕ, which is a partial function from Δ to Δ.

Terms - the set of terms, ranged over by t, u and defined by the BNF-like notation:

$$t ::= Nil \mid t+u \mid t|u \mid \alpha x.t \mid \alpha(e,t) \mid \tau.t \mid$$
$$t[\phi] \mid \underline{if}\ b\ \underline{then}\ t\ \underline{else}\ u \mid P(e_1,\ldots,e_n) \mid$$
$$(\mu P(x_1,\ldots,x_n).t)(e_1,\ldots,e_n)$$

The notations $\alpha x.t$ and $(\mu P(x_1,\ldots,x_n).t)(e_1,\ldots,e_n)$ denote the binding of variables and procedures respectively in t. The term t+u is called summation and can behave as either u or t. The term t|u is called composition and the components t and u may execute simultaneously and communicate with each other. The terms $\alpha x.t$, $\alpha(e,t)$ and $\tau.t$ are actions, they can perform an input, output or internal action respectively and then become t. It should be mentioned that the forms of actions used here are sometimes written in the forms $\alpha?x.t$, $\alpha!e.t$ and $\tau.t$ (see [Milner 80]). The term $t[\phi]$ is a renaming; the function ϕ renames or restricts the line names contained in t. Finally, $(\mu P(x_1,\ldots x_n).t)(e_1,\ldots,e_n)$ is a recursive procedure with parameters $x_1,\ldots x_n$. We can understand that the behaviour of $(\mu P(x_1,\ldots,x_n).t)(e_1,\ldots,e_n)$ is the same as its body t with $\mu P(x_1,\ldots,x_n).t$ substituted for P and with the free occurrences of the formal parameters $x_1,\ldots, x_n$ set to $[\![e_1]\!],\ldots, [\![e_n]\!]$. For convenience we will use the following notations:

Let $\{t_i \mid 1\leq i\leq n\}$ be a finite set of terms; then Σt_i denotes the term $(t_1+(t_2+(\ldots+t_n)\ldots))$ and $|t_i$ denotes the term

$t_1|(t_2|(\ldots|t_n)\ldots)$. Let X be a subset of line names; then t\X denotes the term $t[\phi_X]$, where ϕ_X is a renaming defined by:

$$\phi_X(\alpha)=\begin{cases}\alpha & \text{if } \alpha\notin X\\ \\ \text{undefined} & \text{otherwise}\end{cases}$$

Finally, the term $t[\alpha_i/\beta_i]$, $1\leq i\leq n$ denotes $t[\phi]$ where

$$\phi(\gamma)=\begin{cases}\alpha_i & \text{if } \gamma=\beta_i\\ \\ \gamma & \text{otherwise}\end{cases}$$

We also extend FV(t) (free variables of a term t), and define FP(t) (free procedure names of t) and FL(t) (free line names of t) by the following tables:

	Nil	t + u	t ‖ u	αx.t	α(e,t)
FV	∅	FV(t)∪FV(u)	FV(t)∪FV(u)	FV(t)\{x}	FV(e)∪FV(t)
FP	∅	FP(t)∪FP(u)	FP(t)∪FP(u)	FP(t)	FP(t)
FL	∅	FL(t)∪FL(u)	FL(t)∪FL(u)	{α}∪FL(t)	{α}∪FL(t)

	τ.t	t[φ]	if b then t else u
FV	FV(t)	FV(t)	FV(b)∪FV(t)∪FV(u)
FP	FP(t)	FP(t)	FP(t)∪FP(u)
FL	FL(t)	φ(FL(t))	FL(t)∪FL(u)

	$P(e_1,\ldots,e_n)$	$(\mu P(x_1,\ldots,x_n).t)(e_1,\ldots,e_n)$
FV	$FV(e_1)\cup\ldots\cup FV(e_n)$	$FV(t)\setminus\{x_1,\ldots,x_n\}$
FP	$\{P\}$	$FP(t)\setminus\{P\}$
FL	$\emptyset$	$FL(t)$

where $\phi(FL(t))$ denotes the image under ϕ of $FL(t)\subseteq\Delta$.

Remarks We also assume the substitution $e'[e/x]$ of expressions for variables in expressions is extended in the usual way to allow the substitution $t[e/x]$ of expressions for variables in terms. We also assume the substitution $t[t'/P]$ of terms for procedure names in terms is defined in the usual way. (Of course in both cases such substitutions may require change of bound variables such as in $\alpha x.t$, or bound procedure names such as in $\mu P.t$ to avoid clashes).

Finally, a program is defined as follows:

Definition 1.7 CCS program

A term t is a <u>program</u> iff $FV(t)=\emptyset$ and $FP(t)=\emptyset$.

That is there are no undefined procedures or variables in a program.

1.4.2 The operational semantics of CCS

To give the operational semantics of CCS we define a labelled transition system $\mathbb{T}_c=\langle\Gamma_c,T_c,\Lambda_c,\xrightarrow{c}\rangle$ where

Γ_c=Terms

$T_c=\{Nil\}$

$\Lambda_c=\{\alpha?v \mid \alpha\in\Delta,\ v\in V\}\cup\{\alpha!v \mid \alpha\in\Delta,\ v\in V\}\cup\{\tau\}$

For any transition label $\lambda\in\Lambda_c$ the complementary label $\bar{\lambda}$ is defined by:

$$\bar{\lambda}=\begin{cases}\alpha!v & \text{if} \quad \lambda=\alpha?v \\ \alpha?v & \text{if} \quad \lambda=\alpha!v \\ \tau & \text{if} \quad \lambda=\tau\end{cases}$$

The transition relation $\xrightarrow[c]{}$ will be defined by rules of the following forms:

1. $t \xrightarrow{\alpha?v} t'$ means informally that t can receive an input value v along line α and thereby be transformed into t'.

2. $t \xrightarrow{\alpha!v} t'$ means that t can output value v along line α and be transformed into t'.

3. $t \xrightarrow{\tau} t'$ means that t can transform itself to t' by some internal communication.

The transition relation is defined as follows:

Action

1. $\alpha x.t \xrightarrow{\alpha?v} t[v/x]$

2. $\alpha(e,t) \xrightarrow{\alpha![\![e]\!]} t$

3. $\tau.t \xrightarrow{\tau} t$

Summation

1. $\dfrac{t \xrightarrow{\lambda} t'}{t+u \xrightarrow{\lambda} t'}$

2. $\dfrac{u \xrightarrow{\lambda} u'}{t+u \xrightarrow{\lambda} u'}$

These two rules mean that the behaviour of the term t+u is that of its components t and u, with commitment to whichever is executed.

Composition

1. $\dfrac{t \xrightarrow{\lambda} t'}{t \mid u \xrightarrow{\lambda} t' \mid u}$

2. $\dfrac{u \xrightarrow{\lambda} u'}{t \mid u \xrightarrow{\lambda} t \mid u'}$

3. $\dfrac{t \xrightarrow{a?v} t',\ u \xrightarrow{a!v} u'}{t \mid u \xrightarrow{\tau} t' \mid u'}$

4. $\dfrac{t \xrightarrow{a!v} t',\ u \xrightarrow{a?v} u'}{t \mid u \xrightarrow{\tau} t' \mid u'}$

Rules 1 and 2 above mean that an action of t or of u in the composition t|u yields an action of this composition in which the other component is not affected. In other words, parallelism is achieved by interleaving the execution of its components (t and u). Rules 3 and 4 mean that communication between terms consists of the simultaneous occurrence of certain specified complementary actions of these terms. This kind of communication mechanism is often referred to in the literature as handshaking. Since in this thesis, we mainly deal with parallelism by interleaving and handshake communication, these four transition rules express the essence of our approach to parallelism and communication.

Renaming

1. $\dfrac{t \xrightarrow{a?v} t'}{t[\phi] \xrightarrow{\phi(a)?v} t'[\phi]}$ if $\phi(a)$ is defined.

2. $\dfrac{t \xrightarrow{a!v} t'}{t[\phi] \xrightarrow{\phi(a)!v} t'[\phi]}$ if $\phi(a)$ is defined.

3. $\dfrac{t \xrightarrow{\tau} t'}{t[\phi] \xrightarrow{\tau} t'[\phi]}$

These rules mean that renaming either relabels the transition action (when ϕ is defined) or removes the transition action (when ϕ is undefined).

Conditional

1. $$\frac{t \xrightarrow{\lambda} t',\ [\![b]\!]=tt}{\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u \xrightarrow{\lambda} t'}$$

2. $$\frac{u \xrightarrow{\lambda} u',\ [\![b]\!]=ff}{\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ u \xrightarrow{\lambda} u'}$$

These rules mean that the behaviour of a conditional term is that of t or u, depending on the value of the boolean expression b (tt or ff).

Procedure

$$\frac{t[\mu P(x_1,\ldots x_n).t/P][[\![e_1]\!]/x_1]\ldots[[\![e_n]\!]/x_n] \xrightarrow{\lambda} t'}{(\mu P(x_1,\ldots x_n).t)(e_1,\ldots,e_n) \xrightarrow{\lambda} t'}$$

This means that the behaviour of $(\mu P(x_1,\ldots x_n).t)(e_1,\ldots,e_n)$ is the same as its body t substituted for P in t by $\mu P(x_1,\ldots x_n).t$ and with the formal parameters $x_1,\ldots,x_n$ set to $[\![e_1]\!],\ldots,[\![e_n]\!]$.

It should be mentioned that CCS is an applicative language and therefore all these rules fail to describe the meaning of terms with free variables. For example, the action rule 2 gives no meaning to the term:

$$\alpha(x^2+1,Nil)$$

and the conditional rules do not make sense for

$$\underline{\text{if}}\ y>0\ \underline{\text{then}}\ \alpha(y,Nil)\ \underline{\text{else}}\ \beta(y+1,Nil)$$

since all rules involve values,not variables.

1.4.3 Properties and examples

One may wonder whether the operational semantics given above is well defined. The following results may convince us:

Lemma 1.5

If $t \xrightarrow{\lambda} t'$ then $FV(t') \subseteq FV(t)$, $FP(t') \subseteq FP(t)$ and $FL(t') \subseteq FL(t)$.

Proof Let us just prove the result for FV leaving FP and FL to the interested reader. The proof is by induction on inferences $t \xrightarrow{\lambda} t'$.

For the case that the inference rules are axioms we need only examine the following three cases:

case 1. $t=\alpha(e,t_1)$ and $\alpha(e,t_1) \xrightarrow{\alpha!\llbracket e \rrbracket} t_1$. So $t'=t_1$ therefore

$$FV(t')=FV(t_1) \subseteq (FV(e) \cup FV(t_1))=FV(t)$$

case 2. $t=\alpha x.t_1$ and $\alpha x.t_1 \xrightarrow{\alpha?v} t[v/x]$. Then $t'=t_1[v/x]$. Noticing the second remarks given in subsection 1.4.1 we have:

$$FV(t')=FV(t_1[v/x])=(FV(t_1)\setminus\{x\}) \cup FV(v)=FV(t_1)\setminus\{x\}=FV(t)$$

case 3. $t=\tau.t_1$ and $\tau.t \xrightarrow{\tau} t$. Then the result is obvious.

For the induction step we must examine all rules defined in the previous section. Let us check the following interesting cases:

case 4. $t=t_1+t_2$. There are two possibilities:

a. $t'=t_1'$ and $t_1 \xrightarrow{\lambda} t_1'$; then

$$FV(t')=FV(t_1') \subseteq FV(t_1) \quad \text{by the induction hypothesis}$$
$$\subseteq FV(t_1+t_2) \quad \text{by the definition of FV}$$

b. $t'=t_2'$ and $t_2 \xrightarrow{\lambda} t_2'$. Then the proof is similar to (a).

case 5. $t=t_1 \mid t_2$. There are three subcases to check:

a. $t'=t_1' \mid t_2$ and $t_1 \xrightarrow{\lambda} t_1'$. We have, by the induction hypothesis that $FV(t_1') \subseteq FV(t_1)$ so

$$FV(t')=FV(t_1' \mid t_2)=(FV(t_1') \cup FV(t_2)) \subseteq (FV(t_1) \cup FV(t_2))=FV(t)$$

b. $t'=t_1 \mid t_2'$ and $t_2 \xrightarrow{\lambda} t_2'$. The proof is similar to (a).

c. $t'=t_1' \mid t_2'$, $\lambda=\tau$ and $t_1 \xrightarrow{\rho} t_1'$ and $t_2 \xrightarrow{\bar{\rho}} t_2'$ for some $\rho \varepsilon \Lambda_c$. By the induction hypothesis $FV(t_1') \subseteq FV(t_1)$ and $FV(t_2') \subseteq FV(t_2)$, so

$$FV(t')=(FV(t_1') \cup FV(t_2')) \subseteq (FV(t_1) \cup FV(t_2))=FV(t)$$

case 6. t is a recursive procedure. For simplicity we just consider the parameterless case $t=\mu P.t_1$. Then by the procedure rule

$$t'=t_1' \text{ and } t_1[\mu P.t_1/P] \xrightarrow{\lambda} t_1'.$$

By the induction hypothesis we have:

$$FV(t_1') \subseteq FV(t_1[\mu P.t_1/P])$$

Since

$$FV(t_1[\mu P.t_1/P])=\begin{cases} FV(t_1) \cup FV(\mu P.t_1) & \text{if } P \varepsilon FP(t_1) \\ FV(t_1) & \text{if } P \notin FP(t_1) \end{cases}$$

by the definition of FV we have:

$$FV(t)=FV(\mu P.t_1)=FV(t_1)=FV(t_1[\mu P.t_1/P])$$

Thus the result has been proved. □

The next theorem follows directly from this lemma.

Theorem 1.2

If t is a CCS program and $t \xrightarrow{\lambda} t'$ $\lambda \varepsilon \Delta$ then t' is a program.

Let us now study some CCS examples.

Example 1.3

Consider a term $t=a(5,Nil) \mid \bar{a}x.Nil \mid \bar{a}.(a(x,Nil))$. According to the operational semantics there are many computations from t. Here we examine two of them.

computation 1.

$$t \xrightarrow[1]{\tau} Nil \mid \bar{a}x.Nil \mid a(5,Nil)$$
$$\xrightarrow[2]{\tau} Nil \mid Nil \mid Nil$$

The transition step 1 is obtained by the composition rule 3, since

a. $\alpha(5,Nil) \xrightarrow{\alpha!5} Nil$ and

b. $\alpha x.Nil \mid \alpha x.(\alpha(x,Nil)) \xrightarrow{\alpha?5} \alpha x.Nil \mid \alpha(5,Nil)$ by the composition rule 2 since

$\alpha x.(\alpha(x,Nil)) \xrightarrow{\alpha?5} \alpha(5,Nil)$ by the action rule 1.

Similarly we have:

computation 2.

$t \xrightarrow{\tau} Nil \mid Nil \mid \alpha x.(\alpha(x,Nil))$ since

$\alpha(5,Nil) \xrightarrow{\alpha!5} Nil$ by the action rule 2 and

$\alpha x.Nil \xrightarrow{\alpha?5} Nil$ by the action rule 1

then apply rule 3 followed by rule 1.

These two computations tell us that CCS unlike the transition system for evaluating expressions (using binary relations) or the lambda calculus, is not confluent. □

Example 1.4 Hoare-Zhou protocol

A simple example is given to show how to specify the kind of protocol given by Hoare and Zhou (see [Zhou and Hoare 81] and [Hennessy 81] for a general discussion).

A communication protocol may be composed of a sender and a receiver connected by a medium which may corrupt the message (see figure 1.5). It accepts a value at the transmitting end (the line IC) and accurately reproduces this value at the receiving end (the line OC).

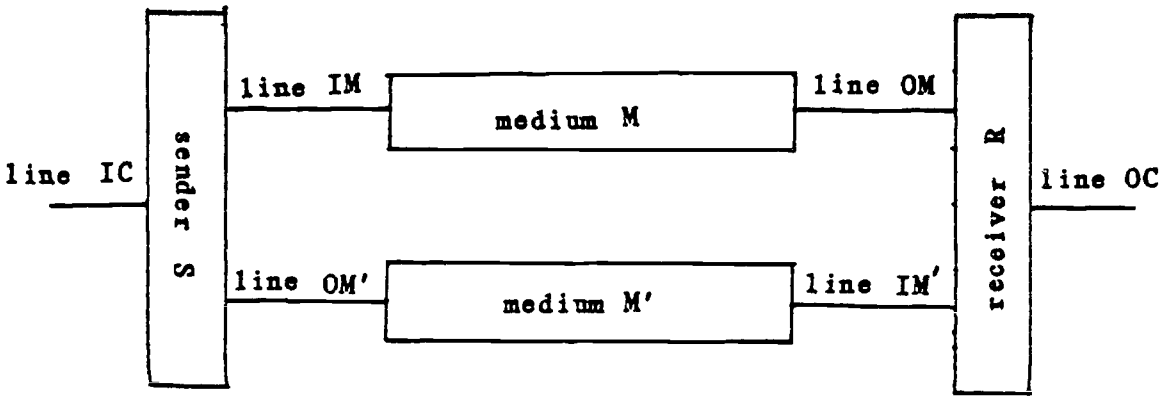


Figure 1.5

The protocol can be specified in CCS as follows,

corrupt(x) is a given function which models the degradation of value x during transmission. We assume corrupt(error)=error and that error can be transmitted along lines as a normal value.

correct(x) is another function which renews the value x, producing an error if renewing is impossible.

A uni-directional communication medium M can be specified by

$$M=\mu M.IMx.OM(corrupt(x),M)$$

This means that M receives information from the line IM and outputs it through the line OM. The information may be corrupted when it passes through the medium.

The sender can be specified by:

$$S=\mu S.ICx.S'$$

where S' is $\mu S'.IM(x,OM'y.[$ if y=error then S' else S])

This means that the sender S receives a value x from outside (through line IC) and sends it to the receiver through medium M (the line IM), then receives information y from the second medium M' (the

line OM'). If it receives an error (a fail transmission), it sends the value x again until the received information indicates successful transmission (y≠error), then whereupon it accepts a new value from outside. Similarly, the receiver is defined by:

$R=\mu R.OMx.R'$ where R' is:

IM'(correct(x), if correct(x)=error then R' else OC(correct(x),R))

Finally, the protocol is defined by

$$(S|M|M'|R)\setminus\{OM, IM, OM', IM'\}$$

where $M'=M[OM'/OM,\ IM'/IM]$. □

Example 1.5 A sorting algorithm

Given two sets of numbers S0 and T0, the problem is to design an algorithm which produces sets S', T' such that

$$S0 \cup T0 = S' \cup T',\ |S'|=|S0|,\ |T'|=|T0|$$

and

$$\forall n \in S',\ m \in T'.\ n \leq m$$

where |H| denotes the number of elements in any given set H.

Suppose Var, Exp defined in section 1.4.1 contains the usual set operations and let max(S), min(S) denote the maximum and minimum elements of S respectively. Then the algorithm can be described as follows:

$$(P(S0) \mid Q(T0))\setminus\{\alpha,\beta\}$$

where P(S) and Q(T) are defined by

$P(S) = \mu P(S).\beta(\max(S), P'(S))$

$P'(S) = \alpha x.[\underline{if}\ \max(S) > x\ \underline{then}\ P(\{x\} \cup S \backslash \max(S))\ \underline{else}\ \gamma(S, Nil)]$

$Q(T) = \mu Q(T).\beta y.(\alpha(\min(T), Q'(T))$

$Q'(T) = \underline{if}\ \min(T) < y\ \underline{then}\ Q(\{y\} \cup S \backslash \min(T))\ \underline{else}\ \delta(S, Nil)$

where γ and δ are line names communicating with outside of the algorithm.

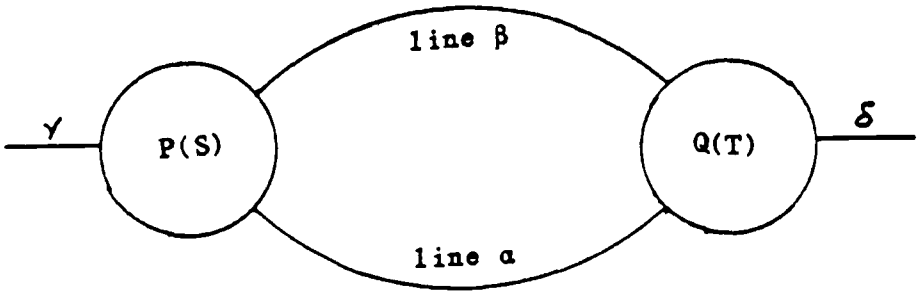


Figure 1.6

Informally, P(S) means

a. send max(S) to T through line β (see Figure 1.6)

b. receive min(T) from T through line α

c. if max(S)>min(T) then exchange these two elements

d. repeat a, b and c until $\max(S) \leq \min(S)$.

e. output result and finish.

The meaning of Q(T) is similar. □

2. An operational semantics for CSP

Hoare proposed in [Hoare 78] a language called Communicating Sequential Processes which has exerted a profound influence on both theoretical and practical research in computer science. The design of CSP is based on the following basic ideas:

1. Parallelism is obtained by concurrently executing a number of sequential processes, where the sequential structures are straight-forwardly adapted from Dijkstra's guarded command language ([Dijkstra 76]). All processes start simultaneously, and a parallel structure reaches completion when and only when all its component processes have finished.

2. Input and output commands are basic components of the language and are the sole means for communication between concurrent processes. Communication between two concurrent processes occurs only when one process - the sender - names another as target for output and the other - the receiver - names the first as source for input and both processes are ready to communicate simultaneously. Communication between processes via global variables is not allowed and there is no automatic buffering. Handshaking is the only method of communication and synchronization.

In this chapter an operational semantics of CSP is given. The idea of defining a semantics for CSP using labelled transition systems is due to Plotkin (see [Hennessy, Li and Plotkin 81], [Plotkin 82]). In section 1 an abstract syntax of CSP is given which is slightly different from Hoare's notation. In section 2 the concept of static semantics is introduced , then in section 3 we give the operational semantics for CSP. In section 4 a number of facts are proved to show that this operational semantics is

well-defined. Some simple examples are also given to show how this semantics works. In section 5 some further problems concerned with failure and termination are studied.

It should be mentioned that the this chapter is an improved version of [Plotkin 82]. Both static and dynamic semantics are improved, and the interaction between static and dynamic semantics is given and proved.

2.1 The syntax of CSP

As we have already pointed out, CSP is obtained from Dijkstra's guarded command language (see [Dijkstra 76]) by adding input, output and parallel commands which are introduced to express communication and parallelism. The syntax of CSP is parameterised on the following syntactic categories:

Plab - a given countably infinite set of process labels, ranged over by P, Q, R.

Pten - a given countably infinite set of pattern symbols, ranged over by W.

The categories of guarded commands and commands of CSP are defined using a BNF-like notation.

Gcm - the set of guarded commands, ranged over by GC

$$GC ::= b \Rightarrow C \mid GC \,[]\, GC$$

Com - the set of commands, ranged over by C

C ::= skip | abort | x:=e | C;C | if GC fi | do GC od |
P?W(x) | Q!W(e) | C||C | P::C | process P;C

where GC [] GC is called an alternative (guarded command), if GC fi is called a conditional (command), do GC od is called a repetitive (command) and P?W(x), Q!W(e) are called input and output commands respectively. Finally, P::C denotes a process named P with body C and process P;C is the declaration of the label P. The main differences from Hoare's original proposal in [Hoare 78] are as follows:

A. The form of guarded commands is different

In [Hoare 78] a guarded command can take one of the following forms

b→C, b,P?x→C or b,P!e→C

where b, b,P?x and b,P!e are called guards (see 7.2 and 7.8 in [Hoare 78]). The motivation of guarded commands is that the command C may be executed if and only if the guard does not fail. For example, in b→C the command C may be executed iff the boolean expression b is true, and in b,P?x→C the command C may be executed iff b is true and the input command in the guard is executed; for the latter both conditions are necessary. Instead of the single arrow → we use here the double arrow ⇒ introduced by Plotkin (it is also introduced in Ada in a restricted form --- see section 9.7 in [DoD 80]) and employ the strong form of guarded command b⇒C. The motivation is that C is executed if and only if b is true and the command C can be executed (for at least one step), i.e, the first transition of b⇒C is the first transition of C if b is true. For example, in the guarded commands

b⇒P?x;C, b⇒P!e;C or b⇒(do true ⇒P?x od);C

the guarded command is executed if and only when b is true and

respectively P?x or P!e or do true ⇒P?x od is executed. It is obvious that b→C can be written as b⇒skip;C and b,P!e→C can be written as b⇒P!e;C and so on. The merit of this form is that it is simpler and more general than the original form. The following example clarifies the difference between these two forms of guards.

Example 2.1

Three commands are given below

C1 (Hoare's form)

```
[ R::if true,P?x→skip[]true,Q?y→P!1 fi
|| Q::R!0
|| P::R?z]
```

The process R must choose the second alternative of the guarded command since the guard does not fail (Q?y can be executed), thus process R communicates with process Q first and then communicates with process P and terminates normally.

C2 (Hoare's form)

```
[ R::if true→P?x[]true→Q?y;P!0 fi
|| Q::R!1
|| P::R?z ]
```

Since both guards in process R do not fail, either alternative can be chosen; if the first is chosen then the program will deadlock.

C3 (Plotkin's form)

```
[ R::if true⇒P?x[]true⇒Q?y;P!0 fi
‖ Q::R!0
‖ P::R?z ]
```

The first alternative true⇒P?x cannot be chosen since P?x can never be executed, therefore the situation is the same as that of C1.

B. The parallel structure is different.

In [Hoare 78] a normal parallel command may be written as

```
[ P::C1 ‖ C ‖ P2::C2 ]
```

In fact this form combines three different language features:

1. A parallel structure is constructed from its constituent structures using the symbol ‖.

2. These constituent structures may be specified as named processes here P1::C1 and P2::C2.

3. Square brackets delimit the scope of process labels.

Our syntax specifies this parallel structure in the traditional way that separates these three features and uses three different syntactic forms:

C ‖ C and P::C and process P;C

The first specifies parallel execution, the second (sometimes called process body) defines a constituent sequential process (which may be named) and the third declares the scope of the given process label. Thus the command [P1::C1‖C‖P2::C2] can be written as

process P1;(process P2;(P1::C1‖(C‖P2::C2)))

And nested parallel commands such as

[P::Q!5‖Q::[P::R!0‖Q::P?x]]

can be written as

process P;process Q;(P::Q!5
‖Q::(process P;process Q;(P::R!0‖Q::P?x)))

We will see later in the next chapter that this feature makes it easier to treat the various scoping situations.

C. The pattern-matching mechanism is introduced explicitly.

For example, in the commands P!W(e) and Q?W(x) the letter W indicates a pattern-matching mechanism, the command P!W(e) says to send the value of e with pattern named W to the process P. The command Q?W(x) says that variable x receives a value from process Q matching the pattern W. Different W's preclude communication from taking place.

D. Miscellaneous matters

Finally, it should be mentioned that variable declarations, process arrays and guarded commands with ranges are omitted in the syntax, but communication and parallel commands are carefully treated, permitting nested parallel structures and even parallel commands inside repetitive commands.

2.2 Static semantics

Before giving a formal operational semantics for CSP, it is necessary to guarantee that all syntactic clauses are valid (or well-formed) . By valid we mean that a program could be successfully compiled, i.e, there is no error in the program according to the requirements of the language manual. For example, the following commands (programs)

P::C1 || P::C2 and P1::x:=5+y || P2::Q?x

are invalid because the first violates Hoare's requirement "Each process label in a parallel command must occur only once" and the second violates "Each process of a parallel command must be disjoint from every other process of the command, in the sense that it does not mention any variable which occurs as a target variable in any other process" (see 2.1 [Hoare 78]). But the command

P1::x:=5+y || P2::z:=2+y

is valid since it satisfies these requirements and these are the only such requirements in CSP. The above fact tells us that a formal description of static semantics is needed to specify such requirements before giving the operational semantics . The metnod used here is due to Plotkin in [Plotkin 81, 82]. To this end, let Syn denote the union GcomuCom and Ω be a metavariable of Syn, i.e, Ω can be a guarded command or a command. We define the following sets:

RV(Ω) - the set of variables in Ω that may be read.

WV(Ω) - the set of variables in Ω that may be written to.

FPL(Ω) - the set of free (agent) process labels contained in Ω.

For a guarded command GC we need the following predicate:

Bool(GC) - the disjunction of guards occurring in GC. We call the guarded command GC open in state s if $[\![Bool(GC)]\!]_s$=tt, otherwise it is called closed.

All these sets and the the predicate Bool(GC) are defined by induction on the structure of (guarded) commands Ω in the tables below:

	b⇒C	GC1▯GC2
RV	FV(b)∪RV(C)	RV(GC1)∪RV(GC2)
WV	WV(C)	WV(GC1)∪WV(GC2)
FPL	FPL(C)	FPL(GC1)∪FPL(GC2)
Bool	b	Bool(GC1)∨Bool(GC2)

	skip	abort	x:=e	C1;C2	if GC fi	do GC od
RV	∅	∅	FV(e)	RV(C1)∪RV(C2)	RV(GC)	RV(GC)
WV	∅	∅	{x}	WV(C1)∪WV(C2)	WV(GC)	WV(GC)
FPL	∅	∅	∅	FPL(C1)∪FPL(C2)	FPL(GC)	FPL(GC)

	P?W(x)	Q!W(e)	C1‖C2	R::C	process R;C
RV	∅	FV(e)	RV(C1)∪RV(C2)	RV(C)	RV(C)
WV	{x}	∅	WV(C1)∪WV(C2)	WV(C)	WV(C)
FPL	∅	∅	FPL(C1)∪FPL(C2)	{R}	FPL(C)\{R}

It is intended to be evident from these tables that RV(C1;C2)=RV(C1)∪RV(C2) and WV(C1‖C2)=WV(C1)∪WV(C2) and FPL(GC1▯GC2)=FPL(GC1)∪FPL(GC2) and so on. Finally, the set of free variables contained in a command or a guarded command is defined by

$FV(C)=RV(C)\cup WV(C)$ and $FV(GC)=RV(GC)\cup WV(GC)$

The property of being valid or well-formed is written as $\vdash$ and

$\vdash_{gc}$ GC means that the guarded command GC is valid.

$\vdash_{c}$ C means that the command C is valid.

These are also defined by structural induction but they are presented in a more illuminating way by rules similar to but simpler than those for transition relations.

Guarded commands

1. $\dfrac{\vdash_c C}{\vdash_{gc} b\Rightarrow C}$

2. $\dfrac{\vdash_{gc} GC1,\ \vdash_{gc} GC2}{\vdash_{gc} GC1[]GC2}$

Commands

1. $\vdash_c$ skip

2. $\vdash_c$ abort

3. $\vdash_c x:=e$

4. $\dfrac{\vdash_c C1,\ \vdash_c C2}{\vdash_c C1;C2}$

5. $\dfrac{\vdash_{gc} GC}{\vdash_c$ if GC fi$}$

6. $\dfrac{\vdash_{gc} GC}{\vdash_c$ do GC od$}$

7. $\vdash_c P?W(x)$

8. $\vdash_c Q!W(e)$

9. $\dfrac{\vdash_c C,\ FPL(C)=\emptyset}{\vdash_c R::C}$

10. $\dfrac{\vdash_c C}{\vdash_c$ process R;C$}$

11. $\dfrac{\vdash_c C1,\ \vdash_c C2}{\vdash_c C1\|C2}$ (if $FV(C1)\cap WV(C2)=FV(C2)\cap WV(C1)=\emptyset$ $FPL(C1)\cap FPL(C2)=\emptyset$)

The requirements of non-interference and disjoint process names

are imposed in rule 11. The condition FPL(C)=∅ in rule 9 requests that a process declaration and the corresponding process body must occur in the same innermost process body containing them and declaration first. Let us check the following example:

Example 2.2

Consider the command:

C1 = process P;(
process R;(P::R!W(6)
‖R::(process A;(process P;(A::P?W(x)‖P::A!W(5))))

The corresponding Hoare's form of the command is

[P::R!W(6)‖R::[A::P?W(x)‖P::A!W(5)]]

This command consists of two levels of processes: the processes R and (outer) P construct the first level, the processes A and (inner) P construct the second level. The command is valid because:

1. By rule 7 and 8 commands P?W(x) and A!W(5) are valid.
2. Noticing FPL(P?W(x))=FPL(A!W(5))=∅ by rule 9 the commands A::P?W(x) and P::A!W(5) are valid.
3. According to rule 11 the parallel command A::P?W(x)‖P::A!W(5) is valid.
4. Let C11 denote the command:
process A;(process P;(A::P?W(x)‖P::A!W(5))).
Using rule 10 twice C11 is valid.
5. By the definition of FPL we have FPL(C11)=∅, so according to rule 9 the command R::C11 is valid.
6. By rule 11 the command P::R!W(6)‖R::C is valid.
7. Finally, using rule 10 twice we know the command C1 is valid.

Let us consider the command:

C2 = process P;(

process R;(P::R!W(6) ‖ R::(A::P?W(x)‖P::A!W(5))))

The Hoare's form of C2 is:

[P::R!W(6)‖R::(A::P?W(x)‖P::A!W(5))]

It is obviously invalid since the command

R::(A::P?W(x)‖P::A!(5))

is invalid, as FPL(A::P?W(x)‖P::A!W(5))={A,P}. □

It is hoped the above illustrates the beginnings of a method for specifying the static semantics of programming languages(it is also called the context-sensitive aspects).

2.3 Operational semantics

In this section an operational semantics for CSP is given using a labelled transition system. In contrast with the static semantics defined in the previous section, the operational semantics is sometimes called dynamic semantics. The idea of defining operational semantics is similar to that in evaluating expressions (see section 1.3). The execution of a command C should contain the following steps:

1. Since the execution of C must start with an initial state, say s, putting these together we obtain a configuration ⟨C,s⟩.

2. The execution may be infinite (a loop occurs) or finite. If it is finite then it may deadlock or terminate; and for the termination case it may terminate normally or abnormally. Thus, naturally, the terminal configurations should be s (for a normal termination) and abortion (for an abnormal termination).

3. We use a transition to model one step of execution and a transition sequence to model an execution. All transitions are

defined using a generalised inductive definition. For example, consider the execution of the output command P!W(x) in a state s where s(x)=2. Intuitively, this execution is to evaluate x (in this case $[\![x]\!]_s=2$), output the value of x with pattern W to the process P and then transform to a new configuration, say $\langle C',new(s)\rangle$. Let us choose C'=skip, and obviously the state s is not changed. Using a transition to describe this working process we have:

$$\langle P!W(x),\ s\rangle \xrightarrow{(*,P)!W(2)} \langle \underline{skip},\ s\rangle$$

The transition label (*,P)!W(2) is called an output action and denotes interaction with the outside world, meaning that the value 2 with pattern W is output to process P. The symbol * denotes the name of the process where in the output command occurs. Since at this stage the process containing the output command is unknown the symbol * is called the unknown process name. Similarly, for an input command Q?W(y) with state s, the transition is:

$$\langle Q?W(y),\ s\rangle \xrightarrow{(*,Q)?W(v)} \langle \underline{skip},\ s[v/y]\rangle$$

This means that the value v with a matched pattern W is received from process Q and the transformed configuration is $\langle skip, s[v/y]\rangle$. The notation s[v/y] indicates that the value of y is changed to v (the received value). The transition label (*,Q)?W(v) is called an input action and denotes that a value v is received from process Q with a matched pattern W. Again * is the name of unknown process in which the input command occurs.

When the commands P!W(x) and Q?W(y) appear in the processes Q and P respectively, for example, we have the commands Q::P!W(e) and P::Q?W(x). Then the transitions must be:

$$\langle Q::P!W(x),s\rangle \xrightarrow{(Q,P)!W(2)} \langle Q::\underline{skip},s\rangle$$

and

$$\langle P::Q?W(y),s\rangle \xrightarrow{(P,Q)?W(2)} \langle P::\underline{skip},s[2/y]\rangle$$

The transition label (Q,P)!W(2) is obtained from (*,P)W!(2) by substituting Q for *; it means that the agent (process) Q sends the value 2 with pattern W to the object (process) P. The transition label (P,Q)?W(2) is obtained from (*,Q)?W(2) by substituting P for *; it means that the agent process P receives the value 2 with matched pattern W from the object process Q. This embodies the proposition that a process name "serves as a name for the command list to which it is prefixed" (see 2.1 [Hoare 78]). In fact, transition labels (Q,P)!W(2) and (P,Q)?W(2) are complementary labels (actions) which "fuse together" in handshake communication. Thus for the parallel command Q::P!W(x)‖P::Q?W(y) the expected transition is:

$$\langle Q::P!W(x)\|P::Q?W(y),s\rangle \xrightarrow{\varepsilon} \langle Q::\underline{skip}\|P::\underline{skip},s[2/y]\rangle$$

The transition label ε means that no interaction occurs with the outside of the parallel command; that is, the interaction is internal to the command during its execution.

We are now ready to give the formal operational semantics using labelled transition systems. It is clear that different syntactic categories need different labelled transition systems, here $\langle \Gamma_c, T_c, \Lambda_c, \xrightarrow{c} \rangle$ for commands and $\langle \Gamma_{gc}, T_{gc}, \Lambda_{gc}, \xrightarrow{gc} \rangle$ for guarded commands. The configuration sets are defined by

$$\Gamma_c = \{\langle C,s\rangle \mid C \in Com,\ s \in States\} \cup States \cup \{\underline{abortion}\}$$

$$\Gamma_{gc} = \{\langle GC,s\rangle \mid GC \in Gcm,\ s \in States\} \cup \Gamma_c$$

and the terminal configurations are defined by:

$$T_c = States \cup \{\underline{abortion}\}$$

$$T_{gc} = \Gamma_c$$

As we have already explained above, in the configuration $\langle C,s\rangle$, C and s stand for the current statement to be executed and state respectively. A state s alone denotes normal termination and abortion stands for abnormal termination corresponding to Hoare's failure execution.

The sets of transition labels are defined by:

$$\Lambda_c = \Lambda_{gc} = \{\varepsilon\} \cup \{(N,P)?W(v) \mid N \in Plab \cup \{*\},\ P \in Plab,\ W \in Pten,\ v \in V\}$$
$$\cup \{(N,P)!W(v) \mid N \in Plab \cup \{*\},\ P \in Plab,\ W \in Pten,\ v \in V\}$$

The meaning of these labels is given by the following table:

label λ	agent	action	object	pattern
ε		internal action		
$(N,P)!W(v)$	N	outputs v to	P	with W
$(N,P)?W(v)$	N	inputs v from	P	matching W

where $P \in Plab$, $N \in Plab \cup \{*\}$ and * denotes a unknown process name. We use $\bar{\lambda}$ to denote the complement of the label λ defined by:

$$\bar{\lambda} = \begin{cases} (P,Q)?W(v) & \text{if } \lambda = (Q,P)!W(v) \\ (P,Q)!W(v) & \text{if } \lambda = (Q,P)?W(v) \end{cases}$$

where P,QεPlab, WεPten, vεV.

Finally, we introduce the following notations to describe the transition rules:

Notation 2.1

1. $r \xrightarrow{\lambda} r_1 \mid r_2 \mid \ldots \mid r_n$ means that $r \xrightarrow{\lambda} r_i$ for i=1,..., n.

2. $\dfrac{r \xrightarrow{\lambda} r_1 \mid r_2 \mid \ldots \mid r_n}{r' \xrightarrow{\lambda'} r_1' \mid r_2' \mid \ldots \mid r_n'}$ is an abbreviation for

$\dfrac{r \xrightarrow{\lambda} r_i}{r' \xrightarrow{\lambda'} r_i'}$ where i=1,..., n

The transition rules are given below (note all the subscripts under the arrows are left off and both $\xrightarrow[c]{}$ and $\xrightarrow[gc]{}$ are written as $\rightarrow$ when the intended transition can be derived from the context).

Guarded commands

guards

1. $\dfrac{\langle C,s\rangle \xrightarrow{\lambda} r,\ [\![b]\!]_s = tt}{\langle b \Rightarrow C,s\rangle \xrightarrow{\lambda} r}$

2. $\dfrac{[\![b]\!]_s = \underline{error}}{\langle b \Rightarrow C,s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$

Alternative

1. $\dfrac{\langle GC_1,s\rangle \xrightarrow{\lambda} \langle C_1,s'\rangle \mid s' \mid \underline{abortion}}{\langle GC_1 [\!] GC_2,s\rangle \xrightarrow{\lambda} \langle C_1,s'\rangle \mid s' \mid \underline{abortion}}$

2. $\dfrac{\langle GC_2,s\rangle \xrightarrow{\lambda} \langle C_2,s'\rangle \mid s' \mid \underline{abortion}}{\langle GC_1 [\!] GC_2,s\rangle \xrightarrow{\lambda} \langle C_2,s'\rangle \mid s' \mid \underline{abortion}}$

The guard rules mean that if boolean expression b is true then the behaviour of b⇒C is the same as the behaviour of C; it will result

in abortion if the value of b is an error.

The alternative rules are in keeping with the requirement "if all guards fail, the alternative command fails. Otherwise an arbitrary one with successfully executable guard is selected and executed" (see 2.4 [Hoare 78]).

Commands

$$\text{assign}\ 1.\ \frac{[\![e]\!]_s = v}{\langle x:=e, s\rangle \xrightarrow{\varepsilon} \langle \text{skip}, s[v/x]\rangle}$$

$$2.\ \frac{[\![e]\!]_s = \text{error}}{\langle x:=e, s\rangle \xrightarrow{\varepsilon} \text{abortion}}$$

$$\text{skip}\qquad \langle \text{skip}, s\rangle \xrightarrow{\varepsilon} s$$

$$\text{abort}\qquad \langle \text{abort}, s\rangle \xrightarrow{\varepsilon} \text{abortion}$$

$$\text{input}\qquad \langle P?W(x), s\rangle \xrightarrow{(*,P)?W(v)} \langle \text{skip}, s[v/x]\rangle$$

$$\text{output}\ 1.\ \frac{[\![e]\!]_s = v}{\langle P!W(e), s\rangle \xrightarrow{(*,P)!W(v)} \langle \text{skip}, s\rangle}$$

$$2.\ \frac{[\![e]\!]_s = \text{error}}{\langle P!W(e), s\rangle \xrightarrow{\varepsilon} \text{abortion}}$$

$$\text{composition}\quad \frac{\langle C_1, s\rangle \xrightarrow{\lambda} \langle C_1', s'\rangle \mid s' \mid \text{abortion}}{\langle C_1; C_2, s\rangle \xrightarrow{\lambda} \langle C_1'; C_2, s'\rangle \mid \langle C_2, s'\rangle \mid \text{abortion}}$$

The first three rules are obvious and we have already seen the input and output rules. The composition rule means that the execution of $C_1;C_2$ consists of the execution of C_1 followed by the execution of C_2, i.e, if C_1 aborts so does the composition $C_1;C_2$, if C_1

terminates normally then the subsequent behaviour of $C_1;C_2$ is identical to that of C_2, otherwise the execution of $C_1;C_2$ is the same (initially) as the execution of C_1.

conditional 1. $$\frac{\langle GC,s\rangle \xrightarrow{\lambda} \langle C,s'\rangle \mid s' \mid \text{abortion}}{\langle \underline{\text{if}}\ GC\ \underline{\text{fi}},s\rangle \xrightarrow{\lambda} \langle C,s'\rangle \mid s' \mid \underline{\text{abortion}}}$$

2. $$\frac{[\![\text{Bool}(GC)]\!]_s = ff}{\langle \underline{\text{if}}\ GC\ \underline{\text{fi}},s\rangle \xrightarrow{\varepsilon} \underline{\text{abortion}}}$$

repetition 1. $$\frac{\langle GC,s\rangle \xrightarrow{\lambda} \langle C,s'\rangle}{\langle \underline{\text{do}}\ GC\ \underline{\text{od}},s\rangle \xrightarrow{\lambda} \langle C;\underline{\text{do}}\ GC\ \underline{\text{od}},s'\rangle}$$

2. $$\frac{\langle GC,s\rangle \xrightarrow{\lambda} s' \mid \text{abortion}}{\langle \underline{\text{do}}\ GC\ \underline{\text{od}},s\rangle \xrightarrow{\lambda} \langle \underline{\text{do}}\ GC\ \underline{\text{od}},s'\rangle \mid \underline{\text{abortion}}}$$

3. $$\frac{[\![\text{Bool}(GC)]\!]_s = ff}{\langle \underline{\text{do}}\ GC\ \underline{\text{od}},s\rangle \xrightarrow{\varepsilon} s}$$

The conditional rules mean that the behaviour of the conditional command is the same as the guarded command it contains, if this guarded command is open; otherwise it will result in abortion. The repetition rules say that if the guarded command is closed (rule 3) then the repetitive command terminates with no effect; if the guarded command aborts so does the repetitive command (rule 2); otherwise execute the guarded command followed by the repetitive command.

parallel 1. $$\frac{\langle C_1, s\rangle \xrightarrow{\lambda} \langle C_1', s'\rangle \mid s' \mid \underline{\text{abortion}}}{\langle C_1 \| C_2, s\rangle \xrightarrow{\lambda} \langle C_1' \| C_2, s'\rangle \mid \langle C_2, s'\rangle \mid \underline{\text{abortion}}}$$

2. $$\frac{\langle C_2, s\rangle \xrightarrow{\lambda} \langle C_2', s'\rangle \mid s' \mid \underline{\text{abortion}}}{\langle C_1 \| C_2, s\rangle \xrightarrow{\lambda} \langle C_1 \| C_2', s'\rangle \mid \langle C_1, s'\rangle \mid \underline{\text{abortion}}}$$

3. $$\frac{\langle C_1, s\rangle \xrightarrow{\lambda} \langle C_1', s'\rangle,\ \langle C_2, s\rangle \xrightarrow{\bar{\lambda}} \langle C_2', s\rangle}{\langle C_1 \| C_2, s\rangle \xrightarrow{\varepsilon} \langle C_1' \| C_2', s'\rangle,\ \langle C_2 \| C_1, s\rangle \xrightarrow{\varepsilon} \langle C_1' \| C_2', s'\rangle}$$

Rules 1 and 2 mean that the execution of a parallel command $C_1 \| C_2$ is achieved by <u>interleaving</u> its constituents and the parallel command terminates <u>normally</u> only if and when all its constituents have terminated normally; if one of the constituents aborts the parallel command aborts.

Rule 3 means that communication between the constituents of the parallel command is similar to CCS and is achieved by the simultaneous occurrence of complementary input and output actions.

L-process $$\frac{\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle \mid s' \mid \underline{\text{abortion}},\ \phi_R(\lambda)\downarrow}{\langle R::C, s\rangle \xrightarrow{\phi_R(\lambda)} \langle R::C', s'\rangle \mid s' \mid \underline{\text{abortion}}}$$

where $\phi_R : \Lambda_c \to \Lambda_c$ is the partial function defined by

$$\phi_R(\lambda) = \begin{cases} \varepsilon & \text{if } \lambda = \varepsilon \\ (R,P)?W(v) & \text{if } \lambda = (*,P)?W(v) \quad R \neq P \\ (R,Q)!W(v) & \text{if } \lambda = (*,Q)!W(v) \quad R \neq Q \end{cases}$$

where $\phi(\lambda)_R\downarrow$ means $\phi_R(\lambda)$ is defined.

The L-process rule defines the transitions for process bodies. The function $\phi_R(\lambda)$ models the requirement that a process label

serves as a name for the process body; the case $\lambda=\varepsilon$ means that if a transition action is internal (without any interaction with outside world) when viewed from within the body, then it is internal when viewed from outside the body as well. The case $\lambda=(*,P)?W(v)$ indicates that if a unknown agent N receives value v with pattern W from an object P, when viewed from within the body, then when viewed from outside the body (R::C) this action is that the agent R receives value v with pattern W from object (to object) P; the condition $R\neq P$ means that a process which tries to communicates with itself will deadlock. The case $\lambda=(*,Q)!W(v)$ is similar.

$$\underline{\text{D-process}}\quad \frac{\langle C,s\rangle \xrightarrow{\lambda} \langle C',s'\rangle \mid s' \mid \underline{\text{abortion}},\ \eta_{R,FPL(C)}(\lambda)\downarrow}{\langle \underline{\text{process}}\ R;C,s\rangle \xrightarrow{\eta_{R,FPL(C)}(\lambda)} \langle \underline{\text{process}}\ R;C',s'\rangle \mid s' \mid \underline{\text{abortion}}}$$

where $\eta_{R,L}:\Lambda_c\rightarrow\Lambda_c$ is the partial function defined by

$$\eta_{R,L}(\lambda)=\begin{cases}\lambda & \text{if } \lambda=\varepsilon\\ \lambda & \text{if } \lambda=(N,P)?W(v) \text{ and } R\neq N,\ R\neq P\\ \lambda & \text{if } \lambda=(N,Q)!W(v) \text{ and } R\neq N,\ R\neq Q\\ (*,P)?W(v) & \text{if } \lambda=(R,P)?W(v) \text{ and } P\notin L\\ (*,Q)!W(v) & \text{if } \lambda=(R,Q)!W(v) \text{ and } Q\notin L\end{cases}$$

where $N\in Plab\cup\{*\}$ and $\eta_{R,L}(\lambda)\downarrow$ means $\eta_{R,L}(\lambda)$ is defined.

The D-process rule defines the transitions for process declarations and the intention of $\eta_{R,L}(\lambda)$ is to implement the scope rules. To see the meaning of the rule we analyse the case $\lambda=(N,P)?W(v)$. There are two possibilities for the name N:

1. $N\neq *$. Then according to the L-process rule we know that there is a process body immediately occurring in process R. There are three subcases to analyse:

a. N=R. Then $\lambda=(R,P)?W(v)$ and this means that agent R receives value v with pattern W from object P. Since the command process R;C defines the scope of R, from outside of the command process R;C this action must be $(*,P)?W(v)$, i.e, a unknown agent receives the value v with pattern W from object P if P is not a free agent of C ($P\notin FPL(C)$). That is the fourth case of the definition:

$$\eta_{R,FPL(C)}((R,P)?W(v))=(*,P)?W(v) \quad \text{if } P\notin FPL(C)$$

b. $N\neq R$ and R=P. Then $\lambda=(N,R)?W(v)$, it means that agent N receives the value v from object R. Noticing process R;C defines the scope of R so the R in the action $(N,R)?W(v)$ must refers the R declared in process R;C, thus $\eta_{R,FPL(C)}(\lambda)$ must be undefined in this case.

c. $N\neq R$ and $P\neq R$. Then $\lambda=(N,P)?W(v)$ means that agent N receives the value v with pattern W from object P. Since process R;C only concerns with the scope of R and is nothing to do with N and P. Therefore in this case

$$\eta_{R}((N,P)?W(v))=(N,P)?W(v)$$

This is the second case of the definition.

2. N=*. Then there can exist only two subcases:

a. An input command P?W(x) occurs immediately in process R;C (see the input rule).

b. A command process R;C occurs immediately in process R;C (see the fourth case of the definition).

In both subcases if P=R then the situation is similar to the above subcase (b) and then $\eta_{R,FPL(C)}$ must be undefined, otherwise the actions are nothing to do with R, so $\eta_{R,FPL(C)}(\lambda)=\lambda$. These two subcases are contained in the second case of the definition. For the case $\lambda=(N,Q)!W(v)$ the analysis is similar.

In the next section we will study some examples and see how these L-process and D-process rules work.

2.4 Properties and examples

When looking at the rules given in the previous section, we might be concerned about the possibility that certain clauses had been overlooked or that some of them may be contradictory. The following results may help to convince us that this is not the case:

Lemma 2.1

Suppose $r, r' \varepsilon \Gamma_c \cup \Gamma_{gc}$, $\lambda \varepsilon \Lambda_c$ and $s \varepsilon$ States.

1. if $r \xrightarrow{\lambda} s$ then $\lambda = \varepsilon$ and s equals the state in r.

2. if $r \xrightarrow{\lambda} \underline{\text{abortion}}$ then $\lambda = \varepsilon$.

3. if $r \xrightarrow{\lambda} r'$ and λ has the form $(N, Q)!W(v)$ where $N \varepsilon Plab \cup \{*\}$ then r and r' have the forms $\langle \Omega, s \rangle$ and $\langle C', s \rangle$ respectively, and
$\langle \Omega, s \rangle \xrightarrow{(N,Q)!W(v)} \langle C', s \rangle$.

4. if $r \xrightarrow{\lambda} r'$ and λ has the form $(N, Q)?W(v)$ then r and r' have the forms $\langle \Omega, s \rangle$ and $\langle C', s[v/x] \rangle$ respectively, and for any value v'
$\langle \Omega, s \rangle \xrightarrow{(N,Q)?W(v')} \langle C', s[v'/x] \rangle$.

proof. All of these assertions can be easily proved by structural induction on r. Let us prove the fourth one. Since r cannot be in T_c, let $r = \langle \Omega, s \rangle$ where $\Omega \varepsilon Syn$. The proof is by structural induction on Ω. It is obvious that Ω cannot be one of the form: skip, abort, $x := e$, $P!W(e)$. We consider the other cases one by one.

case 1. Ω is $Q?W(x)$. Take $C' = \underline{\text{skip}}$, $s' = s[v'/x]$ the result is immediate.

case 2. Ω is $b \Rightarrow C$. Then according to the guards rule $[\![b]\!]_s = tt$ and for any v'

$\langle C,s\rangle \xrightarrow{(N,Q)?W(v')} r'$ and $\langle b \Rightarrow C,s\rangle \xrightarrow{(N,Q)?W(v')} r'$.

By the induction hypothesis we have $r'=\langle C',s[v'/x]\rangle$ for some $C'\varepsilon Com$ and x.

case 3. Ω is GC1[]GC2. Then according to the alternative rule we have for some i i=1,2

$$\langle GCi,s\rangle \xrightarrow{(N,Q)?W(v')} r'$$

By the induction hypothesis r' has the form $\langle C',[v'/x]\rangle$.

case 4. Ω is $C_1;C_2$. Then according to the composition rule r' can be s', abortion, C_2 and $C_1';C_2$. By clause 1 to 3 of this lemma r' only can be the last and:

$$\langle C_1;C_2,s\rangle \xrightarrow{\lambda} \langle C_1';C_2,s'\rangle \text{ and } \langle C_1,s\rangle \xrightarrow{\lambda} \langle C_1',s'\rangle$$

By the induction hypothesis on C_1 we have $s'=s[v'/x]$

for some x and and any v' $\langle C_1,s\rangle \xrightarrow{(N,Q)?W(v')} \langle C_1',s[v'/x]\rangle$

so $\langle C_1;C_2,s\rangle \xrightarrow{(N,Q)?W(v')} \langle C_1';C_2,s'\rangle$.

case 5. Ω is do GC od. Then according to the repetition rule r' can be s', abortion or $\langle C,s'\rangle$. Then by clause 1 to 3 of this lemma we have $[\![Bool(GC)]\!]_s=tt$ and

$$\langle \underline{do}\ GC\ \underline{od},s\rangle \xrightarrow{\lambda} \langle C;\underline{do}\ GC\ \underline{od},s'\rangle \text{ and } \langle GC,s\rangle \xrightarrow{\lambda} \langle C,s'\rangle$$

By the induction on GC we have for some x and any v'

$$\langle GC,s\rangle \xrightarrow{(N,Q)?W(v')} \langle C,s[v'/x]\rangle \text{ so}$$

$$\langle \underline{do}\ GC\ \underline{od},s\rangle \xrightarrow{(N,Q)?W(v')} \langle C;\underline{do}\ GC\ \underline{od},s[v'/x]\rangle$$

case 6. Ω is if GC fi. The proof is similar to case 5.

case 7. Ω is $C_1 \| C_2$. The by the parallel rule we have $\langle C_i,s\rangle \xrightarrow{\lambda} r'$ i=1 or 2. For example, let i=1 then by the induction hypothesis

$$\langle C_1,s\rangle \xrightarrow{(N,Q)?W(v')} \langle C_1',s[v'/x]\rangle$$

for some x and any v'. Thus take $r'=\langle C_1' \| C_2,s[v'/x]\rangle$ the result is proved.

case 8. Ω is R::C. Then by L-process rule we have N=R and for some N'

$$\langle C,s\rangle \xrightarrow{(N',Q)?W(v)} \langle C',s'\rangle \text{ and } \phi_R((N',Q)?W(v))=(R,Q)?W(v)$$

By the induction hypothesis we have for some x and any v'

$$\langle C,s\rangle \xrightarrow{(N,Q)?W(v')} \langle C',s[v'/x]\rangle$$

then take $r'=\langle R::C',s[v'/x]\rangle$ the result is proved.

case 9. Ω is process R;C. The proof is similar to case 8. □

Cases 1 and 2 of this lemma mean that only an internal action can lead a command (or guarded command) to a normal or abnormal termination directly in one transition. Case 3 means that an output action cannot change the state and case 4 means an input action can change only one variable of the state and any change of that variable is possible. The following result is derived immediately from the above lemma:

Corollary 1

If C is $C_1 \| C_2$ and

$$\langle C_1,s\rangle \xrightarrow{\lambda} \langle C_1',s'\rangle, \quad \langle C_2,s\rangle \xrightarrow{\bar{\lambda}} \langle C_2',s''\rangle$$

then one of s' and s'' must equals s.

This indicates that the parallel rule 3 is properly defined. The following fact will show some of the interaction between the static and dynamic semantics:

Lemma 2.2

Let $\Omega \in Syn$ be a guarded command or a command. If $\langle \Omega,s\rangle \xrightarrow{\lambda} \langle \Omega',s'\rangle$ then $\Omega' \in Com$ and

$$RV(\Omega') \subseteq RV(\Omega),\ WV(\Omega') \subseteq WV(\Omega),$$

$$FV(\Omega') \subseteq FV(\Omega) \text{ and } FPL(\Omega') \subseteq FPL(\Omega)$$

Proof. The proof is still by structural induction. Let us prove the result for RV only examining a few interesting cases:

case 1. Ω is P?W(x). Then Ω' must be skip and the result is immediate.

case 2. Ω is b⇒C. Then Ω' only can be $C' \in Com$ and

$\langle C,s\rangle \xrightarrow{\lambda} \langle C',s'\rangle$. By the induction hypothesis $RV(C') \subseteq RV(C)$ therefore

$$RV(C') \subseteq RV(C) \subseteq FV(b) \cup RV(C) = RV(\Omega)$$

case 3. Ω is $C_1;C_2$. Then according to the composition rule Ω' can be $C_1';C_2$ or C_2. For the first case by the induction hypothesis $RV(C_1') \subseteq RV(C_1)$ therefore

$$RV(C_1';C_2) = RV(C_1') \cup RV(C_2) \subseteq RV(C_1) \cup RV(C_2) = RV(C_1;C_2)$$

For the second case the proof is similar.

case 4. Ω is do GC od. Then Ω' only can be

$C;$do GC od and $\langle GC,s\rangle \xrightarrow{\lambda} \langle C,s'\rangle$

or do GC od and $\langle GC,s\rangle \xrightarrow{\lambda} s'$

For the first case by the induction hypothesis $RV(C) \subseteq RV(GC)$ thus

$$RV(C;\text{do GC od}) = RV(C) \cup RV(\text{do GC od}) \subseteq RV(GC) = RV(\text{do GC od})$$

The proof of the second case is similar. □

Using this lemma we can easily prove the following major result which is that no static error can arise during computation (execution).

Theorem 2.1

If $\langle \Omega,s\rangle \xrightarrow{\lambda} r$ and $\vdash \Omega$ then either $r=\langle C,s'\rangle$ and $\vdash_c C$ or r is one of the forms abortion, $s' \in States$.

Proof The proof is by structural induction on Ω.

case 1. Ω is of one of the forms: skip, abort, x:=e, P?W(x) and Q!W(e). According to the corresponding axioms the result is immediate.

case 2. Ω is $b \Rightarrow C$. As $\vdash_{gc} GC$ we have $\vdash_c C$. Then if $[\![b]\!]_s$=error then r' is abortion; otherwise r' can be s', abortion or $\langle C',s'\rangle$ and $\langle C,s\rangle \xrightarrow{\lambda} \langle C',s'\rangle$. For the last case by the induction hypothesis we have $\vdash_c C'$, since $\vdash_c C$.

case 3. Ω is $GC_1 [] GC_2$. Then by the guards rule the transition of

$\langle GC_1 [] GC_2, s\rangle$ is the same as $\langle GC_i, s\rangle$ i=1 or 2. By the induction hypothesis the result is true since $\vdash_{gc}$ GC1[]GC2 implies $\vdash_{gc}$ GCi.

case 4. Ω is $C_1;C_2$. Then $\vdash_c C_1$, $\vdash_c C_2$ and r' can only be $\langle C_1';C_2, s'\rangle$, $\langle C_2, s\rangle$ or abortion. Consider the first case. By the induction hypothesis $\vdash_c C_1'$ then according to the static rules we have $\vdash_c C_1';C_2$. The proof of the second case is similar.

case 5. Ω is if GC fi. Then $\vdash_{gc}$ GC and if $[\![Bool(GC)]\!]_s$=ff r' is abortion and if $[\![Bool(GC)]\!]_s$=tt then r' can be $\langle C, s'\rangle$, s' or abortion. For the first case by the induction hypothesis we have $\vdash_c C$.

case 6. Ω is do GC od. The proof is similar to case 5.

case 7. Ω is $C_1 \| C_2$. Then $\vdash C_1$, $\vdash C_2$ and

$$FV(C_1) \cap WV(C_2) = FV(C_2) \cap WV(C_1) = \emptyset \text{ and } FPL(C_1) \cap FPL(C_2) = \emptyset$$

According to the parallel rules r can be of one of the forms: $\langle C_1' \| C_2, s'\rangle$, $\langle C_1 \| C_2', s'\rangle$, $\langle C_1' \| C_2', s'\rangle$, $\langle C_1, s\rangle$, $\langle C_2, s\rangle$ or abortion. For the last three cases the result is immediate. Let us check the first three cases:

a. r' is $\langle C_1' \| C_2, s'\rangle$. Then by the parallel rule $\langle C_1, s\rangle \xrightarrow{\lambda} \langle C_1', s'\rangle$. By the induction hypothesis $\vdash_c C_1'$ and by the lemma 2.2 we have:

$$RV(C_1') \subseteq RV(C_1),\ WV(C_1') \subseteq WV(C_1) \text{ and } FPL(C_1') \subseteq FPL(C_1)$$

Therefore

$$FV(C_1') \cap WV(C_2) \subseteq FV(C_1) \cap WV(C_2) = \emptyset$$
$$FV(C_2) \cap WV(C_1') \subseteq FV(C_2) \cap WV(C_1) = \emptyset$$
$$FPL(C_1') \cap FPL(C_2) \subseteq FPL(C_1) \cap FPL(C_2) = \emptyset$$

Thus by the static rule 11 $\vdash_c C_1';C_2$.

b. r is $\langle C_1 \| C_2', s'\rangle$. The proof is similar to subcase (a).

c. r' is $\langle C_1' \| C_2', s' \rangle$, Then for i,j=1,2 and i≠j

$\langle C_i, s \rangle \xrightarrow{\lambda} \langle C_i', s \rangle$ and $\langle C_j, s \rangle \xrightarrow{\bar{\lambda}} \langle C_j', s' \rangle$

By the induction hypothesis $\vdash_c C_i'$ i=1,2. By lemma 2.2 we have:

$FV(C_i') \subseteq FV(C_i)$, $WV(C_i') \subseteq WV(C_i)$ and $FPL(C_i') \subseteq FPL(C_i)$

where i=1 or 2. Thus

$FV(C_i') \cap WV(C_j') \subseteq FV(C_i) \cap WV(C_j) = \emptyset$

$FPL(C_1') \cap FPL(C_2') \subseteq FPL(C_1) \cap FPL(C_2) = \emptyset$

where i,j=1,2 and i≠j. By the static rule 11 we have $\vdash_c C_1' \| C_2'$.

case 8. Ω is R::C. Then r' can only be s' , abortion or $\langle R::C', s' \rangle$ and $\langle C, s \rangle \xrightarrow{\lambda'} \langle C', s' \rangle$ where $\phi_R(\lambda') = \lambda$. By the induction hypothesis on C we have $\vdash_c C'$; according to lemma 2.2

$FPL(C') \subseteq FPL(C) = \emptyset$

therefore $\vdash_c R::C'$.

case 9. Ω is process R;C. The proof is similar to case 8.

The theorem has been proved. □

Similarly to FPL(C) given in section 2.2, we can define FPO(Ω) the set of possible objects with which the syntactic entity Ω can communicate. The definition of FPO(Ω) is just the same as FPL(Ω) except that

$FPO(P?W(x)) = FPO(P!W(e)) = \{P\}$

$FPO(P::C) = FPO(C)$

$FPO(\text{process } P;C) = FPO(C) \setminus \{P\}$

Then the following result can be proved:

Lemma 2.3

Let $\langle \Omega, s \rangle \xrightarrow{\lambda} r$, P, R ε Plab and N ε Plab∪{*}.

1. If λ is one of (R,P)?W(v) or (R,P)!W(v) then R∈FPL(Ω).

2. if λ is one of $(N,R)?W(v)$ or $(N,R)!W(v)$ then $R \varepsilon FPO(\Omega)$.

3. if r is $\langle C',s' \rangle$ then $FPO(C') \subseteq FPO(C)$.

Proof. By structural induction on Ω. Let us check the case of the second assertion where $\lambda=(N,R)?W(v)$. Then Ω cannot be skip, abort, x:=e, or P!W(e). For the rest of the cases we only examine the following interesting ones:

case 1. Ω is $R?W(x)$. Then the result is immediate since $FPO(R?W(x))=\{R\}$.

case 2. Ω is $C_1;C_2$. Then by the composition rule and lemma 2.2 we only have one transition:

$$\langle C_1,s \rangle \xrightarrow{(N,R)?W(v)} \langle C_1',s' \rangle$$

By the induction hypothesis $R \varepsilon FPO(C_1)$ thus

$$R \varepsilon FPO(C_1) \subseteq FPO(C_1) \cup FPO(C_2)=FPO(C_1;C_2)$$

case 3. Ω is do GC od. Then according to the repetition rule and lemma 2.2 $[\![Bool(GC)]\!]_s=tt$ and

$$\langle \underline{do}\ GC\ \underline{od},s \rangle \xrightarrow{\lambda} \langle C;\underline{do}\ GC\ \underline{od},s' \rangle \text{ and } \langle GC,s \rangle \xrightarrow{\lambda} \langle C,s' \rangle$$

By the induction hypothesis $R \varepsilon FPO(GC)$ so $R \varepsilon FPO(GC)=FPO(\underline{do}\ GC\ \underline{od})$.

case 4. Ω is P::C. Then according to the L-process rule and lemma 2.1 and 2.2

$$\langle C,s \rangle \xrightarrow{(N',R)?W(v)} \langle C',s' \rangle \text{ and } P \neq R \text{ and } \phi_P(N',P)?W(v))=(P,R)?W(v)$$

By the induction hypothesis $R \varepsilon FPO(C)$ so

$$R \varepsilon FPO(C)=FPO(P::C).$$

case 5. Ω is process P;C. Then according to the D-process rule and lemma 2.2

$$\langle C,s \rangle \xrightarrow{(N',R)?W(v)} \langle C',s' \rangle \text{ and } P \neq R$$

By the induction hypothesis $R \varepsilon FPO(C)$ so

$$R \varepsilon FPO(C) \setminus \{P\}=FPO(\Omega). \quad \square$$

The meaning of the lemma is obvious, for example, the case $\lambda=(R,P)?W(v)$ means that if $\langle \Omega,s \rangle$ can be derived to r by an input

action (R,P)?W(v) then the name of the source P must occur in Ω as a possible process with which Ω can communicate (i.e, the command P?W(x) occurs in Ω), and the label R must also occur in Ω as a free process label (i.e, the process body R::C' occurs in Ω). In other words any actual process which is the object or agent of a communication in the execution of C must also be a possible one in that it is in FPL(Ω) or FPO(Ω).

The rest of this section is devoted to the study of some examples in order to demonstrate how the semantics works.

Example 2.3 One character buffer

Let BUFF be the command B::C, where C is

$$\underline{\text{do}}\ (b \Rightarrow IN?CH(x);OUT!CH(x)\ \square\ b \Rightarrow IN?STOP(\ );b:=\underline{\text{false}})\ \underline{\text{od}}$$

C acts as a one character buffer between processes IN and OUT until sent a stop signal by IN. Let the initial state be (b=tt,x=m) denoted by s, and let the number i under an arrow $\xrightarrow[i]{\lambda}$ indicate i^{th}-transition step in a transition sequence. According to the semantics given in the last section there are many ways to execute this program. Let us consider one of them :

$$\begin{aligned}
\langle B::C,s\rangle &\xrightarrow[1]{(B,IN)?CH(n)} \langle B::\underline{\text{skip}};OUT!CH(x);C,(b=tt,x=n)\rangle \\
&\xrightarrow[2]{\varepsilon} \langle B::OUT!CH(x);C,(b=tt,x=n)\rangle \\
&\xrightarrow[3]{(B,OUT)!CH(n)} \langle B::\underline{\text{skip}};C,(b=tt,x=n)\rangle \\
&\xrightarrow[4]{(B,IN)?STOP(\ )} \langle B::b:=\underline{\text{false}};C,(b=tt,x=n)\rangle \\
&\xrightarrow[5]{\varepsilon} \langle B::C,(b=ff,x=n)\rangle \\
&\xrightarrow[6]{\varepsilon} (b=ff,x=n)
\end{aligned}$$

It is not difficult to determine the correctness of these transition steps. For example, step 1 is obtained by the L-process rule since

the following transition:

$\langle C,s\rangle \xrightarrow{(*,IN)?CH(n)} \langle OUT!CH(x);C,(b=tt,x=n)\rangle$

is true by the repetition rule since

$\langle b \Rightarrow IN?CH(x);OUT!CH(x)\ \square\ b \Rightarrow IN?STOP(\ );b:=\underline{false},s\rangle$

$\xrightarrow{(*,IN)?CH(n)} \langle OUT!CH(x),(b=tt,x=n)\rangle$

by the guard alternative rule since $[\![b]\!]_s=tt$ and

$\langle IN?CH(x);OUT!CH(x),s\rangle \xrightarrow{(*,IN)?CH(n)} \langle \underline{skip};OUT!CH(x),(b=tt,x=n)\rangle$

by the composition rule since

$\langle IN?CH(x),s\rangle \xrightarrow{(*,IN)?CH(n)} \langle \underline{skip},(b=tt,x=n)\rangle$

by the input rule. □

The following series of examples shows how the L-process and D-process rules manage communication between processes (possibly nested)

Example 2.4 Communication

Consider the following CSP programs:

$\underline{process}\ P;(\underline{process}\ Q;(P::Q!W(x)\ \|\ Q::P?W(y)))$

The scopes of both P and Q are the whole parallel command. According to the parallel rule 3, they can communicate with each other and a computation sequence of the parallel command in the state (x=n,y=m) is:

$\langle P::Q!W(x)\|Q::P?W(y),(x=n,y=m)\rangle$

$\xrightarrow{\varepsilon} \langle P::\underline{skip}\|Q::\underline{skip},(x=n,y=n)\rangle$

$\xrightarrow{\varepsilon}$ <P::skip,(x=n,y=n)>

$\xrightarrow{\varepsilon}$ (x=n,y=n)

The first transition step is obtained by parallel 2 since the following two transition relations hold:

a. $\langle P::Q!W(x),(x=n,y=m)\rangle \xrightarrow{(P,Q)!W(n)} \langle P::\underline{skip},(x=n,y=m)\rangle$

b. $\langle Q::P?W(y),(x=n,y=m)\rangle \xrightarrow{(Q,P)?W(n)} \langle Q::\underline{skip},(x=n,y=n)\rangle$

Transition (a) is derived from the L-process rule since

$\langle Q!W(x),(x=n,y=m)\rangle \xrightarrow{(*,Q)!W(n)} \langle \underline{skip},(x=n,y=m)\rangle$

by the output rule 1. Transition (b) is derived using the L-process and input rules.

We sometimes use the notation $\xrightarrow[\lambda]{C}$ to indicate that the command C (in some state s) can evolve under transition action λ and dotted line ------ to indicate no further transition action. Thus the communication situation of command C1 can be written as

$$\underbrace{\underbrace{P::\underbrace{Q\ !\ W(\ x\ )}_{(Q,*)!W(v)}}_{(Q,P)!W(v)} \parallel \underbrace{Q::\underbrace{P\ ?\ W(\ y\ )}_{(*,P)?W(v)}}_{(Q,P)?W(v)}}_{\varepsilon}$$

Example 2.5 Communication of a process with itself leads to deadlock

Consider the following two commands

command C1 command C2

P:: P ? W(x) P:: P ! W(y)

(*,P)?W(v) (*,P)!W(v)

According to the L-process rule (note the conditions) neither P::P?W(x) nor P::P!W(y) can move further (from any state).

Similarly, for the following commands:

command C3 command C4

process P;R:: P?W(x) process P;R:: P!W(y)

(R,P)?W(v) (R,P)!W(v)

According to the D-process rule (note the conditions in the second and the third cases) no further evolution is possible in either case.

The next three examples concern communication between nested processes with or without overlapping, and let $[\![x]\!]_s$=v.

Example 2.6 Nested communication between nested processes

Consider the following command

(process P;(P::C1‖ R::Q ? W(y))) ‖ Q::R ! W(x)

(*,Q)?W(v) (*,R)!W(v)

(R,Q)?W(v)

(R,Q)?W(v)

(R,Q)?W(v) (Q,R)!W(v)

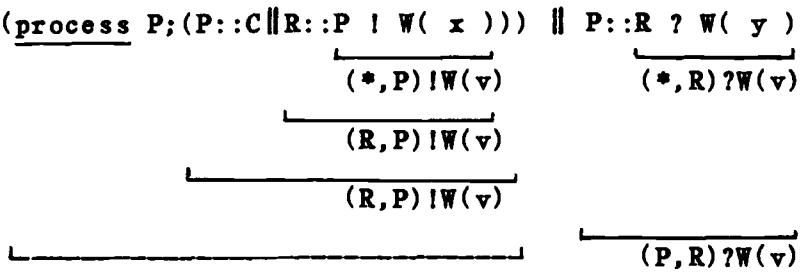
ε

R and Q are in the same declaration area and each is visible to the

other, and the above example shows that our semantics does allow communication between them. □

Example 2.7 Local process labels

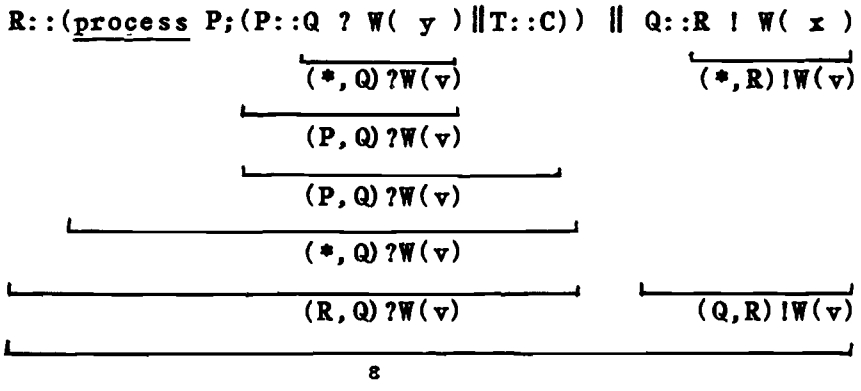
Consider the following command:



The scope of the outer P (the rightmost P) has a hole in the inner parallel command. According to D-process rule $\eta_{P,L}((R,P)!W(v))$ (L=FPL(P::C‖R::P!W(x)) is undefined , so it cannot get out of the inner parallel command, i.e, the reference to P by a transition action of R::P!W(x) refers to the innermost P.

Example 2.8 Communication between nested processes

Consider the following command:



The body of process P is in the scope of label Q, therefore Q is visible to the command Q?W(x). The label P is not visible to the body of process Q but the process R is. In this case communication

between processes P and Q is still possible but in an indirect manner. Even when R is replaced everywhere by P communication between P and Q still works.

2.5 Further discussion

The previous sections demonstrate that our approach of defining the operational semantics is structural: we construct the semantics in the following three steps:

1. Carefully select the set of configurations Γ and the set of transition actions Λ, in order to describe all phenomena of computation in which we are interested. Here they are intermediates, states and abortion.

2. Define the semantics (transitions) for non-structured commands using axioms. Here they are skip, abort, assignment, the input and output commands.

3. Construct the rules for every structured command by enumerating and combining as needed all the possibilities of the transitions of their constituents. For example, let us check the composition commands C1;C2. Its transitions depend only on the transitions of C1, and for the given Γ_c and Λ_c either the configuration $\langle C1,s\rangle$ cannot move any more (deadlock) or can move by a transition action into one of the following three configurations: $\langle C_1',s'\rangle$, s' (normal termination) or abortion (abnormal termination). Thus the successors of $C_1;C_2$ are naturally $\langle C_1';C_2,s'\rangle$, $\langle C_2,s'\rangle$ or abortion. Examining the transition rules for composition, we see that all these possibilities have been taken into account.

The command $C_1 \| C_2$ can be transformed only in the following two ways:

a. It is transformed by interleaving one of its constituents C_i i=1 or 2. Consider the case i=1. Since C_1 can only be transformed to C_1', s' or abortion, $C_1 \| C_2$ can only be transformed to $C_1' \| C_2$, C_2 or abortion. This analysis gives the parallel rule 1. Similarly, for i=2 we obtain the parallel rule 2.

b. It is transformed by handshake communication. Then the transitions of C_1 and C_2 should be:

$$\langle C_i, s\rangle \xrightarrow{\lambda} \langle C_i', s'\rangle \text{ and } \langle C_j, s\rangle \xrightarrow{\bar{\lambda}} \langle C_j', s\rangle$$

where i,j=1,2 and $i \neq j$, then the transition of $C_1 \| C_2$ naturally is:

$$\langle C_1 \| C_2, s\rangle \xrightarrow{\varepsilon} \langle C_1' \| C_2', s'\rangle$$

These give the parallel rule 3. Therefore the parallel rules have taken all possibilities of transitions of $C_1 \| C_2$ into account.

In fact, given Γ and Λ the above step (2) and (3) provide an systematic approach to the construction of an operational semantics.

It should be stressed that the choice of the "world of computation" Γ, T and Λ (the sets of the configurations and transition actions) plays an important role. Changing these sets will lead to changes of semantics. For instance, introduce a set L of labels of living process into the configurations and define

$$\Gamma_{gc} = \{\langle GC, s, L\rangle \mid GC \varepsilon Gcm,\ L \supseteq FPL(GC),\ s \varepsilon States\} \cup \Gamma_c$$
$$\Gamma_c = \{\langle C, s, L\rangle \mid C \varepsilon Com,\ L \supseteq FPL(C),\ s \varepsilon States\} \cup T_c$$
$$T_{gc} = \Gamma_c$$
$$T_c = \{\langle s, L\rangle \mid s \varepsilon States,\ L \varepsilon Plab\} \cup \{\underline{abortion}\} \cup \{\underline{failure}\}$$

Here the configuration failure is introduced to model the requirement that input and output commands should fail if the target process has already terminated (see 2.4 [Hoare 78]). Then the corresponding transition relations should be modified as follows:

Guarded commands

guards

$$1.\ \frac{\langle C,s,L\rangle \xrightarrow{\lambda} r,\ [\![b]\!]_s = tt}{\langle b \Rightarrow C,s,L\rangle \xrightarrow{\lambda} r}$$

$$2.\ \frac{[\![b]\!]_s = \underline{error}}{\langle b \Rightarrow C,s,L\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

Alternative

$$1.\ \frac{\langle GC_1,s,L\rangle \xrightarrow{\lambda} \langle C_1,s',L'\rangle \mid \langle s',L'\rangle \mid \underline{abortion}}{\langle GC_1 [] GC_2,s,L\rangle \xrightarrow{\lambda} \langle C_1,s',L'\rangle \mid \langle s',L'\rangle \mid \underline{abortion}}$$

$$2.\ \frac{\langle GC_2,s,L\rangle \xrightarrow{\lambda} \langle C_2,s',L'\rangle \mid \langle s',L'\rangle \mid \underline{abortion}}{\langle GC_1 [] GC_2,s,L\rangle \xrightarrow{\lambda} \langle C_2,s',L'\rangle \mid \langle s',L'\rangle \mid \underline{abortion}}$$

$$3.\ \frac{\langle GC_i,s,L\rangle \xrightarrow{\lambda} \underline{failure},\ i=1,2}{\langle GC_1 [] GC_2,s,L\rangle \xrightarrow{\lambda} \underline{failure}}$$

Commands

assign, skip and abort are similar to the previous rules but adding L to the configurations.

input

$$1.\ \frac{P \in L}{\langle P?W(x),s,L\rangle \xrightarrow{(*,P)?W(v)} \langle \underline{skip},s[v/x],L\rangle}$$

$$2.\ \frac{P \notin L}{\langle P?W(x),s,L\rangle \xrightarrow{\varepsilon} \underline{failure}}$$

output

1. $$\frac{[\![e]\!]_s = v,\ P \in L}{\langle P!W(e), s, L\rangle \xrightarrow{(*,P)!W(v)} \langle \underline{skip}, s, L\rangle}$$

2. $$\frac{[\![e]\!]_s = \underline{error}}{\langle P!W(e), s, L\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

3. $$\frac{[\![e]\!]_s \neq \underline{error},\ P \notin L}{\langle P!W(e), s, L\rangle \xrightarrow{\varepsilon} \underline{failure}}$$

The condition $P \notin L$ in the input and output rules means that the process P has already terminated.

composition

$$\frac{\langle C_1, s, L\rangle \xrightarrow{\lambda} \langle C_1', s', L'\rangle \mid \langle s', L'\rangle \mid \underline{abortion} \mid \underline{failure}}{\langle C_1; C_2, s, L\rangle \xrightarrow{\lambda} \langle C_1'; C_2, s', L'\rangle \mid \langle C_2, s', L'\rangle \mid \underline{abortion} \mid \underline{failure}}$$

conditional

1. $$\frac{\langle GC, s, L\rangle \xrightarrow{\lambda} \langle C, s', L'\rangle \mid \langle s, L'\rangle \mid \underline{failure} \mid \underline{abortion}}{\langle \underline{if}\ GC\ \underline{fi}, s, L\rangle \xrightarrow{\lambda} \langle C, s', L'\rangle \mid \langle s, L'\rangle \mid \underline{abortion} \mid \underline{abortion}}$$

2. $$\frac{[\![Bool(GC)]\!]_s = ff}{\langle \underline{if}\ GC\ \underline{fi}, s, L\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

repetition

1. $$\frac{\langle GC, s, L\rangle \xrightarrow{\lambda} \langle C, s', L'\rangle}{\langle \underline{do}\ GC\ \underline{od}, s, L\rangle \xrightarrow{\lambda} \langle C; \underline{do}\ GC\ \underline{od}, s', L'\rangle}$$

2. $$\frac{\langle GC, s, L\rangle \xrightarrow{\lambda} \langle s', L'\rangle \mid \underline{failure} \mid \underline{abortion}}{\langle \underline{do}\ GC\ \underline{od}, s, L\rangle \xrightarrow{\lambda} \langle \underline{do}\ GC\ \underline{od}, s', L'\rangle \mid \langle s, L\rangle \mid \underline{abortion}}$$

3. $$\frac{[\![Bool(GC)]\!]_s = ff}{\langle \underline{do}\ GC\ \underline{od}, s, L\rangle \xrightarrow{\varepsilon} \langle s, L\rangle}$$

The case of failure of the conditional rule 1 says that if the guarded command fails then the corresponding conditional statement aborts. The case failure of the repetition means that if the guarded

command fails then the repetitive command terminates with no effect.

parallel rules are similar to the previous rules.

L-process

$$1.\ \frac{\langle C,s,L\rangle \xrightarrow{\lambda} \langle C',s',L'\rangle \mid \langle s',L'\rangle}{\langle R::C,s,L\rangle \xrightarrow{\phi_R(\lambda)} \langle R::C',s',L'\rangle \mid \langle s',L'\backslash\{R\}\rangle}$$

$$2.\ \frac{\langle C,s,L\rangle \xrightarrow{\lambda} \underline{\text{failure}} \mid \underline{\text{abortion}}}{\langle R::C,s,L\rangle \xrightarrow{\phi_R(\lambda)} \underline{\text{failure}} \mid \underline{\text{abortion}}}$$

The function $\phi_R(\lambda)$ is defined the same as before, the second case of rule 1 means that the process can no longer be communicated with when execution reaches the end of its body.

D-process

$$1.\ \frac{\langle C,s,(L\backslash\{R\})\cup\{R\mid R\varepsilon K\}\rangle \xrightarrow{\lambda} \langle C',s',L'\rangle}{\langle \underline{\text{process}}\ \ R;C,s,L\rangle \xrightarrow{\eta_{R,K}(\lambda)} \langle \underline{\text{process}}\ R;C',s',L^*\rangle}$$

$$2.\ \frac{\langle C,s,(L\backslash\{R\})\cup\{R\mid R\varepsilon K\}\rangle \xrightarrow{\lambda} \langle s',L'\rangle \mid \underline{\text{failure}} \mid \underline{\text{abortion}}}{\langle \underline{\text{process}}\ \ R;C,s,L\rangle \xrightarrow{\eta_{R,K}(\lambda)} \langle s',L^*\rangle \mid \underline{\text{failure}} \mid \underline{\text{abortion}}}$$

Where $L^*=(L'\backslash\{R\})\cup\{R\mid R\varepsilon K\}$ and $K=FPL(C)$ and $\eta_{R,K}(\lambda)$ is defined the same as before. The meaning of this rule can be best understood through the following two examples:

Example 2.9 Communication between living processes

Consider the simple program below:

$$P1 = \underline{\text{process}}\ R;(R::P!W(x) \parallel P::R?W(y))$$

Let the initial s and L be (x=n,y=m) and {P} respectively. Then the computation is:

$$\langle P1,s,\{P\}\rangle \xrightarrow[1]{\varepsilon} \langle \underline{process}\ R;(R::\underline{skip} \| P::\underline{skip}),(x=n,y=n),\{P\}\rangle$$
$$\xrightarrow[2]{\varepsilon} \langle \underline{process}\ R;(R::\underline{skip}),(x=n,y=n),\emptyset\rangle$$
$$\xrightarrow[3]{\varepsilon} \langle (x=n,y=n),\emptyset\rangle$$

Transition step 1 is obtained by the D-process rule since the following transition relationship holds:

$$\langle R::P!W(x) \| P::R?W(y),s,\{P\}\cup\{R\}\rangle$$

$$\xrightarrow{\varepsilon} \langle R::\underline{skip} \| P::\underline{skip},(x=n,y=n),\{P,R\}\rangle$$

this is by the parallel rule 3 since

$$\langle R::P!W(x),s,\{P\}\cup\{R\}\rangle \xrightarrow{(R,P)!W(n)} \langle R::\underline{skip},(x=n,y=m),\{P,R\}\rangle$$

$$\langle P::R?W(y),s,\{P\}\cup\{R\}\rangle \xrightarrow{(P,R)?W(n)} \langle P::\underline{skip},(x=n,y=n),\{P,R\}\rangle$$

The first is by the L-process rule and output rule 1 since $P\varepsilon\{P,R\}$, and the second is true by the L-process rule and input rule 1 since $R\varepsilon\{P,R\}$. Thus in the case of normal communication the D-process rule works properly. □

Let us study the following example:

Example 2.10 Communication with a terminated process

Consider the program P2 below:

$$P2 = \underline{process}\ R;(R::P!W(x)) \ \|\ P::\underline{skip})$$

Let the initial s and L be (x=n), {P} respectively. A computation of P2 is:

$$\langle P2,s,\{P\}\rangle \xrightarrow[1]{\varepsilon} \langle \underline{process}\ R;(R::P!W(x)),s,\emptyset\rangle$$
$$\xrightarrow[2]{\varepsilon} \underline{failure}$$

Step 1 is true by parallel rule 2 since the transition relation

$\langle P::\underline{skip}, s, \{P\}\rangle \xrightarrow{\varepsilon} \langle s, \emptyset\rangle$

is true by the L-process rule. Step 2 follows from the D-process rule since

$\langle R::P!W(x), s, \{R\}\rangle \xrightarrow{\varepsilon} \underline{failure}$ by the L-process rule since

$\langle P!W(x), s, \{R\}\rangle \xrightarrow{\varepsilon} \underline{failure}$ since $P \notin \{R\}$. □

Finally, in section 2.1 we analysed the original form of parallel command [C1 ‖ .. ‖ Ci ‖.. ‖ Cn] given by Hoare and divided it into three syntactic entities: process R;C (process declaration), P::C (process body) and $C_1 \| C_2$ (parallel command). In fact, the transition rules can be defined directly for the original form without any difficulties:

parallel-B

$$1.\ \frac{\langle Ci, s\rangle \xrightarrow{\lambda} \langle C_i', s'\rangle \mid \underline{abortion},\ i=1..n}{\langle [..\|Ci\|..], s\rangle \xrightarrow{\lambda} \langle [..\|C_i'\|..], s'\rangle \mid \underline{abortion}}$$

$$2.\ \frac{\langle Ci, s\rangle \xrightarrow{\lambda} s',\ i=1..n}{\langle [..\|Ci\|..], s\rangle \xrightarrow{\lambda} \langle [..C_{i-1}\|C_{i+1}\|..], s'\rangle}$$

$$3.\ \frac{\langle Ci, s\rangle \xrightarrow{\lambda} \langle C_i', s'\rangle,\ \langle Cj, s\rangle \xrightarrow{\overline{\lambda}} \langle C_j', s\rangle,\ i\neq j}{\langle [..\|Ci\|..\|Cj\|..], s\rangle \xrightarrow{\varepsilon} \langle [..\|C_i'\|..\|C_j'\|..], s'\rangle}$$

The meaning of the above rules is the same as that of the parallel rules with the exception of the form. From the aesthetic point of view the separated form given in section 2.3 seems better than this.

3. An operational semantics for Ada multitasking and exception handling

Considerable interest has been generated within the computer science community by the problem of giving a semantics for the language Ada, especially that part of the semantics which is concerned with multitasking and exception handling (see [Bjórner and Oest 80], [Luckham and Polak 80], [Hennessy and Li 82], [Li 82]). The purpose of this chapter is to give an operational semantics for multitasking and exception handling in Ada using labelled transition systems. To focus our attention on these features, we first carefully select a small subset of Ada called Ada.1; this is essentially Dijkstra's guarded command language ([Dijkstra 76]) enriched by constructs concerned with multitasking. We then study how an operational semantics can be given for this language using transition systems. For exception handling we first study exceptions for the sequential case and then join this with multitasking and study the interaction between them.

It should be mentioned that an early version of the semantics of the rendezvous mechanism is published in [Hennessy and Li 82], and the remaining content of this chapter is an improved version of [Li 82].

In section 3.1 we review the multitasking and communication mechanisms and exception handling as given in the Ada manual (see [DoD 80]) and discuss some of the intuition behind the abstract syntax which we will use in later sections. In section 3.2 a static and dynamic semantics are given for Ada.1 which is concerned only with multitasking. Some properties are proved and some examples are studied to show that the semantics is consistent with the Ada manual. In section 3.3 we construct and study a small sequential

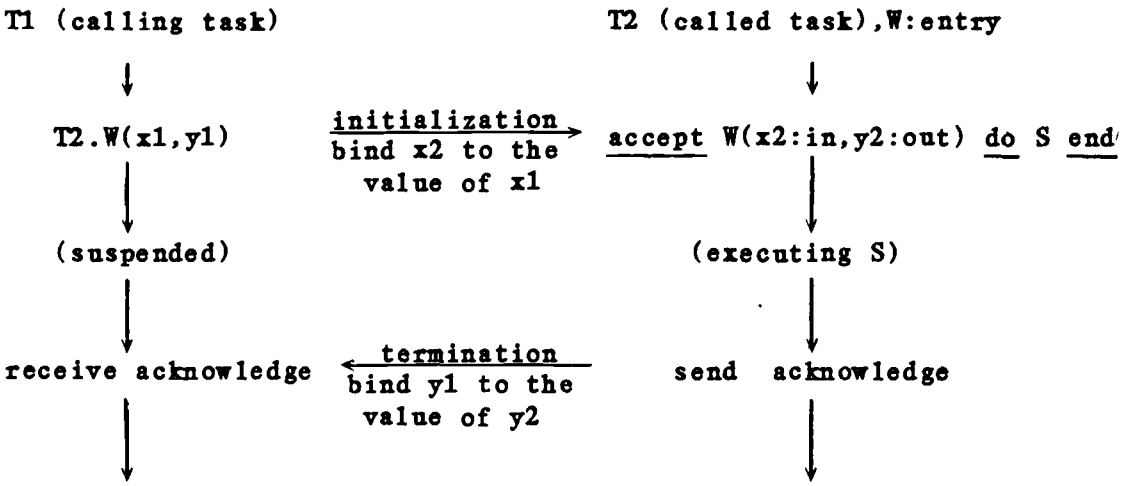
language Ada.2 which only contains exceptions for the sequential case. Finally, in section 3.4 putting Ada.1 and Ada.2 together we obtain a language Ada.3 and give a semantics for the interaction between exceptions and multitasking.

3.1 An outline of multitasking and exceptions in Ada

Before giving a formal description of Ada, it may be helpful to give a brief overview of multitasking and exceptions in Ada.

3.1.1 Multitasking in Ada

Tasks are the basic entities in Ada that may execute in parallel. A task must be declared before its execution; this declaration consists of a task specification and the corresponding task body. The specification gives the task name and entries. The task body contains the code which the task is to execute. A task (the calling task) wishing to communicate with another task (the called task) must issue an entry call statement, specifying an entry of the called task. The called task may use accept statements to respond to this entry call. Communication and synchronization between tasks is in general achieved by non-instantaneous rendezvous. The working process of a rendezvous is shown in the following diagram:



When both the calling task T1 and the called task T2 reach the entry call and the corresponding accept statement respectively the rendezvous begins and the value of the appropriate parameter of the caller, x1, is passed to the corresponding parameter, x2, of the called task. We call this the initialization phase. Then the calling task is suspended and the called task executes S the body of the accept statement. This is the second phase of the rendezvous. When the execution of S is finished the called task sends an acknowledgement accompanied by the value of the out parameter y2 back to the caller where this value is bound to y1 and the rendezvous is completed. We call this the termination phase. After the rendezvous both tasks again execute independently of each other.

It is easy to see that the rendezvous mechanism in Ada is a generalization of Hoare and Milner's handshake communication mechanism. Handshake communication is instantaneous, but in general, a rendezvous is not. It consists of two separate instantaneous handshake communications (initialization and termination) and a non-instantaneous period of the execution of the accept statement body S. This allows other entry call statements and accept statements to occur in an accept statement body, even for the same entry.

As with CSP, Ada also provides a select statement to cope with nondeterministic communication. The select statement has three different forms: select wait, conditional entry call and timed entry call. The forms and interpretation of the select statement are close to those of the guarded commands given in section 2.1 but with some elaborations such as termination, else and delay alternatives (see section 9.7 [DoD 80]). In this chapter we only deal with the simple cases as in CSP.

Finally, a task can call an entry of another task if the body of the first is in the scope of the second. Since the body of a task must appear after its specification, but need not follow it immediately , the scope of a task name and its entries depends on where the task specification occurs in the text and the calling feature depends on where the body of the calling task occurs relative to the specification of the called task. Let us examine the following two examples of nested task structure.

Example 3.1

The task T0 contains two subtasks: T1 and T2; and the task T1 contains another subtask T11. The bodies of the tasks T1 and T2 do not follow their specifications immediately (see the next page).

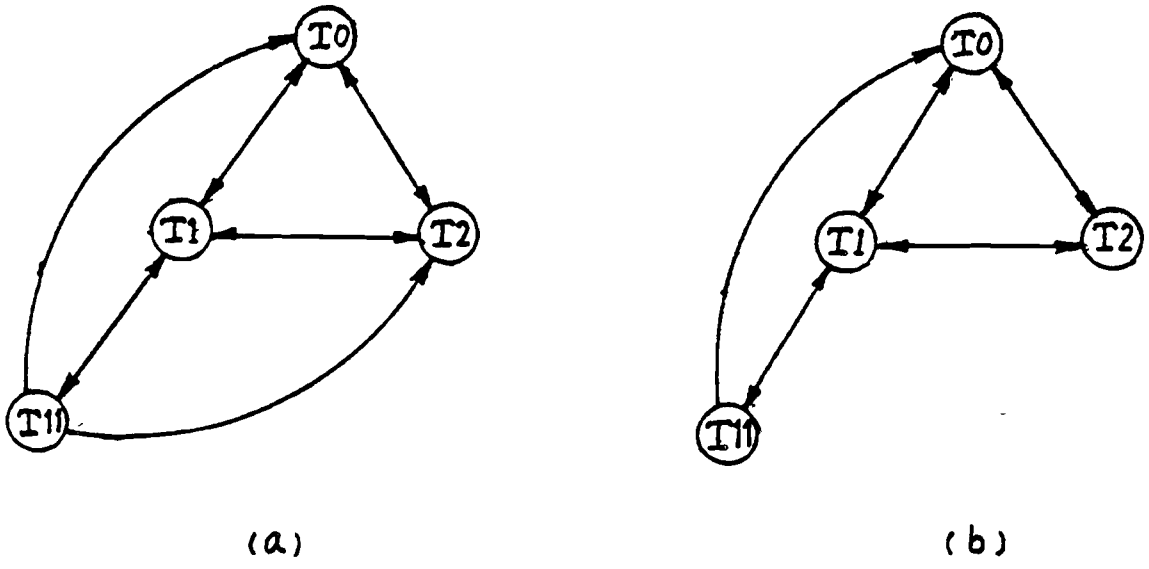


Figure 3.1 calling diagrams

```
task T0 is                                          scope of T0,E0
    entry E0;
end  T0;
task body T0 is
                                                    scope of T1,E1
        task T1 is
            entry E1;
        end  T1;
        task T2 is                                  scope of T2,E2
            entry E2;
        end  T2;
        task body T1 is
                task T11 is                         scope of T11,E11
                    entry E11;
                end  T11;
                task body T11 is
                begin
                 ---
                end T11;
         begin
          ---
         end T1;
         task body T2 is

         begin
          ---
         end T2;
begin
 ---
end T0;
```

According to the scope rules of Ada (see [DoD 80] section 8.2) the scopes of T1, E1 and T2, E2 extend from their specification points to the end of T0's body, and the scope of T11, E11 extends from its specification point to the end of T1's body. The calling diagram is shown in figure 3.1.a, where

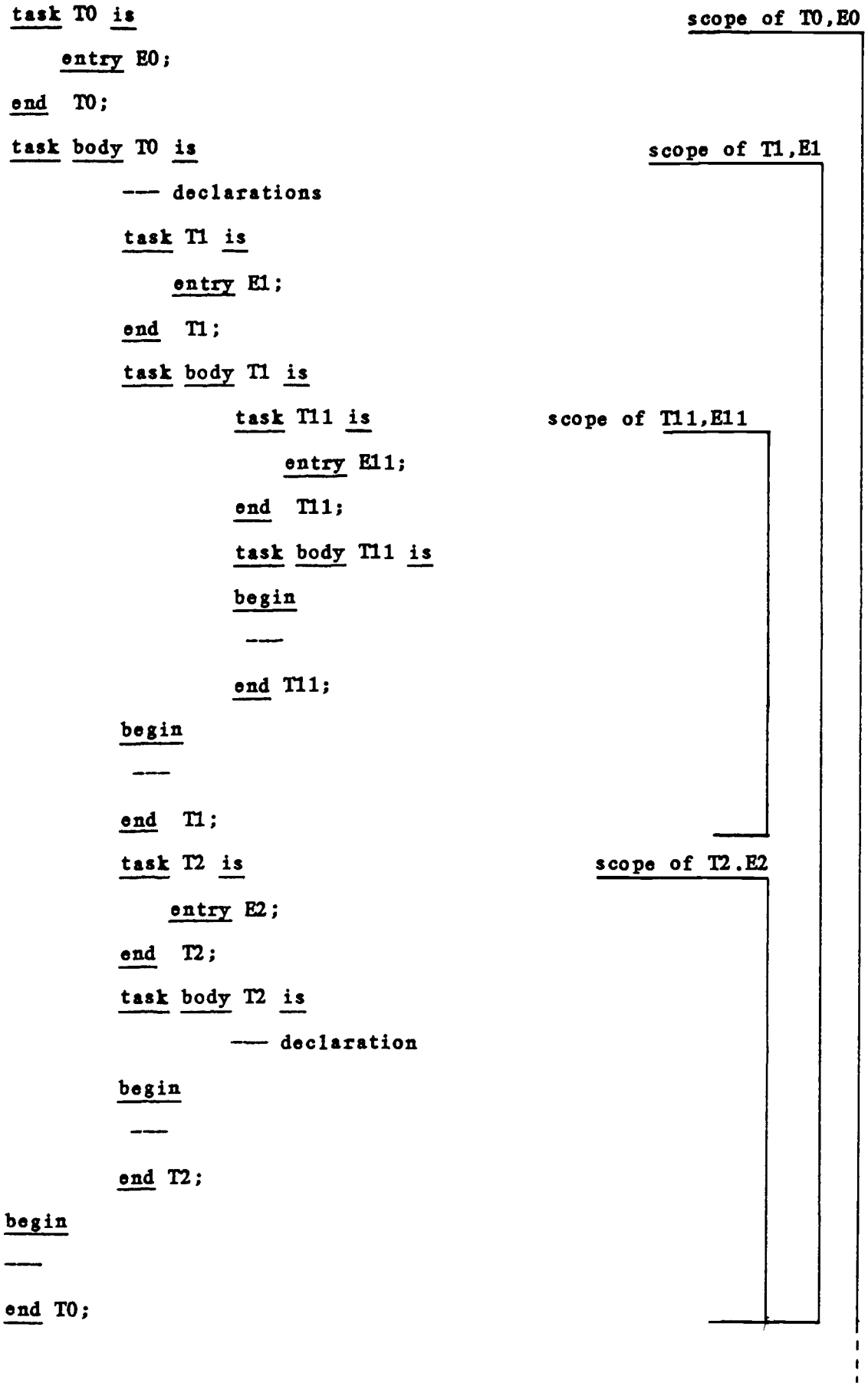
$$T \longrightarrow T'$$

means that the body of T is in the scope of the specification of T' and so T can call any entry of T'. Thus in the body of T11 one can call E1, E2 and E0 using corresponding entry call statements directly, but from the body of T2 and T0 one cannot call E11.

According to the Ada manual tasks T0, T1, T2 and T11 can be executed in parallel, i.e, the activation of task T0 consists of the activation of tasks T1 and T2 which are declared immediately within the body of T0; after activation the first statement of T0 is executed, and similarly the activation of task T1 contains the activation of task T11 and so on. This tells us that the parallel structure in Ada is implicit, rather than being given explicitly by constructs such as $\|$ in CSP.

Example 3.2

The tasks are the same as the previous example, but the task bodies follow their specifications immediately.


task T0 is
entry E0;
end T0;
task body T0 is
--- declarations
task T1 is
entry E1;
end T1;
task body T1 is
task T11 is
entry E11;
end T11;
task body T11 is
begin
---
end T11;
begin
---
end T1;
task T2 is
entry E2;
end T2;
task body T2 is
--- declaration
begin
---
end T2;
begin
---
end T0;
scope of T0,E0
scope of T1,E1
scope of T11,E11
scope of T2.E2

The scopes of T11, E11 and T1, E1 are the same as before, but the scopes of T2, E2 are changed since their specification appears in a different place. The calling diagram is shown in figure 3.1.b. In this case one cannot now call E2 from the body of T11. □

To describe these various scopes and to make the implicit parallel structure explicit by abstract syntax we introduce the following multitask statements:

MS ::= task T::E;MS | T::E;S | MS||MS

where S is a statement. The multitask statement task T::E;MS is a binary specification structure, where task T::E is the specification of task T and its entries having scope MS. In the multitask statement T::E;S the statement S denotes the body of the task T. The multitask statement MS||MS is a parallel structure; it means that the constituents execute simultaneously.

It should be pointed out that the multitask statements here are different from the commands process R;C, R::C and C||C given in CSP. The constituents of MS||MS must be multitask statements and the MS occurring in task T::E;MS must also be a multitask statement. For example, the following forms

task T::E; x:=e and T.W(e,x)||accept W(x2,y2) do S end

are not multitask statements.

Using these syntactic clauses example 3.1 can be rewritten as:

```
task T0::E0;(
    task T1::E1;(
        task T2::E2;(
            task T11::E11;(T11::E11;S11‖T1::E1;S1)
            ‖T2::E2;S2
            ‖T0::E0;S0)))
```

where S11, S1, S2, S0 are bodies of T11, T1, T2, T0 respectively. Example 3.2 can be written as:

```
task T0::E0;(
    task T1::E1;(
        task T11::E11;(T11::E11;S11‖T1::E1;S1)
        ‖task T2::E2;(T2::E2;S2‖T0::E0;S0)))
```

This rewriting process makes both the scope of a task and its entries and parallel structure explicit. This may allow some redundant structures. For instance, example 3.1 can also be rewritten as

```
task T0::E0;(
    task T1::E1;(
        task T2::E2;(
            ‖T0::E0;S0
            task T11::E11;(T11::E11;S11‖T1::E1;S1)
            ‖T2::E2;S2)))
```

Even so we will still adopt the above abstract syntax. The reader is requested to take the correctness of the rewriting process as given.

3.1.2 Exceptions in Ada

Exceptions in Ada are intended to provide a facility for dealing with errors or other exceptional situations that arise during the execution of programs. The main points of exceptions concerns their declaration, raising and handling:

Every exception is associated with an identifier called its exception name which must be declared explicitly unless it is predefined. An exception can be raised explicitly by a raise statement: raise U (U is an exception name). When an exception is raised during execution, control jumps from the current point of execution to the exception handler, if any, which corresponds to this exception. An exception handler may appear at the end of a unit (a unit may be a procedure, function, package or task) containing the raise statement and may consist of several exception choices which have the form: when U→S (S is a sequence of statements and U is an exception name).

If an exception is raised in a unit that does not contain a handler for this exception or if a further exception is raised during the execution of the handler then the execution of the unit is abandoned and the same exception is raised again implicitly in the unit whose body immediately surrounds the previous one. In this case, the exception is said to be propagated. If the unit is itself the main program, then execution of the main program is aborted.

Exceptions interact with communication and parallelism. That is: any task can raise a failure exception in another visible task (say T) using a statement raise T'FAILURE. The execution of this statement has no direct effect on the task performing it. When the task T receives this failure information an exception is raised immediately at the current point of its execution unless this task has issued an entry call and the corresponding rendezvous has

already started; in this case the rendezvous is allowed to finish. Failure is the only exception that can be raised explicitly by one task in another. If an exception is raised in an accept statement body during rendezvous, and it is not handled locally (by the body of the accept statement), then the exception is propagated to the unit immediately surrounding the accept statement, and the same exception is raised simultaneously in the calling task at the point of the entry call. Finally, any attempt to propagate an exception beyond a task body results in an abnormal termination of the task and no further propagation of the exception. Two illustrative examples follow:

Example 3.3

Consider the following program. A raise statement (raise MINUS)occurs in the inner block which does not itself contain a handler, but the outer block does provide a handler.

```
begin
     MINUS:exception;                          the scope of MINUS
      ---
     begin
          ZERO:exception;                  the scope of ZERO
           ---
          if x<0 then raise MINUS;    the first raise
           ---
     end;
     exception
        when ZERO→S1;
     end;
     ---
     raise MINUS;                      the second raise
     ---
end;
exception
   when MINUS→S2
end;
```

A MINUS exception may be raised in two cases:

1. It is raised in the outer block (the second raise statement); then control jumps to S2.

2. It is raised in the inner block (the first raise statement); then it is propagated from the inner block and raised again in the outer block implicitly, whereupon control jumps to S2. □

Example 3.4

Consider the following program in which a failure exception may be raised by T2 in T1 and propagated during communication:

```
begin
     task T1 is
         entry W1(x1:in,y1:out)
     end  T1;
     task T2 is
         entry W2(x2:in,y2:out);
     end  T2;
     task body T1 is
          z1:integer;
     begin
          accept W1(x1,y1) do
                 T2.W2(x1-1,z1);
                 y1:=x1+z1
          end;
          exception FAILURE→null
     end T1;
     task body T2 is
     begin
          accept W2(x2,x2) do
                 select when x2=0⇒raise T1'FAILURE
                     or
                        when x2≠0⇒ S
                 end;
          end;
     end T2;
begin
     T1.W1(1,y);
end;
exception
    when FAILURE→y:=3
end;
```

The communication starts when the main block calls entry W1 of task T1. During the rendezvous T1 calls W2 of T2 and then T2 raises a failure exception in T1. When this failure exception is raised in T1 the rendezvous with the main block has not yet finished, a FAILURE exception will be propagated to the main block. □

To model all these features by abstract syntax we introduce the following syntactic clauses:

S ::= ... | exception U;S | S except XC | raise U | traise T

XC ::= U→S | XC1|XC2

where S denotes a statement and XC denotes an exception handler. The statement exception U;S denotes the declaration of the exception U with scope S. The statement S except XC models an exception handler XC following a statement S. Furthermore, we decompose the functions of a raise statement into raising an ordinary exception (raise U), and raising a failure exception in another task (traise T). Thus example 3.3 can be rewritten as:

```
exception MINUS,(...;
                exception ZERO;(...;
                                raise MINUS;
                                ...;
                               )except ZERO→S1;
                ...;
                raise MINUS;
                ...;
               )except MINUS→S2;
```

Example 3.4 can be rewritten as:

```
task T::E;(
    task T1::E1;(
        task T2::E2;(T1::E1;S1 except FAILURE →null
                    ‖T2::E2;S2
                    ‖T::E;S except FAILURE →S4)))
```

where S1, S2 and S are the bodies of tasks T1, T2 and the main block respectively; the statement S4 is the outermost exception handler.

After this brief informal explanation, we are now ready to give a formal description of multitasking and exception handling in Ada.

3.2 An operational semantics for multitasking in Ada

The purpose of this section is to give an operational semantics for multitasking in Ada. In subsection 3.2.1 an abstract syntax is given for a small subset of Ada which contains multitasking and communication mechanisms in Ada. We call it Ada.1. A static and dynamic semantics of Ada.1 are studied in subsections 3.2.2 and 3.2.3 respectively.

3.2.1 The syntax of Ada.1

The abstract syntax of Ada.1 is parameterised on the following syntactic categories:

The sets Var, Exp, Bexp, are those of CSP as given in section 2.1.

Tnm - a given countably infinite set of task names, ranged over by T.

Win - a given countably infinite set of entry names, ranged over by W.

The main syntactic categories of Ada.1 are defined below using a BNF-like notation:

Entr - a set of entries, ranged over by E and defined by:

$$E ::= \underline{\text{empty}} \mid W \mid E;E$$

Gstm - a set of guarded statements, ranged over by GS and defined by:

$$GS ::= b \Rightarrow S \mid GS \ \underline{\text{or}} \ GS$$

Stm - a set of statements, ranged over by S and defined by:

$$\begin{aligned} S ::= {} & \underline{\text{skip}} \mid \underline{\text{abort}} \mid x:=e \mid S;S \mid \underline{\text{select}} \ GS \ \underline{\text{end}} \mid \\ & \underline{\text{loopselect}} \ GS \ \underline{\text{end}} \mid \underline{\text{accept}} \ W(x,e) \ \underline{\text{do}} \ S \mid \\ & T.W(e,x) \mid MS \end{aligned}$$

Mstm - a set of multitask statements, ranged over by MS and defined by:

$$MS ::= T::E;S \mid \underline{\text{task}} \ T::E;MS \mid MS \| MS$$

Many interesting aspects of Ada are omitted, such as types, declarations, and packages. However, by reducing the complexity of the language we can concentrate our attention on those aspects we wish to scrutinise. It should be mentioned that we do not understand how to specify the semantics of realtime control of multitasking in Ada. It also should be pointed out that from the semantic viewpoint Ada's if-statements, case-statements and loop-statements can all be

described using select and loopselect statements. For example, the statement if b then S1 else S2 can be written as select b⇒S1 or not(b)⇒S2 end; the statement while b loop S can be written as loopselect b⇒S end. A block in Ada can be viewed either as a task with no entries if it contains subtasks or a list of statements, and a program can be viewed as a special task:

Pr ::= task P::empty;MS.

3.2.2 Static semantics

Just like CSP, Ada.1 also requires a static semantics to guarantee that all program to be executed are valid, i.e, they do not violate the requirements of the Ada manual and they contain no syntactic errors.

The following sets are needed to define the static semantics:

Syn - the union of the sets Gstm and Stm and Mstm, ranged over by Ω, i.e, Ω can be a guarded statement or a statement or a multitask statement.

DEN(E) - the set of entry names contained in the entry declaration E.

FTA(Ω) - the set containing a pair for each free task agent occurrence in Ω, giving the task name together with the set of entries of the task.

FTO(Ω) - the set of pairs, where for each entrycall statement T.W(e,x) in Ω, FTO(Ω) contains the pair (T,W) consisting of the free task object name together with the entry.

FEO(Ω) - the set of free entry object names occurring in accept statements contained in Ω.

For the guarded statements we define the following predicate:

Bool(GS) - the disjunction of the guards occurring in GC.

All these sets and the above predicate are defined by structural induction in the following tables:

	empty	W	E1;E2
DEN	$\emptyset$	{W}	DEN(E1) $\cup$ DEN(E2)

	b $\Rightarrow$ S	GS1 or GS2
FTA	FTA(S)	FTA(GS1) $\cup$ FTA(GS2)
FTO	FTO(S)	FTO(GS1) $\cup$ FTO(GS2)
FEO	FEO(S)	FEO(GS1) $\cup$ FEO(GS2)
Bool	b	Bool(GS1) $\vee$ Bool(GS2)

	skip	abort	x:=e	accept W(x,e) do S	T.W(e,x)
FTA	$\emptyset$	$\emptyset$	$\emptyset$	FTA(S)	$\emptyset$
FTO	$\emptyset$	$\emptyset$	$\emptyset$	FTO(S)	{(T,W)}
FEO	$\emptyset$	$\emptyset$	$\emptyset$	{W} $\cup$ FEO(S)	$\emptyset$

	select GS end	loopselect GS end	S1;S2
FTA	FTA(GS)	FTA(GS)	FTA(S1) $\cup$ FTA(S2)
FTO	FTO(GS)	FTO(GS)	FTO(S1) $\cup$ FTO(S2)
FEO	FEO(GS)	FEO(GS)	FEO(S1) $\cup$ FEO(S2)

	task T::E;MS	T::E;S	MS1 $\|\|$ MS2
FTA	FTA(MS) $\setminus$ {(T,DEN(E))}	{(T,DEN(E))}	FTA(MS1) $\cup$ FTA(MS2)
FTO	FTO(MS) $\setminus$ {(T,W) $\mid$ W $\in$ DEN(E)}	FTO(S)	FTO(MS1) $\cup$ FTO(MS2)
FEO	FEO(MS) $\setminus$ DEN(E)	FEO(S)	FEO(MS1) $\cup$ FEO(MS2)

Similar to CSP we use the following notations:

$\vdash_e$ E to mean that the entry declaration E is valid.

$\vdash_{gs}$ GS to mean that the guarded S statement is valid.

$\vdash_s$ S to mean that the statement S is valid.

$\vdash_{ms}$ MS to mean that the multitask statement MS is valid.

All valid syntactic forms are given by structural induction below:

Entries

1. $\vdash_e$ empty

2. $\vdash_e$ W

3. $\dfrac{\vdash_e E1,\ \vdash_e E2}{\vdash_e E1;E2}$ if $DEN(E1) \cap DEN(E2) = \emptyset$

Guarded statements

1. $\dfrac{\vdash_s S}{\vdash_{gs} b \Rightarrow S}$

2. $\dfrac{\vdash_{gs} GS1,\ \vdash_{gs} GS2}{\vdash_{gs} GS1 \text{ or } GS2}$

Statements

1. $\vdash_s$ skip

2. $\vdash_s$ abort

3. $\vdash_s x:=e$

4. $\dfrac{\vdash_s S1,\ \vdash_s S2}{\vdash_s S1;S2}$

5. $\dfrac{\vdash_{gs} GS}{\vdash_{s} \underline{\text{select}}\ GS\ \underline{\text{end}}}$ 6. $\dfrac{\vdash_{gs} GS}{\vdash_{s} \underline{\text{loopselect}}\ GS\ \underline{\text{end}}}$

7. $\vdash_{s} T.W(e,x)$ 8. $\dfrac{\vdash_{s} S}{\vdash_{s} \underline{\text{accept}}\ W(x,e)\ \underline{\text{do}}\ S}$

9. $\dfrac{\vdash_{ms} MS}{\vdash_{s} MS}$

Multitask

1. $\dfrac{\vdash_{s} S,\ \vdash_{e} E}{\vdash_{ms} T::E;S}$ if $FTA(S)=\emptyset$ and $FEO(S) \subseteq DEN(E)$

2. $\dfrac{\vdash_{ms} MS,\ \vdash_{e} E}{\vdash_{ms} \underline{\text{task}}\ T::E;MS}$

3. $\dfrac{\vdash_{ms} MS1,\ \vdash_{ms} MS2}{\vdash_{ms} MS1 \| MS2}$ if $el_1(FTA(MS1)) \cap el_1(FTA(MS2))=\emptyset$

The first condition of multitask rule 1 expresses the requirement that a task specification and corresponding task body "must occur in the same declaration area, the specification first" (see [DoD 80] section 9.1). Thus a task without a specification is illegal, and a task of which the entries declared in task specification are not same as the entries given in the head of the task body is still illegal. For example,

T::E;(T1::E1,S1 ‖ T2::E2,S2) and T'::E';task T::W1;(T::W2;S) are invalid forms. The second condition of multitask rule 1 means that if an accept statement is in a task body then the entry used by the accept statement must be contained in the head of the task body. The condition of multitask rule 3 says that task names in a declaration area must be disjoint. For example, the form T::E1;S1‖T::E2;S2 is not a valid multitask statement. The static semantics of a program is given by:

$\vdash_p$ task T::empty;MS if FTA(task T::empty;MS)=$\emptyset$ and FTO(MS)=$\emptyset$

The first condition means that all task bodies in MS must be declared properly and the second condition means that all called tasks and entries must be declared properly.

Remarks From now on we always use $\vdash \Omega$ to replace $\vdash_\sigma \Omega$ if the category Σ can be understood from the context.

It should be pointed out that we did not consider the sets RV, WV and FV as we did for CSP. The reason is because the use of shared variables is allowed according to the Ada manual, it says "if shared variables are used, it is the programmer's responsibility to ensure that two tasks do not simultaneously modify the same shared variables" (see [DoD 80] section 9.11). This discipline provides great freedom for programmers to use P and V operations and monitor-like mechanisms. In chapter 7 we will discuss this further.

3.2.3 Operational semantics

In this subsection we give an operational semantics for Ada.1. First of all, to describe the termination phase of a rendezvous, we need to add two extra syntactic clauses:

wait(T.W,x) and ack(T,W,e)

to the set Stm of statements. The wait statement means that calling task awaits an acknowledgement from the called task T through its entry W. The ack statement means that the called task sends the value of e through its entry W to the calling task T. The corresponding FTA, FTO and FEO are defined below:

	wait(T.W,x)	ack(T,W,e)
FTA	$\emptyset$	$\emptyset$
FTO	{(T,W)}	$\emptyset$
FEO	$\emptyset$	{W}

and

$$\vdash_s \underline{wait}(T.W,x) \qquad\qquad \vdash_s \underline{ack}(T,W,e)$$

Let $\mathbb{T}_{gs}=\langle \Gamma_{gs}, T_{gs}, \Lambda_{gs}, \xrightarrow{gs} \rangle$, $\mathbb{T}_s=\langle \Gamma_s, T_s, \Lambda_s, \xrightarrow{s} \rangle$, and $\mathbb{T}_{ms}=\langle \Gamma_{ms}, T_{ms}, \Lambda_{ms}, \xrightarrow{ms} \rangle$ be the transition systems for guarded statements, statements and multitask statements respectively. The sets of configurations are defined by:

$$\Gamma_{gs}=\{\langle GS,s\rangle, \langle S,s\rangle \mid GS \in Gstm,\ S \in Stm,\ s \in States\} \cup T_s$$
$$\Gamma_s=\{\langle S,s\rangle \mid S \in Stm,\ s \in States\} \cup T_s$$
$$\Gamma_{ms}=\{\langle MS,s\rangle \mid MS \in Mstm,\ s \in States\} \cup T_s$$

$$T_{gs}=\Gamma_s$$
$$T_s=States \cup \{\underline{abortion}\}$$
$$T_{ms}=T_s$$

The meanings of $\langle \Omega,s\rangle$, s, and abortion are all the same as in CSP. The set of transition actions is given by:

$$\Lambda_{gs}=\Lambda_s$$
$$\Lambda_s = \{\varepsilon\} \cup \{i(N,T.W)!v,\ i(N.W,T)?v,\ f(N.W,T)!v, f(N,T.W)?v \mid T \in Tnm,\ N \in Tnm \cup \{*\},\ v \in V\}$$
$$\Lambda_{ms}=\Lambda_s$$

where * denotes a unknown task name. The meaning of these transition actions is described by the following table:

label λ	task	entry	action	task entry
i(N,T.W)!v	N		sends v to (initialization)	T through W
i(N.W,T)?v	N	through W	receives v from (initialization)	T
f(N.W,T)!v	N	through W	sends v to (termination)	T
f(N,T.W)?v	N		receives v from (termination)	T through W

The symbols i and f indicate that the corresponding transition action are in the initialization phase and termination phase respectively. The action ε means that a transition action is an internal one.

Given a label λ the complementary label $\overline{\lambda}$ is defined by:

$$\overline{\lambda} = \begin{cases} i(T,T'.W)!v & \text{if } \lambda = i(T'.W,T)?v \\ i(T.W,T')?v & \text{if } \lambda = i(T',T.W)!v \\ f(T.W,T')!v & \text{if } \lambda = f(T',T.W)?v \\ f(T,T'.W)?v & \text{if } \lambda = f(T'.W,T)!v \end{cases}$$

where $T,T' \varepsilon Tnm$ and $T \neq T'$.

And we are finally in a position to give all the transitions:

Guarded statements

guards

$$1.\ \frac{\langle S,s\rangle \xrightarrow{\lambda} r,\ [\![b]\!]_s = tt}{\langle b \Rightarrow S,s\rangle \xrightarrow{\lambda} r}$$

$$2.\ \frac{[\![b]\!]_s = \underline{error}}{\langle b \Rightarrow S, s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

alternative

$$1.\ \frac{\langle GS_1, s\rangle \xrightarrow{\lambda} \langle S_1, s'\rangle | s' | \underline{abortion}}{\langle GS_1\ \underline{or}\ GS_2, s\rangle \xrightarrow{\lambda} \langle S_1, s'\rangle | s' | \underline{abortion}}$$

$$2.\ \frac{\langle GS_2, s\rangle \xrightarrow{\lambda} \langle S_2, s'\rangle | s' | \underline{abortion}}{\langle GS_1\ \underline{or}\ GS_2, s\rangle \xrightarrow{\lambda} \langle S_2, s'\rangle | s' | \underline{abortion}}$$

The guard rules mean that if the boolean expression b is true then the behaviour of $b \Rightarrow S$ is the same as the behaviour of S; if the value of b is an error then it will result in abortion.

Alternative rules say that an arbitrary constituent with successfully executable guard is selected and executed.

Statements

assign $1.\ \dfrac{[\![e]\!]_s = v,\ v \neq \underline{error}}{\langle x:=e, s\rangle \xrightarrow{\varepsilon} \langle \underline{skip}, s[v/x]\rangle}$

$$2.\ \frac{[\![e\ ]\!]_s = \underline{error}}{\langle x:=e, s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

skip $\quad \langle \underline{skip}, s\rangle \xrightarrow{\varepsilon} s$

abort $\quad \langle \underline{abort}, s\rangle \xrightarrow{\varepsilon} \underline{abortion}$

composition $\quad \dfrac{\langle S_1, s\rangle \xrightarrow{\lambda} \langle S_1', s'\rangle | s' | \underline{abortion}}{\langle S_1; S_2, s\rangle \xrightarrow{\lambda} \langle S_1'; S_2, s'\rangle | \langle S_2, s'\rangle | \underline{abortion}}$

All these rules are similar to those in CSP.

select 1. $$\frac{\langle GS,s\rangle \xrightarrow{\lambda} \langle S,s'\rangle \mid s' \mid \underline{abortion}}{\langle \underline{select}\ GS\ \underline{end},s\rangle \xrightarrow{\lambda} \langle S,s'\rangle \mid s' \mid \underline{abortion}}$$

2. $$\frac{[\![Bool(GS)]\!]_s = ff}{\langle \underline{select}\ GS\ \underline{end},s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

loopselect 1. $$\frac{\langle GS,s\rangle \xrightarrow{\lambda} \langle S,s'\rangle}{\langle \underline{loopselect}\ GS\ \underline{end},s\rangle \xrightarrow{\lambda} \langle S;\underline{loopselect}\ GS\ \underline{end},s'\rangle}$$

2. $$\frac{\langle GS,s\rangle \xrightarrow{\lambda} s'}{\langle \underline{loopselect}\ GS\ \underline{end},s\rangle \xrightarrow{\lambda} \langle \underline{loopselect}\ GS\ \underline{end},s'\rangle}$$

3. $$\frac{\langle GS,s\rangle \xrightarrow{\lambda} \underline{abortion}}{\langle \underline{loopselect}\ GS\ \underline{end},s\rangle \xrightarrow{\lambda} \underline{abortion}}$$

4. $$\frac{[\![Bool(GS)]\!]_s = ff}{\langle \underline{loopselect}\ GS\ \underline{end},s\rangle \xrightarrow{\varepsilon} s}$$

The select rule means that the behaviour of the select statement is the same as its guarded statement, if this guarded statement does not fail. Otherwise it will result in abortion. The loopselect rule says that if the guarded statement aborts so does the repetitive statement, if the guarded statement fails then the repetitive statement terminates with no effect, otherwise the guarded statement is executed followed by the loopselect statement.

parallel 1. $$\frac{\langle MS_1,s\rangle \xrightarrow{\lambda} \langle MS_1',s'\rangle \mid s' \mid \underline{abortion}}{\langle MS_1 \| MS_2,s\rangle \xrightarrow{\lambda} \langle MS_1' \| MS_2,s'\rangle \mid \langle MS_2,s'\rangle \mid \underline{abortion}}$$

2. $$\frac{\langle MS_2,s\rangle \xrightarrow{\lambda} \langle MS_2',s'\rangle \mid s' \mid \underline{abortion}}{\langle MS_1 \| MS_2,s\rangle \xrightarrow{\lambda} \langle MS_1 \| MS_2',s'\rangle \mid \langle MS_1,s'\rangle \mid \underline{abortion}}$$

$$3.\ \frac{\langle MS_1, s\rangle \xrightarrow{\lambda} \langle MS_1', s'\rangle,\ \langle MS_2, s\rangle \xrightarrow{\bar{\lambda}} \langle MS_2', s\rangle}{\langle MS_1 \| MS_2, s\rangle \xrightarrow{\varepsilon} \langle MS_1' \| MS_2', s'\rangle}$$

$$4.\ \frac{\langle MS_1, s\rangle \xrightarrow{\lambda} \langle MS_1', s'\rangle,\ \langle MS_2, s\rangle \xrightarrow{\bar{\lambda}} \langle MS_2', s\rangle}{\langle MS_2 \| MS_1, s\rangle \xrightarrow{\varepsilon} \langle MS_2' \| MS_1', s'\rangle}$$

Rules 1 and 2 mean that the first step of a multitask statement $MS_1 \| MS_2$ can be that of any one of its constituents, and the parallel statement terminates normally only if and when all its constituents have terminated normally; if one of the constituents aborts then the multitask statement aborts.

Rules 3 and 4 mean that communication between the constituents of the multitask statement is similar to that of CSP and is achieved by the simultaneous occurrence of complementary sending and receiving actions, which can be either at the initialization or termination of a rendezvous.

statements

initialization:

entrycall 1. $$\frac{[\![e]\!]_s = v,\ v \neq \underline{error}}{\langle T.W(e,x), s\rangle \xrightarrow{i(*,T.W)!v} \langle \underline{wait}(T.W,x), s\rangle}$$

2. $$\frac{[\![e]\!]_s = \underline{error}}{\langle T.W(e,x), s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

accept $$\langle \underline{accept}\ W(x,e)\ \underline{do}\ S, s\rangle \xrightarrow{i(*.W,T)?v} \langle S; \underline{ack}(T,W,e), s[v/x]\rangle$$

termination:

wait $\quad \langle \underline{wait}(T.W,x),s\rangle \xrightarrow{f(*,T.W)?v} \langle \underline{skip}, s[v/x]\rangle$

ack $\quad 1.\ \dfrac{[\![e]\!]_s = v,\ v \neq \underline{error}}{\langle \underline{ack}(T,W,e),s\rangle \xrightarrow{f(*.W,T)!v} \langle \underline{skip},s\rangle}$

$\quad 2.\ \dfrac{[\![e]\!]_s = \underline{error}}{\langle \underline{ack}(T,W,e),s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$

The entrycall and accept rules model the initialisation phase of a rendezvous between appropriate tasks; the entry call rule says that if the expression can be evaluated properly then send the value of the expression e to the task T through entry W and suspend (wait for acknowledgement), otherwise abort. The accept rule says receive a value v from task T and then execute the body of the accept statement followed an acknowledgement statement which will terminate the rendezvous.

Similarly, the wait and the ack rules model the termination phase of a rendezvous. The wait rule means receive a value v from the task T through its entry W and become $\langle \underline{skip}, s[v/x]\rangle$. The ack rule says that if the expression e can be evaluated properly then send this value to the task T through entry W and become $\langle \underline{skip}, s\rangle$, otherwise abort.

We have seen that these rules together with the parallel rule model the requirement of the Ada manual:

". if a calling task issues an entry call before a corresponding accept statement is reached by the task owning the entry, the execution of the calling task is suspended.
if a task reaches an accept statement prior to any call of that entry, the execution of the task is suspended until such a call occurs.

When an entry has been called and a corresponding accept statement is reached, the sequence of statements, if any, of the accept statement is executed by the called task (while the calling task remains suspended). This interaction is called rendezvous. Thereafter, the calling task and the task owning the entry can continue their execution in parallel" (see [DoD 80] section 9.5).

$$\text{B-task}\quad \frac{\langle S,s\rangle \xrightarrow{\lambda} \langle S',s'\rangle \mid s' \mid \underline{\text{abortion}},\ \phi_T(\lambda)\downarrow}{\langle T::E;S,s\rangle \xrightarrow{\phi_{T,E}(\lambda)} \langle T::E;S',s'\rangle \mid s' \mid \underline{\text{abortion}}}$$

where $\phi_T: \Lambda_a \to \Lambda_a$ is the partial function defined by

$$\phi_T(\lambda) = \begin{cases} \varepsilon & \text{if } \lambda=\varepsilon \\ i(T,T'.W)! & \text{if } \lambda=i(*,T'.W)!v \quad T\neq T' \\ i(T.W,T')?v & \text{if } \lambda=f(*.W,T')?v \quad T\neq T' \\ f(T.W,T')!v & \text{if } \lambda=f(*.W,T')!v \quad T\neq T' \\ f(T,T'.W)?v & \text{if } \lambda=f(*,T'.W)?v \quad T\neq T' \end{cases}$$

The B-task rule defines transitions for the task bodies and the meaning of the function ϕ_T is as follows:
The case $\lambda=\varepsilon$ means that if a transition action is internal (without any interaction with the environment) viewed from within the body, then it remains so when viewed from outside the body.
The second case $\lambda=i(*,T'.W)!v$ indicates that in the initialization of a rendezvous the action as viewed from within the body of sending

the value v to the task T' through its entry W, when viewed from outside of the body becomes the action of the task T sending v to T' through its entry W.

The third case $\lambda = i(*.W,T')?v$ indicates that the action as viewed from within the body of receiving the value v from the task T' through entry W, when viewed from outside of the body becomes the action of task T receiving the value v through entry W from task T'. The condition $T \neq T'$ means that a task which tries to communicates with itself will deadlock.
Similarly, the fourth and fifth cases deal with the actions in termination of a rendezvous.

D-task
$$\frac{\langle S,s\rangle \xrightarrow{\lambda} \langle S',s'\rangle \mid s' \mid \underline{abortion},\ \eta_T(\lambda)\downarrow}{\langle \underline{task}\ T::E;MS,s\rangle \xrightarrow{\eta_T(\lambda)} \langle \underline{task}\ T::E;S',s'\rangle \mid s' \mid \underline{abortion}}$$

where $\eta_T: \Lambda_a \to \Lambda_a$ is the
partial function defined by

$$\eta_T(\lambda) = \begin{cases} \lambda & \text{if } \lambda = \varepsilon \\ \lambda & \text{if } \lambda = i(T',T''.W)!v \text{ and } T \neq T'' \\ \lambda & \text{if } \lambda = i(T'.W,T'')?v \text{ and } T \neq T',\ T \neq T'' \\ \lambda & \text{if } \lambda = f(T'.W,T'')!v \text{ and } T \neq T',\ T \neq T'' \\ \lambda & \text{if } \lambda = f(T',T''.W)?v \text{ and } T \neq T'' \\ \text{undefined} & \text{otherwise} \end{cases}$$

The D-task rule defines transitions for the task specifications. The intention of $\eta_T(\lambda)$ is to implement the scope rules. To explain the meaning of η_T we analyse the two cases: consider the case $\lambda = i(T',T''.W)!v$ then η_T deals with the action in initialization of a rendezvous. There two cases:
1. $T' \neq *$. Then by the B-task rule there is a multitask statement T'::E;S immediately occurring in task T::E;MS. There are still two

subcases to analyse:

a. T=T". Then λ=i(T',T.W)!v. This means that T' sends v to task T through its entry W. Since task T::E;MS defines the scope of T, the T in i(T',T.W)!v must refer to the T in task T::E;MS and η_T must be undefined and this is included in the last case of the definition.

b. T≠T" Then λ=i(T',T".W)!v. It means that T' sends v to task T" through its entry W. Since task T::E;MS only deals with the scope of T, from the outside of task T::E;MS the action should still be the same and this is the second case of the definition. Note that here T' can be T and this says that T is a local task contained in the task T".

2. T'=*. This means that there is a statement T.W(e,x) immediately occurring in task T::E;MS. According to the definition of MS it is impossible.

Let us analyse the case λ=i(T'.W,T")?v. As in the above case we know that T'≠* and that there is a multitask statement T'::E';S immediately occurring in task T::E;MS. There are three subcases to examine:

a. T'=T. Then λ=i(T.W,T")?v. This means that the task T receives v through entry W from the task T". Since task T::E;MS defines the scope of T task T is not visible from the outside of task T::E;MS. So η_T must be undefined in this case.

b. T≠T' and T=T". Then λ=i(T'.W,T)?v and this means that task T' receives v through entry W from the task T. Similarly, since task T::E;MS defines the scope of T the T in i(T'.W,T)?v must refer to the T in task T::E;MS. So the action λ is meaningless from the outside of task T::E;MS and η_T is still undefined in this case.

c. T≠T' and T≠T". Then λ=i(T'.W,T")?v and it means that task T' receives v through entry W from task T". Again task T::E;MS only deals with the scope of T, so $\eta_T(\lambda)=\lambda$ and this is the third case of

the definition.

Similarly, we can analyse the fourth and the fifth cases in the definition.

3.2.4 Properties and examples

As with the semantics of CSP, a number of facts show that no cases have been overlooked and that the semantics contains no contradictions.

Lemma 3.1

Suppose $r, r' \varepsilon \Gamma_{gs} \cup \Gamma_{s} \cup \Gamma_{ms}$, $\lambda \varepsilon \Lambda_{s}$ and s ε States and m=i or f. Then

1. If $r \xrightarrow{\lambda} s$ then $\lambda=\varepsilon$ and s equals the state in r.

2. If $r \xrightarrow{\lambda} \underline{\text{abortion}}$ then $\lambda=\varepsilon$.

3. If $r \xrightarrow{\lambda} r'$ and λ has the form m(N,T.W)!v where N ε Tnm∪{*} then r and r' have the forms ⟨Ω,s⟩ and ⟨S',s⟩ respectively, and
$\langle \Omega, s \rangle \xrightarrow{m(N,T.W)!v} \langle S', s \rangle$.

4. If $r \xrightarrow{\lambda} r'$ and λ has the form m(N.W,T)?v then r and r' have the forms ⟨Ω,s⟩ and ⟨S',s[v/x]⟩ respectively, and for any value v'
$\langle \Omega, s \rangle \xrightarrow{m(N.W,T)?v'} \langle S', s[v'/x] \rangle$.

Proof. The proof is by structural induction. □

The meaning of this lemma is obvious and is similar to that of CSP (see lemma 2.1).

Lemma 3.2

If $\langle \Omega, s \rangle \xrightarrow{\lambda} \langle \Omega', s' \rangle$ then

$$FTA(\Omega') \subseteq FTA(\Omega),\ FTO(\Omega') \subseteq FTO(\Omega) \text{ and } FEO(\Omega') \subseteq FEO(\Omega)$$

Proof. The proof is by structural induction and is similar to those of CSP. Let us prove the result for FTA just examining the following

interesting cases:

case 1. Ω is accept W(x,e) do S. Then Ω' must be S;ack(T,W,e) and noticing:

$$FTA(ack(T,W,e))=\emptyset \quad \text{so}$$

$$FTA(\Omega')=FTA(S)\cup FTA(ack(T,W,e))=FTA(S)=FTA(\underline{accept}\ W(x,e)\ \underline{do}\ S)$$

case 2. Ω is $S_1;S_2$. Then Ω must be $S_1';S_2$ or S_2. For the first case according to the composition rule we have $\langle S_1,s\rangle \xrightarrow{\lambda} \langle S_1',s'\rangle$. By the induction hypothesis $FTA(S_1')\subseteq FTA(S_1)$, so

$$FTA(\Omega')=FTA(S_1';S_2)=FTA(S_1')\cup FTA(S_2$$
$$\subseteq FTA(S_1)\cup FTA(S_2)=FTA(S_1;S_2)$$

For the second case the proof is similar.

case 3. Ω is loopselect GS end. Then Ω' can only be

S;loopselect GS end or loopselect GS end.

For the first case by the loopselect rule we have: $\langle GS,s\rangle \xrightarrow{\lambda} \langle S,s'\rangle$. By the induction hypothesis $FTA(S)\subseteq FTA(GS)$ therefore:

$$\begin{aligned} FTA(\Omega') &= FTA(S)\cup FTA(\underline{loopselect}\ GS\ \underline{end}) \\ &\subseteq FTA(GS) \\ &= FTA(\underline{loopselect}\ GS\ \underline{end}) \end{aligned}$$

case 4. Ω is task T::E;MS. Then Ω must be task T::E;MS' and $\langle MS,s\rangle \xrightarrow{\lambda'} \langle MS',s'\rangle$ where $\eta_T(\lambda')=\lambda$. By the induction hypothesis $FTA(MS')\subseteq FTA(MS)$. Thus

$$FTA(\Omega')=FTA(\underline{task}\ T::E;MS')=FTA(MS')\setminus\{(T,DEN(E))\}$$
$$\subseteq FTA(MS)\setminus\{(T,DEN(E))\}=FTA(\Omega) \quad \square$$

This lemma shows the interaction between static semantics and the dynamic semantics and it enables us to prove the following theorem:

Theorem 3.1

Let $\Omega \varepsilon Syn$, $\lambda \varepsilon \Lambda_s$ and $\vdash \Omega$. If $\langle \Omega, s\rangle \xrightarrow{\lambda} r$ then either $r=\langle \Omega', s'\rangle$ and $\vdash \Omega'$ or r is one of abortion, $s' \varepsilon States$.

Proof. The proof is by structural induction on Ω employing the above lemma. Let us examine the following cases:

case 1. Ω is skip, abort, x:=e, wait(T.W,x) and ack(T,W,e). Then according to the corresponding rules the result is immediate.

case 2. Ω is $b \Rightarrow S$. Then if $[\![e]\!]_s$=error then r' must be abortion; otherwise r' is one of the forms: abortion, s' and $\langle S', s'\rangle$ where $\langle S, s\rangle \xrightarrow{\lambda} \langle S', s'\rangle$. For the last case by the induction hypothesis we have $\vdash S'$.

case 3. Ω is $S_1; S_2$. The proof is similar to the corresponding case of theorem 2.1.

case 4. Ω is accept W(x,e) do S. Then $\vdash S$ and r' must be $\langle S; ack(T,W,e), s[v/x]\rangle$. Since $\vdash ack(T,W,e)$ we have $\vdash S; ack(T,W,e)$.

case 5. Ω is loopselect GS end. Then $\vdash GS$ and if $[\![Bool(GS)]\!]_s$=ff the r' must be s'; otherwise r' can only be one of the forms: abortion, $\langle$loopselect GS end,s$\rangle$ and $\langle S;$loopselect GS end, s'$\rangle$. For the last case we have $\langle GS, s\rangle \xrightarrow{\lambda} \langle S, s'\rangle$. By the induction hypothesis $\vdash S$. thus

$$\vdash S; \text{loopselect GS end}.$$

case 6. Ω is $MS_1 \| MS_2$. Then $\vdash MS_1$, $\vdash MS_2$ and $el_1(FTA(MS_1)) \cap el_1(FTA(MS_2)) = \emptyset$. According to the parallel rule r' must be one of the following forms:

abortion, $\langle MS_1, s'\rangle$, $\langle MS_2, s'\rangle$, $\langle MS_1' \| MS_2, s'\rangle$, $\langle MS_1 \| MS_2', s'\rangle$ and $\langle MS_1' \| MS_2', s'\rangle$. For the first three cases the result is obvious, let us check the other cases:

a. r' is $\langle MS_1' \| MS_2, s' \rangle$. Then $\langle MS_1, s \rangle \xrightarrow{\lambda} \langle MS_1', s' \rangle$. By the induction hypothesis $\vdash MS_1'$. By the lemma 3.2 we have:

$$el_1(FTA(MS_1')) \subseteq el_1(FTA(MS_1))$$

thus

$$el_1(FTA(MS_1')) \cap el_1(FTA(MS_2)) \subseteq el_1(FTA(MS_1)) \cap el_1(FTA(MS_2)) = \emptyset$$

b. r' is $\langle MS_1 \| MS_2', s' \rangle$. The proof is similar to subcase (a).

c. r' is $\langle MS_1' \| MS_2', s' \rangle$. Then $\lambda = \varepsilon$ and $\langle MS_1, s \rangle \xrightarrow{\rho} \langle MS_1', s \rangle$ and $\langle MS_2, s \rangle \xrightarrow{\bar{\rho}} \langle MS_2' s' \rangle$. By the induction hypothesis and lemma 3.2 we have:

$\vdash MS_i'$ and $el_1(FTA(MS_i')) \subseteq el_1(FTA(MS_i))$.

where i=1 or 2. Therefore

$$el_1(FTA(MS_1')) \cap el_1(FTA(MS_2')) \subseteq el_1(FTA(MS_1)) \cap el_1(FTA(MS_2)) = \emptyset$$

case 7. Ω is T::E;S. Then $\vdash S$ and $\vdash E$ and $FTA(S) = \emptyset$ and $FEO(S) \subseteq DEN(E)$, r' can be <u>abortion</u>, s' or $\langle T::E;S', s' \rangle$. For the last case we have $\langle S, s \rangle \xrightarrow{\lambda'} \langle S', s' \rangle$ where $\phi_T(\lambda') = \lambda$. By the induction hypothesis $\vdash S'$. By lemma 3.2 $FTA(S') \subseteq FTA(S)$ and $FEO(S') \subseteq FEO(S)$, so $FTA(S') = \emptyset$ and $FEO(S') \subseteq DEN(E)$ and $\vdash T::E;S'$. □

We now run the two examples given in section 3.1 using our semantics.

Example 3.5

Consider the program Pr1 given in example 3.1:

```
task T1::E1;(
     task T2::E2;((task T11::E11; T11::E11;T2.E2(1,x)||T1::E1;S1)
                  ||T2::E2;accept E2(y,z-2) do (z:=y+1)
                  ||T0::empty,S0))
```

Let the initial state be s. To emphasize the interesting variables in the examples we will use x=n to replace the form s[n/x] and (x=n,y=m,z=k) to mean s[n/x][m/y][k/z] and so on.

A computation of Pr1 should be as follows:

$\langle Pr1, s\rangle \xrightarrow[1]{\varepsilon}$

task T1::E1;(
task T2::E2;((task T11::E11; T11::E11;wait(T2.E2,x) ‖ T1::E1;S1) ,y=1
‖T2::E2; z:=y+1; ack(T11,E2,z-2)
‖T0::empty;S0))

$\xrightarrow[2]{\varepsilon}$

task T1::E1;(
task T2::E2;((task T11::E11; T11::E11;wait(T2.E2,x) ‖ T1::E1;S1) ,y=1
‖T2::E2; skip; ack(T11,E2,z-2) z=2
‖T0::empty;S0))

$\xrightarrow[3]{\varepsilon}$

task T1::E1;(x=0
task T2::E2;((task T11::E11; T11::E11;skip ‖ T1::E1;S1) , y=1
‖T2::E2; skip z=2
‖T0::empty;S0))

$\xrightarrow{*}$

task T1::E1;(x=0
task T2::E2;((task T11::E11; T1::E1;S1) ,y=1
‖T0:empty;S0)) z=2

It should be pointed out that every transition step is obtained from the transition rules given in section 3.2.2. For example, step 1 is obtained by applying the D-task rule twice since the following transition holds:

$$\left\langle \begin{array}{l} (\underline{\text{task}}\ \text{T11::E11};\ \text{T11::E11};\text{T2.E2}(1,x) \,\|\, \text{T1::E1};\text{S1}) \\ \|\,\text{T2::E2};\underline{\text{accept}}\ \text{E2}(y,z\text{-}2)\ \underline{\text{do}}\ (z:=y+1) \qquad\qquad ,s \\ \|\,\text{T0::}\underline{\text{empty}};\text{S0} \end{array} \right\rangle \xrightarrow{\varepsilon}$$

$$\left\langle \begin{array}{l} (\underline{\text{task}}\ \text{T11::E11};\ \text{T11::E11};\underline{\text{wait}}(\text{T2.E2},x) \,\|\, \text{T1::E1};\text{S1}) \\ \|\,\text{T2::E2};\ z:=y+1;\ \underline{\text{ack}}(\text{T11},\text{E2},z\text{-}2) \qquad\qquad ,y=1 \\ \text{T0::}\underline{\text{empty}};\text{S0} \end{array} \right\rangle$$

This is from the multitask rule 3 since the following two transitions hold:

a. $\langle \underline{\text{task}}\ \text{T11::E11};(\text{T11::E11};\text{T2.E2}(1,x) \,\|\, \text{T1::E1};\text{S1})\ ,s\rangle$

$$\xrightarrow{\text{i(T11,T2.E2)!1}}$$

$\langle \underline{\text{task}}\ \text{T11::E11};(\text{T11::E11};\underline{\text{wait}}(\text{T2.E2},x) \,\|\, \text{T1::E1};\text{S1})\ ,s\rangle$

b. $\langle \text{T2::E2};\ \underline{\text{accept}}\ \text{E2}(y,z\text{-}2)\ \underline{\text{do}}\ (z:=y+1) \,\|\, \text{T0::}\underline{\text{empty}};\text{S0}\ ,s\rangle$

$$\xrightarrow{\text{i(T2.E2,T11)?1}}$$

$\langle \text{T2::E2};\ z:=y+1;\underline{\text{ack}}(\text{T11},\text{E2},z\text{-}2) \,\|\, \text{T0::}\underline{\text{empty}};\text{S0}\ ,y=1\rangle$

Transition (a) is by the D-task rule (where λ=i(T11,T2.E2)!1) since T11$\neq$T2 and

$$\langle \text{T11::E11};\ \text{T2.E2}(1,x) \,\|\, \text{T1::E1};\text{S1}\ ,s\rangle \xrightarrow{\text{i(T11,T2.E2)!1}}$$

$\langle \text{T11::E11};\ \underline{\text{wait}}(\text{T2.E2},x) \,\|\, \text{T1::E1};\text{S1}\ ,s\rangle$

This is true by the multitask rule 1 since

$$\langle \text{T11::E11};\ \text{T2.E2}(1,x)\ ,s\rangle \xrightarrow{\lambda} \langle \text{T11::E11};\ \underline{\text{wait}}(\text{T2.E2},x)\ ,s\rangle$$

where λ=i(T11,T2.E2)!1, which follows from the B-task rule since

$$\langle \text{T2.E2}(1,x)\ ,s\rangle \xrightarrow{\text{i(*,T2.E2)!1}} \langle \underline{\text{wait}}(\text{T2.E2},x)\ ,s\rangle.$$

The transition (b) is obtained by multitask rule 1 since

$$\langle T2::E2;\ \underline{accept}\ E2(y,z-2)\ \underline{do}\ (z:=y+1)\ ,s\rangle \xrightarrow{i(T2.E2,T11)?1} \langle T2::E2;\ z:=y+1;\underline{ack}(T11,E2,z-2)\ ,\ y=1\rangle$$

this follows from the B-task rule since $T11 \neq T2$, $E2 \varepsilon \{E2\}$ and

$$\langle \underline{accept}\ E2(y,z-2)\ \underline{do}\ (z:=y+1)\ ,s\rangle \xrightarrow{i(*.E2,T11)?1} \langle z:=y+1;\underline{ack}(T11,E2,z-2)\ ,y=1\rangle$$

The rest of the transition steps can be justified in the same fashion. □

Example 3.6

Consider the following program Pr2 which is a special case of example 3.2:

```
task T1::E1;((task T11::E11; T11::E11;T2.E2(1,x)‖T1::E1;S1)
            ‖task T2::E2; (T2::E2;accept E2(y,z-2) do (z:=y+1)
                          ‖T0::empty;S0))
```

T2 cannot be called by T11 this time, because T11 is outside the scope of T2. The semantics shows that T2 cannot evolve further since

$$\langle T2::E2;\ \underline{accept}\ E2(y,z-2)\ \underline{do}\ (z:=y+1)\ ,s\rangle \xrightarrow{i(T2.E2,T11)?1} \langle T2::E2;\ z:=y+1;\underline{ack}(T11,E2,z-2),\ y=1\rangle$$

and $\eta_{T2}(i(T2.E2,T11)?1)$ is undefined. □

3.3 Exception handling in the sequential case of Ada

To study the semantics of exception handling in the sequential parts of Ada, we first consider a small subset of Ada which contains the sequential part of Ada.1 and the syntactic clauses concerned with exceptions for the sequential case. We call this small language Ada.2 and the abstract syntax of Ada.2 is given as follows:

The sets Var, Exp, Bexp are those of Ada.1 given in section 3.2.1.

Exn - a given countably infinite set of exception names, ranged over by U. It is assumed to contain the predefined exception names FAILURE and T-ERROR (tasking error see [DoD 80] section 11.6).

Gstm - a given countably infinite set of guarded statements, ranged over by GS and defined by:

$$GS ::= b \Rightarrow S \mid GS \ \underline{or} \ GS$$

Ehdl - a given countably infinite set of exception handlers, ranged over by XC and defined by:

$$XC ::= U \rightarrow S \mid XC \,|\, XC$$

Stm - a given countably infinite set of statements, ranged over by S and defined by:

$$\begin{aligned} S ::= \ & \underline{skip} \mid \underline{abort} \mid x:=e \mid S;S \mid \underline{select} \ GS \ \underline{end} \mid \\ & \underline{loopselect} \ GS \ \underline{end} \mid \underline{raise} \ U \mid \underline{exception} \ U;S \mid \\ & S \ \underline{except} \ XC \end{aligned}$$

For the guarded statements and the first six statements we have

studied their static and dynamic semantics in the previous section. The exception handlers and the last three statements have been discussed informally in section 3.1.2. In this section we give a formal static and dynamic semantics for these statements.

To define the static semantics we also need the set:

XCH(Ω) - the set of exception names as exception choice occurring freely in Ω (see [DoD 80] section 11.2).

	b⇒S	GS1 or GS2
XCH	XCH(S)	XCH(GS1)∪XCH(GS2)

	skip	abort	x:=e	S1;S2
XCH	∅	∅	∅	XCH(S1)∪XCH(S2)

	select GS end	loopselect GS end
XCH	XCH(GS)	XCH(GS)

	raise U	exception U,S	S except XC
XCH	∅	XCH(S)\{U}	XCH(S)∪XCH(XC)

	U → S	XC1 ▮ XC2
XCH	{U}	XCH(XC1)∪XCH(XC2)

We use $\vdash_s S$ and $\vdash_{xc} XC$ to mean the statement S and the exception handler XC are valid respectively. Now the static semantics of exceptions is given by the following rules:

1. $$\frac{\vdash_s S}{\vdash_{xc} U \rightarrow S}$$

2. $$\frac{\vdash_{xc} XC1,\ \vdash_{xc} XC2}{\vdash_{xc} XC1 \blacksquare XC2} \quad \text{if } XCH(XC1) \cap XCH(XC2) = \emptyset$$

The rule 2 says that an exception name cannot have two different handlers in its declaration area. The static semantics of the exception statements is given by the following rules (the static semantics of the other clauses is the same as given in Ada.1:

exception

1. $\vdash_s$ raise U

2. $$\frac{\vdash_s S,\ \vdash_{xc} XC}{\vdash_s S \text{ except } XC}$$

3. $$\frac{\vdash_s S}{\vdash_s \text{exception } U;S}$$

An operational semantics for Ada.2 is given by the following transition systems:

$$\mathbb{T}_{gs}=\langle \Gamma_{gs}, T_{gs}, \Lambda_{gs}, \overline{\longrightarrow}_{gs} \rangle$$
$$\mathbb{T}_{xc}=\langle \Gamma_{xc}, T_{xc}, \Lambda_{xc}, \overline{\longrightarrow}_{xc} \rangle$$
$$\mathbb{T}_{s}=\langle \Gamma_{s}, T_{s}, \Lambda_{s}, \overline{\longrightarrow}_{s} \rangle$$

The sets of configurations are obtained by adding a new configuration jump to the set of configurations given in Ada.1. The configuration jump denotes a condition in which an exception has been raised and control is about to jump to a corresponding exception handler (if any). It is assumed that abnormal termination is the worst case of execution and cannot be handled by exception handlers.

$$T_s = States \cup \{\text{abortion}\} \cup \{\text{jump}\}$$
$$\Gamma_s = \{\langle S, s\rangle \mid S \in Stm,\ s \in States\} \cup T_s$$

$T_{gs}=\Gamma_s$

$\Gamma_{gs}=\{\langle GS,s\rangle,\langle S,s\rangle \mid GS\varepsilon Gstm,\ S\varepsilon Stm,\ s\varepsilon States\}\cup T_s$

$T_{xc}=\Gamma_s$

$\Gamma_{xc}=\{\langle XC,s\rangle,\langle S,s\rangle \mid XC\varepsilon Ehld,\ S\varepsilon Stm,\ s\ \varepsilon States\}$

The sets of transition labels are given by:

$\Lambda_s=\{\varepsilon\}\cup\{U,\ \bar{U} \mid u\varepsilon Exn\ \}$

$\Lambda_{gs}=\Lambda_{xc}=\Lambda_s$

Here the label U means that a signal is sent through channel U and the label $\bar{U}$ means that a signal is received through channel U.

Finally, the transitions for Ada.2 are given by the following rules:

Guarded statements

guards

$$1.\ \frac{\langle S,s\rangle \xrightarrow{\lambda} r,\ [\![b]\!]_s=tt}{\langle b\Rightarrow S,s\rangle \xrightarrow{\lambda} r}$$

$$2.\ \frac{[\![b]\!]_s=\underline{error}}{\langle b\Rightarrow S,s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

alternative

$$1.\ \frac{\langle GS_1,s\rangle \xrightarrow{\lambda} \langle S_1,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}{\langle GS_1\ \underline{or}\ GS_2,s\rangle \xrightarrow{\lambda} \langle S_1,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}$$

$$2.\ \frac{\langle GS_2,s\rangle \xrightarrow{\lambda} \langle S_2,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}{\langle GS_1\ \underline{or}\ GS_2,s\rangle \xrightarrow{\lambda} \langle S_2,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}$$

Rules for the guarded statements are the same as before except that now $\lambda=U$ or $\bar{U}$ is a possibility and in the last case of alternative

rules is introduced to handle the jump conditions.

Statements

assign 1. $$\frac{[\![e]\!]_s = v,\ v \neq \underline{error}}{\langle x:=e, s\rangle \xrightarrow{\varepsilon} \langle \underline{skip}, s[v/x]\rangle}$$

2. $$\frac{[\![e]\!]_s = \underline{error}}{\langle x:=e, s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

skip $$\langle \underline{skip}, s\rangle \xrightarrow{\varepsilon} s$$

abort $$\langle \underline{abort}, s\rangle \xrightarrow{\varepsilon} \underline{abortion}$$

composition $$\frac{\langle S_1, s\rangle \xrightarrow{\lambda} \langle S_1', s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}{\langle S_1; S_2, s\rangle \xrightarrow{\lambda} \langle S_1', s'\rangle \mid \langle S_2, s'\rangle \mid \underline{abortion} \mid \underline{jump}}$$

select 1. $$\frac{\langle GS, s\rangle \xrightarrow{\lambda} \langle S, s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}{\langle \underline{select}\ GS\ \underline{end}, s\rangle \xrightarrow{\lambda} \langle S, s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}$$

2. $$\frac{[\![Bool(GS)]\!]_s = ff}{\langle \underline{select}\ GS\ \underline{end}, s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

loopselect 1. $$\frac{\langle GS, s\rangle \xrightarrow{\lambda} \langle S, s'\rangle}{\langle \underline{loopselect}\ GS\ \underline{end}, s\rangle \xrightarrow{\lambda} \langle S; \underline{loopselect}\ GS\ \underline{end}, s'\rangle}$$

2. $$\frac{\langle GS, s\rangle \xrightarrow{\lambda} s'}{\langle \underline{loopselect}\ GS\ \underline{end}, s\rangle \xrightarrow{\lambda} \langle \underline{loopselect}\ GS\ \underline{end}, s'\rangle}$$

3. $$\frac{\langle GS, s\rangle \xrightarrow{\lambda} \mid \underline{abortion} \mid \underline{jump}}{\langle \underline{loopselect}\ GS\ \underline{end}, s\rangle \xrightarrow{\lambda} \underline{abortion} \mid \underline{jump}}$$

4. $$\frac{[\![Bool(GS)]\!]_s = ff}{\langle \underline{loopselect}\ GS\ \underline{end}, s\rangle \xrightarrow{\varepsilon} s}$$

Rules for the above statements are the same as given in Ada.1 except that for the composition, select and loopselect statements the case of jump is added to model propagation of exceptions raised in these statements.

exception handlers

1. $\langle U \rightarrow S\ , s\rangle \xrightarrow{\bar{U}} \langle S, s\rangle$

2. $$\frac{\langle XCi, s\rangle \xrightarrow{\lambda} \langle S, s'\rangle \quad i=1 \text{ or } 2}{\langle XC1 \,|\, XC2, s'\rangle \xrightarrow{\lambda} \langle S, s'\rangle}$$

raise $\langle \underline{\text{raise}}\ U, s\rangle \xrightarrow{U} \underline{\text{jump}}$

S-except 1. $$\frac{\langle S, s\rangle \xrightarrow{U} \underline{\text{jump}},\ \langle XC, s\rangle \xrightarrow{\bar{U}} \langle S', s\rangle}{\langle S\ \underline{\text{except}}\ XC, s\rangle \xrightarrow{\varepsilon} \langle S', s\rangle}$$

2. $$\frac{\langle S, s\rangle \xrightarrow{U} \underline{\text{jump}},\ U \notin XCH(XC)}{\langle S\ \underline{\text{except}}\ XC, s\rangle \xrightarrow{U} \underline{\text{jump}}}$$

3. $$\frac{\langle S, s\rangle \xrightarrow{\lambda} \langle S', s'\rangle}{\langle S\ \underline{\text{except}}\ XC, s\rangle \xrightarrow{\lambda} \langle S'\ \underline{\text{except}}\ XC, s'\rangle}$$

4. $$\frac{\langle S, s\rangle \xrightarrow{\lambda} s' \,|\, \underline{\text{abortion}}}{\langle S\ \underline{\text{except}}\ XC, s\rangle \xrightarrow{\lambda} \langle \underline{\text{skip}}, s'\rangle \,|\, \underline{\text{abortion}}}$$

D-except 1. $$\frac{\langle S, s\rangle \xrightarrow{\lambda} \langle S', s'\rangle}{\langle \underline{\text{exception}}\ U; S\ , s\rangle \xrightarrow{\lambda} \langle \underline{\text{exception}}\ U; S', s'\rangle}$$

2. $$\frac{\langle S, s\rangle \xrightarrow{\lambda} s' \,|\, \underline{\text{abortion}}}{\langle \underline{\text{exception}}\ U; S, s\rangle \xrightarrow{\lambda} s' \,|\, \underline{\text{abortion}}}$$

3. $$\frac{\langle S, s\rangle \xrightarrow{U'} \underline{\text{jump}},\ U' \neq U}{\langle \underline{\text{exception}}\ U; S, s\rangle \xrightarrow{U'} \underline{\text{jump}}}$$

As we know, the semantics of raising an exception is that of a

jump to the corresponding handler (if any). This provides a possibility to handle exceptions by handshake communication. That is:

1. The raise rule means that raise U sends a signal through "channel U" and becomes a jump condition.

2. Exception handler rule 1 means that the exception handler receives a signal from "channel U" and then executes the corresponding handler.

3. S-except rule 1 says that the whole process of raising and handling an exception is modelled as an internal communication of an except statement.

4. Propagation of an exception is modelled in S-except rule 2, which says that if an exception cannot be handled within a except statement then the same jump condition will arise again in the context immediately surrounding that except statement.

5. Finally, D-except rule 3 implies that an exception cannot be propagated outside its scope.

Let us now examine a example concerned with exception propagation.

Example 3.7

In the following program Pr3 an exception will occur during the execution. Suppose S1, S2 and S3 are valid statements. The program Pr3 is given by:

```
exception ERROR;(
            x:=0;
            exception U;
                    (select x=0⇒raise ERROR
                        or
                            x≠0⇒S1
                    end) except (U→S2);
            S3)except (ERROR→y:=x)
```

Let the initial state be s. According to the semantics a computation of Pr3 in s is given below:

$\langle$Pr3, s$\rangle$

$$\xrightarrow[1]{\varepsilon^*} \left\{ \begin{array}{l} \texttt{exception ERROR;(} \\ \quad \texttt{exception U;(} \\ \quad\quad \texttt{select x=0⇒raise ERROR} \\ \quad\quad\quad \texttt{or} \\ \quad\quad\quad\quad \texttt{x≠0⇒S1} \\ \quad\quad \texttt{end)except (U→S2)} \\ \quad \texttt{S3)except(ERROR→y:=x)} \end{array} \right. \quad , x=0 \left. \vphantom{\begin{array}{l} \texttt{exception ERROR;(} \\ \quad \texttt{exception U;(} \\ \quad\quad \texttt{select x=0⇒raise ERROR} \\ \quad\quad\quad \texttt{or} \\ \quad\quad\quad\quad \texttt{x≠0⇒S1} \\ \quad\quad \texttt{end)except (U→S2)} \\ \quad \texttt{S3)except(ERROR→y:=x)} \end{array}} \right\}$$

$$\xrightarrow[2]{\varepsilon} \langle \texttt{exception ERROR; y:=x} \ , (x=0) \rangle$$

$$\xrightarrow[3]{\varepsilon} \langle \texttt{exception ERROR; skip} \ , (x=0, y=0) \rangle$$

$$\xrightarrow[4]{\varepsilon} (x=0, y=0)$$

All the transitions are obtained by the semantics given above. For instance, transition 2 is derived from the S-except rule 1 since the following two transitions (a) and (b) are true:

a. $\langle \text{ERROR} \rightarrow y:=x\ ,s\rangle \xrightarrow{\overline{\text{ERROR}}} \langle y:=x\ ,s\rangle$

b. $\left\langle \begin{array}{l} \underline{\text{exception}}\ U;(\\ \quad (\underline{\text{select}}\ x=0 \Rightarrow \underline{\text{raise}}\ \text{ERROR} \\ \qquad \underline{\text{or}} \\ \qquad\quad x\neq 0 \Rightarrow S1 \\ \quad \underline{\text{end}})\ \underline{\text{except}}\ (U \rightarrow S2));S3 \end{array} ,x=0 \right\rangle \xrightarrow{\underline{\text{ERROR}}} \underline{\text{jump}}$

The first is by the raise rule and the second is by the composition rule since

$\left\langle \begin{array}{l} \underline{\text{exception}}\ U;(\\ \quad (\underline{\text{select}}\ x=0 \Rightarrow \underline{\text{raise}}\ \text{ERROR} \\ \qquad \underline{\text{or}} \\ \qquad\quad x\neq 0 \Rightarrow S1 \\ \quad \underline{\text{end}})\ \underline{\text{except}}\ (U \rightarrow S2) \end{array} ,x=0 \right\rangle \xrightarrow{\underline{\text{ERROR}}} \underline{\text{jump}}$

the above transition is by D-exception rule 3 since ERROR $\neq$ U and

$\left\langle \begin{array}{l} (\underline{\text{select}}\ x=0 \Rightarrow \underline{\text{raise}}\ \text{ERROR} \\ \quad \underline{\text{or}} \\ \qquad x\neq 0 \Rightarrow S1 \\ \underline{\text{end}})\ \underline{\text{except}}\ U \rightarrow S2 \end{array} ,x=0 \right\rangle \xrightarrow{\underline{\text{ERROR}}} \underline{\text{jump}}$

and this is by S-except rule 2 since ERROR$\neq$U and

$\left\langle \begin{array}{l} \underline{\text{select}}\ x=0 \Rightarrow \underline{\text{raise}}\ \text{ERROR} \\ \quad \underline{\text{or}} \\ \qquad x\neq 0 \Rightarrow S1 \\ \underline{\text{end}} \end{array} ,x=0 \right\rangle \xrightarrow{\underline{\text{ERROR}}} \underline{\text{jump}}$

The above transition relationship is by the select rule since x=0 and

$\langle$raise ERROR ,x=0$\rangle \xrightarrow{\text{ERROR}}$ jump []

3.4 Interaction between exceptions and task communication

In this section we study how exceptions interact with task communication. This interaction may happen in the following ways:

a. An exception can be propagated in a task communication.

b. A task can raise a FAILURE exception in another task while it is executing.

We will also study the problem of a task aborting anotner task while the latter is executing.

We consider a small language which is a slight extension of the union of Ada.1 and Ada.2, we call it Ada.3 and it is defined by:

The sets Var, Exp, Bexp, Tnm, Win and Exn are defined as in Ada.1 or Ada.2.

Gstm - the guarded statements:

GS ::= b⇒S | GS or GS

Ehdl - the exception handlers:

XC ::= U⟶S | XC|XC

Stm - the statements:

S::= skip | abort | x:=e | S;S | select GS end |
loopselect GS end | accept W(x,e) do S |
T.W(x,e) | MS | raise U | except XC |
exception U;S | traise T | tabort T

Mstm - the multitask statements:

MS::= task T::E;MS | T::E;S | MS||MS

We have already seen all these syntactic clauses except the last

two statements, the traise and tabort statements mean that raising a FAILURE exception in or aborting another task as explained in section 3.1.2.

To model the propagation of exceptions during rendezvous we introduce the following statements:

wait(T.W,x) and S rendz(T,W,e) and ack(T,W,e)

to the set Stm of statements, where S∈Stm and the rendz statement is new statement, which models the execution of the body of an accept statement.

The sets FTA, FTO, FEO and XCH for the syntactic clauses of Ada.3 are given below:

	empty	W	E1;E2
DEN	∅	{W}	DEN(E1) ∪ DEN(E2)

	b ⇒ S	GS1 or GS2
FTA	FTA(S)	FTA(GS1) ∪ FTA(GS2)
FTO	FTO(S)	FTO(GS1) ∪ FTO(GS2)
FEO	FEO(S)	FEO(GS1) ∪ FEO(GS2)
XCH	XCH(S)	XCH(GS1) ∪ XCH(GS2)
Bool	b	Bool(GS1) ∪ Bool(GS2)

	U→ S	XC1 | XC2
FTA	FTA(S)	FTA(XC1) ∪ FTA(XC2)
FTO	FTO(S)	FTO(XC1) ∪ FTO(XC2)
FEO	FEO(S)	FEO(XC1) ∪ FEO(XC2)
XCH	{U}	XCH(XC1) ∪ XCH(XC2)

	skip	abort	x:=e	accept W(x,e) do S	T.W(e,x)
FTA	∅	∅	∅	FTA(S)	∅
FTO	∅	∅	∅	FTO(S)	{(T,W)}
FEO	∅	∅	∅	{W} ∪ FEO(S)	∅
XCH	∅	∅	∅	XCH(S)	∅

	select GS end	loopselect GS end	S1;S2
FTA	FTA(GS)	FTA(GS)	FTA(S1) ∪ FTA(S2)
FTO	FTO(GS)	FTO(GS)	FTO(S1) ∪ FTO(S2)
FEO	FEO(GS)	FEO(GS)	FEO(S1) ∪ FEO(S2)
XCH	XCH(GS)	XCH(GS)	XCH(S1) ∪ XCH(S2)

	raise U	S except XC	exception U;S
FTA	∅	FTA(S) ∪ FTA(XC)	FTA(S)
FTO	∅	FTA(S) ∪ FTA(XC)	FTO(S)
FEO	∅	FEO(S) ∪ FEO(XC)	FEO(S)
XCH	∅	XCH(S) ∪ XCH(XC)	XCH(S)\{U}

	task T::E;MS	T::E;S	MS1 ‖ MS2
FTA	FTA(MS)\{(T,DEN(E))}	{(T,DEN(E))}	FTA(MS1) ∪ FTA(MS2)
FTO	FTO(MS)\{(T,W) | W∈𝕃}	FTO(S)	FTO(MS1) ∪ FTO(MS2)
FEO	FEO(MS)	FEO(S)	FEO(MS1) ∪ FEO(MS2)
XCH	XCH(MS)	XCH(S)	XCH(MS1) ∪ XCH(MS2)

where $\mathbb{L}$=DEN(E) ∪ {#} and the symbol # denotes a unknown entry name.

It is easy to see that for the clauses contained in Ada.1 or Ada.2 these sets are evident. The sets of FTA, FTO, FEO and XCH for the new statements are given below:

	wait(T.W,x)	ack(T,W,e)
FTA	∅	∅
FTO	{(T,W)}	∅
FEO	∅	{W}
XCH	∅	∅

	traise T	tabort T	S rendz(T,W,e)
FTA	∅	∅	FTA(S)
FTO	{(T,#)}	{(T,#)}	FTO(S)
FEO	∅	∅	FEO(S) ∪ {W}
XCH	∅	∅	XCH(S)

where the symbol # denotes a unknown entry name.

The static semantics for the clauses contained in Ada.1 or Ada.2 is exactly the same as before. The static semantics for the new statements is given by:

$$\vdash_s \underline{\text{traise}}\ T$$

$$\vdash_s \underline{\text{traise}}\ T$$

$$\frac{\vdash_s S}{\vdash_s S\ \underline{\text{rendz}}(T,W,e)}$$

The static semantics for the remained statements are the same as those given in Ada.1 and Ada.2.

The dynamic semantics of Ada.3 are given by the following transition systems:

$$\mathbb{T}_{gs}=\langle \Gamma_{gs}, T_{gs}, \Lambda_{gs}, \xrightarrow{gs} \rangle,$$

$$\mathbb{T}_{xc}=\langle \Gamma_{xc}, T_{xc}, \Lambda_{xc}, \xrightarrow{xc} \rangle,$$

$$\mathbb{T}_{s}=\langle \Gamma_{s}, T_{s}, \Lambda_{s}, \xrightarrow{s} \rangle$$

and $$\mathbb{T}_{ms}=\langle \Gamma_{ms}, T_{ms}, \Lambda_{ms}, \xrightarrow{ms} \rangle.$$

The sets of configurations are defined by:

$$T_s = States \cup \{\underline{abortion}\} \cup \{\underline{jump}\}$$
$$\Gamma_s = \{\langle S, s\rangle \mid S \in Stm,\ s \in States\} \cup T_s$$

$$T_{gs} = \Gamma_s$$
$$\Gamma_{gs} = \{\langle GS, s\rangle \mid GS \in Gstm,\ s \in States\} \cup \Gamma_s$$

$$T_{xc} = \Gamma_s$$
$$\Gamma_{xc} = \{\langle XC, s\rangle \mid XC \in Ehdl,\ s \in States\} \cup \Gamma_s$$

$$T_{ms} = T_s \setminus \{\underline{jump}\}$$
$$\Gamma_{ms} = \{\langle MS, s\rangle \mid MS \in Mstm,\ s \in States\} \cup T_{ms}$$

Note: the sets of configurations for multitask statements do not contain jump because exceptions cannot be propagated beyond a task. We will discuss this problem later.

To define the transitions for these statements we now need the following transition labels:

$$\Lambda_{od} = \{(N,T)!v,\ (N,T)?v \mid v \in \{\underline{er}, \underline{fl}, \underline{ab}\},\ T \in Tnm,\ N \in Tnm \cup \{*\}\}$$

We use d to denote an element of the set Λ_{od}. It is assumed that the set V contains the distinguished values er, ab and fl , which denote the signals "to raise a T-ERROR exception" "to abort" and "to raise a FAILURE exception" respectively. We sometimes call the values er, fl and ab "orders", the meaning of these "order" transition actions are:

label d	task	action	task
(N,T)!v	N	sends the order v to	T
(N,T)?v	N	receives the order v from	T

where Nε Tnm $\cup$ {*} and TεTnm and vε{ab,fl}. Given a label d its complement is defined by:

$$\bar{d} = \begin{cases} (T,T')?v & \text{if } d=(T',T)!v \\ (T,T')!v & \text{if } d=(T',T)?v \end{cases}$$

where T,T'εTnm, T$\neq$T' and vε{fl,ab}.

Let Λ_{a1}, Λ_{a2} denote the sets Λ_s given in Ada.1 and Ada.2 respectively. The sets of transition labels of Ada.3 are defined by:

$$\Lambda_s=\Lambda_{a1}\cup\Lambda_{a2}\cup\Lambda_{od} \quad \text{and} \quad \Lambda_{gs}=\Lambda_{xc}=\Lambda_{ms}=\Lambda_s$$

To handle propagations in rendezvous we assume that the values which are sent or received by transition actions can be exception names.

When a task receives an order to abort or to raise a FAILURE exception, this order actually interrupts the the normal execution and forces an abortion or raises a FAILURE exception. To model these we need to introduce an interrupt rule for all syntactic clauses, and the concept of basic statement is introduced to mean that a statement whose execution cannot be interrupted. The set of basic statements Bsc is defined by:

Bsc = { skip, abort, x:=e, raise U, traise T,
tabort T, T.W(e,x), accept W(x,e) do S }

We now give the transition rules:

For the syntactic clauses contained in Ada.2 the transition rules are the same as those given in Ada.2, and we list them below:

Guarded statements

guards

1. $$\frac{\langle S,s\rangle \xrightarrow{\lambda} r,\ [\![b]\!]_s = tt}{\langle b \Rightarrow S,s\rangle \xrightarrow{\lambda} r}$$

2. $$\frac{[\![b]\!]_s = \underline{error}}{\langle b \Rightarrow S,s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

alternative

1. $$\frac{\langle GS_1,s\rangle \xrightarrow{\lambda} \langle S_1,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}{\langle GS_1 \ \underline{or}\ GS_2,s\rangle \xrightarrow{\lambda} \langle S_1,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}$$

2. $$\frac{\langle GS_2,s\rangle \xrightarrow{\lambda} \langle S_2,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}{\langle GS_1 \ \underline{or}\ GS_2,s\rangle \xrightarrow{\lambda} \langle S_2,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}$$

Statements

assign

1. $$\frac{[\![e]\!]_s = v,\ v \neq \underline{error}}{\langle x:=e,s\rangle \xrightarrow{\varepsilon} \langle \underline{skip}, s[v/x]\rangle}$$

2. $$\frac{[\![e\ ]\!]_s = \underline{error}}{\langle x:=e,s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

skip $$\langle \underline{skip},s\rangle \xrightarrow{\varepsilon} s$$

abort $$\langle \underline{abort},s\rangle \xrightarrow{\varepsilon} \underline{abortion}$$

composition $$\frac{\langle S_1,s\rangle \xrightarrow{\lambda} \langle S_1',s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}{\langle S_1;S_2,s\rangle \xrightarrow{\lambda} \langle S_1';S_2,s'\rangle \mid \langle S_2,s'\rangle \mid \underline{abortion} \mid \underline{jump}}$$

select 1. $$\frac{\langle GS,s\rangle \xrightarrow{\lambda} \langle S,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}{\langle \underline{select}\ GS\ \underline{end},s\rangle \xrightarrow{\lambda} \langle S,s'\rangle \mid s' \mid \underline{abortion} \mid \underline{jump}}$$

2. $$\frac{[\![Bool(GS)]\!]_s = ff}{\langle \underline{select}\ GS\ \underline{end},s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

loopselect 1. $$\frac{\langle GS,s\rangle \xrightarrow{\lambda} \langle S,s'\rangle}{\langle \underline{loopselect}\ GS\ \underline{end},s\rangle \xrightarrow{\lambda} \langle S;\underline{loopselect}\ GS\ \underline{end},s'\rangle}$$

2. $$\frac{\langle GS,s\rangle \xrightarrow{\lambda} s'}{\langle \underline{loopselect}\ GS\ \underline{end},s\rangle \xrightarrow{\lambda} \langle \underline{loopselect}\ GS\ \underline{end},s'\rangle}$$

3. $$\frac{\langle GS,s\rangle \xrightarrow{\lambda} \mid \underline{abortion} \mid \underline{jump}}{\langle \underline{loopselect}\ GS\ \underline{end},s\rangle \xrightarrow{\lambda} \underline{abortion} \mid \underline{jump}}$$

4. $$\frac{[\![Bool(GS)]\!]_s = ff}{\langle \underline{loopselect}\ GS\ \underline{end},s\rangle \xrightarrow{\varepsilon} s}$$

exception handlers

1. $\langle U \rightarrow S\ ,s\rangle \xrightarrow{\bar{U}} \langle S,s\rangle$

2. $$\frac{\langle XCi,s\rangle \xrightarrow{\lambda} \langle S,s'\rangle \ i=1 \text{ or } 2}{\langle XC1 \mid XC2,s'\rangle \xrightarrow{\lambda} \langle S,s'\rangle}$$

raise $\langle \underline{raise}\ U,s\rangle \xrightarrow{U} \underline{jump}$

S-except 1. $$\frac{\langle S,s\rangle \xrightarrow{U} \underline{jump},\ \langle XC,s\rangle \xrightarrow{\bar{U}} \langle S',s\rangle}{\langle S\ \underline{except}\ XC,s\rangle \xrightarrow{\varepsilon} \langle S',s\rangle}$$

2. $$\frac{\langle S,s\rangle \xrightarrow{U} \underline{jump},\ U \notin XCH(XC)}{\langle S\ \underline{except}\ XC,s\rangle \xrightarrow{U} \underline{jump}}$$

3. $$\frac{\langle S,s\rangle \xrightarrow{\lambda} \langle S',s'\rangle}{\langle S \ \underline{\text{except}}\ XC,s\rangle \xrightarrow{\lambda} \langle S' \ \underline{\text{except}}\ XC,s'\rangle}$$

4. $$\frac{\langle S,s\rangle \xrightarrow{\lambda} s' \mid \underline{\text{abortion}}}{\langle S \ \underline{\text{except}}\ XC,s\rangle \xrightarrow{\lambda} \langle \underline{\text{skip}},s'\rangle \mid \underline{\text{abortion}}}$$

D-except 1. $$\frac{\langle S,s\rangle \xrightarrow{\lambda} \langle S',s'\rangle}{\langle \underline{\text{exception}}\ U;S\ ,s\rangle \xrightarrow{\lambda} \langle \underline{\text{exception}}\ U;S',s'\rangle}$$

2. $$\frac{\langle S,s\rangle \xrightarrow{\lambda} s' \mid \underline{\text{abortion}}}{\langle \underline{\text{exception}}\ U;S,s\rangle \xrightarrow{\lambda} s' \mid \underline{\text{abortion}}}$$

3. $$\frac{\langle S,s\rangle \xrightarrow{U'} \underline{\text{jump}},\ U' \neq U}{\langle \underline{\text{exception}}\ U;S,s\rangle \xrightarrow{U'} \underline{\text{jump}}}$$

The forms of the above transition rules are the same as those given in Ada.2.

tabort $$\langle \underline{\text{tabort}}\ T,s\rangle \xrightarrow{(*,T)!\underline{\text{ab}}} \langle \underline{\text{skip}},s\rangle$$

traise $$\langle \underline{\text{traise}}\ T,s\rangle \xrightarrow{(*,T)!\underline{\text{fl}}} \langle \underline{\text{skip}},s\rangle$$

The tabort rule means sending an abortion order to a visible task T. The traise rule models sending an order to raise a FAILURE exception in a visible task T; the configuration $\langle \underline{\text{skip}},s\rangle$ means that "the execution of this statement has no direct effect on the task issuing the statement" (see [DoD] section 11.6).

interrupt 1. $$\frac{S \in Bsc}{\langle S,s\rangle \xrightarrow{(*,T)?\underline{\text{ab}}} \langle \underline{\text{abort}},s\rangle}$$

2. $$\frac{S \in Bsc}{\langle S,s\rangle \xrightarrow{(*,T)?\underline{\text{fl}}} \langle \underline{\text{raise}}\ \text{FAILURE},s\rangle}$$

The interrupt rules mean that every basic statement has an

alternative meaning: receive an order from a task and then abort or raise a FAILURE exception. The interrupt rules model the requirement "For the task receiving the FAILURE exception, this exception is raised at the current point of execution" (see [DoD 80] section 11.6).

The interaction between exception and multitasking is indicated by the following rules:

initialization:

entrycall 1. $$\frac{[\![e]\!]_s = v,\ v \neq \underline{error}}{\langle T.W(e,x),s\rangle \xrightarrow{i(*,T.W)!v} \langle \underline{wait}(T.W,x),s\rangle}$$

2. $$\frac{[\![e]\!]_s = \underline{error}}{\langle T.W(e,x),s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

accept $$\langle \underline{accept}\ W(x,e)\ \underline{do}\ S,s\rangle \xrightarrow{i(*.W,T)?v} \langle S\ \underline{rendz}(T,W,e),s[v/x]\rangle$$

rendezvous

rendz 1. $$\frac{\langle S,s\rangle \xrightarrow{\lambda} \langle S',s'\rangle}{\langle S\ \underline{rendz}(T,W,e),s\rangle \xrightarrow{\lambda} \langle S'\underline{rendz}(T,W,e),s'\rangle}$$

2. $$\frac{\langle S,s\rangle \xrightarrow{\varepsilon} s'}{\langle S\ \underline{rendz}(T,W,e),s\rangle \xrightarrow{\varepsilon} \langle \underline{ack}(T,W,e),s'\rangle}$$

3. $$\frac{\langle S,s\rangle \xrightarrow{\varepsilon} \underline{abortion}}{\langle S\ \underline{rendz}(T,W,e),s\rangle \xrightarrow{\varepsilon} \langle \underline{ack}(T,W,\underline{er});\underline{abort},s\rangle}$$

4. $$\frac{\langle S,s\rangle \xrightarrow{U} \underline{jump}}{\langle S\ \underline{rendz}(T,W,e),s\rangle \xrightarrow{\varepsilon} \langle \underline{ack}(T,W,\underline{U});\underline{raise}\ U\ ,s\rangle}$$

termination

wait 1. $$\langle \underline{wait}(T.W,x),s\rangle \xrightarrow{f(*,T.W)?v} \langle \underline{skip},s[v/x]\rangle \quad \text{if } v \neq \underline{er}$$

2. $$\langle \underline{wait}(T.W,x),s\rangle \xrightarrow{f(*,T.W)?er} \langle \underline{raise}\ \text{T-ERROR},s\rangle$$

3. $\langle \underline{wait}(T.W,x),s\rangle \xrightarrow{f(*,T.W)?U} \langle \underline{raise}\ U,s\rangle$

4. $\langle \underline{wait}(T.W,x),s\rangle \xrightarrow{(*,T')?f1} \langle \underline{wait}(T.W,x);\underline{raise}\ FAILURE,s\rangle$

5. $\langle \underline{wait}(T.W,x),s\rangle \xrightarrow{(*,T')?ab} \langle \underline{wait}(T.W,x);\underline{abort},s\rangle$

ack 1. $$\frac{[\![e]\!]_s = v,\ v \neq \underline{error}}{\langle \underline{ack}(T,W,e),s\rangle \xrightarrow{f(*.W,T)!e} \langle \underline{skip},s\rangle}$$

2. $$\frac{[\![e]\!]_s = \underline{error}}{\langle \underline{ack}(T,W,e),s\rangle \xrightarrow{g} \underline{abortion}}$$

The accept rule says that an accept statement in the initialization phase of a rendezvous receives the value v from the task T through its entry W and then becomes a rendezvous statement S $\underline{rendz}W(T,W,e)$.

The rendz rules model the execution of the body of accept statements. Rule 2 says that if the execution of the body of an accept statement terminates normally then execute a normal acknowledgement statement. Rule 3 means that if the execution of the body of an accept statement terminates abnormally then the value $\underline{er}$ is sent to raise a T-ERROR exception in the calling task and the called task aborts. Rule 4 says that if an exception is raised inside an accept statement and not handled normally, then U is sent to the calling task to raise a U exception and the exception is propagated.

The wait rules model the termination phase of a rendezvous for the calling task. Rule 1 is the same as that in Ada.1 and models the normal termination of a rendezvous for the calling task. Rule 2 says

that if a wait statement receives the value er then a T-ERROR exception arises in the calling task. Rule 3 says that if the received value is an exception name U then the rendezvous is terminated and the exception U is raised.

By now we have seen that rendz rule 2 to 4, ack rule 1 and wait rule 2 and 3 model the requirement of the manual:

"A rendezvous can be terminated abnormally in two cases:

(a) When an exception is raised inside an accept statement and not handled normally. In this case, the exception is propagated both to the unit containing the accept statement, and to the calling task at the point of the entry call.

(b) When the task containing the accept statement is terminated abnormally. In this case, the exception TASKING-ERROR is raised in the calling task at the point of the entry call. (see [DoD 80] section 11.5)"

Finally, wait rule 4 and 5 mean that if a task receives the FAILURE exception or an abortion order during rendezvous then "the rendezvous is allowed to complete" and "the called task is unaffected" (see [DoD 80] section 11.6).

parallel 1. $$\frac{\langle MS_1, s\rangle \xrightarrow{\lambda} \langle MS_1', s'\rangle \mid s' \mid \underline{abortion}}{\langle MS_1 \| MS_2, s\rangle \xrightarrow{\lambda} \langle MS_1' \| MS_2, s'\rangle \mid \langle MS_2, s'\rangle \mid \underline{abortion}}$$

2. $$\frac{\langle MS_2, s\rangle \xrightarrow{\lambda} \langle MS_2', s'\rangle \mid s' \mid \underline{abortion}}{\langle MS_1 \| MS_2, s\rangle \xrightarrow{\lambda} \langle MS_1 \| MS_2', s'\rangle \mid \langle MS_1, s'\rangle \mid \underline{abortion}}$$

$$3.\ \frac{\langle MS_1, s\rangle \xrightarrow{\lambda} \langle MS_1', s'\rangle,\ \langle MS_2, s\rangle \xrightarrow{\bar{\lambda}} \langle MS_2', s\rangle}{\langle MS_1 \| MS_2, s\rangle \xrightarrow{\varepsilon} \langle MS_1' \| MS_2', s'\rangle}$$

$$4.\ \frac{\langle MS_1, s\rangle \xrightarrow{\lambda} \langle MS_1', s'\rangle,\ \langle MS_2, s\rangle \xrightarrow{\bar{\lambda}} \langle MS_2', s\rangle}{\langle MS_2 \| MS_1, s\rangle \xrightarrow{\varepsilon} \langle MS_2' \| MS_1', s'\rangle}$$

B-task $$1.\ \frac{\langle S, s\rangle \xrightarrow{\lambda} \langle S', s'\rangle | s' | \underline{\text{abortion}},\ \phi_{T,E}(\lambda)\downarrow}{\langle T::E;S, s\rangle \xrightarrow{\phi_{T,E}(\lambda)} \langle T::E;S', s'\rangle | s' | \underline{\text{abortion}}}$$

where $\phi_{T,E}: \Lambda_a \to \Lambda_a$ is the partial function defined by

$$\phi_T(\lambda) = \begin{cases} \varepsilon & \text{if } \lambda = \varepsilon \\ i(T,T'.W)! & \text{if } \lambda = i(*,T'.W)!v \quad T \neq T' \\ i(T.W,T')?v & \text{if } \lambda = i(*.W,T')?v \quad T \neq T' \\ f(T.W,T')!v & \text{if } \lambda = f(*.W,T')!v \quad T \neq T' \\ f(T,T'.W)?v & \text{if } \lambda = f(*,T'.W)?v \quad T \neq T' \\ (T,T')!v & \text{if } d = (*,T')!v \quad T \neq T' \\ (T,T')?v & \text{if } d = (*,T')?v \quad T \neq T' \end{cases}$$

where $N \varepsilon Tnm \cup \{*\}$, $T,\ T' \varepsilon Tnm$, $v \varepsilon V \cup Exn$.

$$2.\ \frac{\langle S, s\rangle \xrightarrow{U} \underline{\text{jump}}}{\langle T::E, S,\ s\rangle \xrightarrow{\varepsilon} \underline{\text{abortion}}}$$

Rule 1 is the same as that in Ada.1 with the extension of ϕ_T to cover the new labels. Rule 2 says that an attempt to propagate an exception beyond a task body results in abortion and no further propagation of the exception (see [DoD 80] section 11.4.1). Rule 2 ensures that the transition rules for parallel structure and task specification are exactly as those in Ada.1.

D-task $$\frac{\langle S, s\rangle \xrightarrow{\lambda} \langle S', s'\rangle | s' | \underline{\text{abortion}},\ \eta_T(\lambda)\downarrow}{\langle \underline{\text{task}}\ T::E;MS, s\rangle \xrightarrow{\eta_T(\lambda)} \langle \underline{\text{task}}\ T::E;S', s'\rangle | s' | \underline{\text{abortion}}}$$

where $\eta_T: \Lambda_a \rightarrow \Lambda_a$ is the partial function defined by

$$\eta_T(\lambda) = \begin{cases} \lambda & \text{if } \lambda = \varepsilon \\ \lambda & \text{if } \lambda = i(T', T''.W)!v \text{ and } T \neq T'' \\ \lambda & \text{if } \lambda = i(T'.W, T'')?v \text{ and } T \neq T', T \neq T'' \\ \lambda & \text{if } \lambda = f(T'.W, T'')!v \text{ and } T \neq T', T \neq T'' \\ \lambda & \text{if } \lambda = f(T', T''.W)?v \text{ and } T \neq T'' \\ d & \text{if } d = (T', T'')!v \text{ and } T \neq T'' \\ d & \text{if } d = (T', T'')?v \text{ and } T \neq T', T \neq T'' \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $T, T', T'' \in Tnm$, $v \in V \cup Exn$

Thus a semantics for Ada.3 has been given. In fact, we can prove that for the semantics of Ada.3 all lemmas and the theorem given in section 3.2.2 still hold.

Now let us examine the example given in section 3.1 using the semantics given in this subsection.

Example 3.8 Raising and propagating an exception during a rendezvous

In the following program task T2 will raise a FAILURE exception in task T1, which is in the middle of a rendezvous with task T. This FAILURE exception is not handled locally (by the accept statement body); therefore it is propagated in both T1 and T:

```
Pr is task T1::W1;(
      task T2::W2;(T1::W1;(accept W1(x1,y1) do
                                T2.W2(x1-1,z1);
                                y1:=x1+z1)except FAILURE→skip
            ||T2::W2;accept W2(x2,x2) do
                                select x2=0⇒traise T1
                                    or
                                       x2≠0⇒skip
                                end
            ||T::W; T1.W1(1,y) except FAILURE⇒y:=3))
```

A complete computation is:

$\langle Pr, s\rangle \xrightarrow[1]{s}$

```
task T1::W1;(
task T2::W2;(T1::W1;(T2.W2(x1-1,z1);y1:=x1+z1)rendz(T,W1,y1)
                    except (FAILURE⇒skip)                    ,x1=1
          ||T2::W2; ( the body of T2)
          ||T::W; wait(T1.W1,y) except FAILURE⇒y:=3))
```

$\xrightarrow[2]{s}$

```
task T1::W1;(
task T2::W2;(T1::W1;(wait(T2.W2,z1);y1:=x1+z1)rendz(T,W1,y1),x1=1
                    except (FAILURE⇒skip)                    x2=0
          ||T2::W2;(select (x2=0⇒traise T1) or (x2≠0⇒skip)
                          end)rendz (T1,W2,x2)
          ||T::W;wait(T1.W1,y) except FAILURE⇒y:=3))
```

$\xrightarrow[3]{\varepsilon}$

```
task T1::W1;(
task T2::W2;(T1::W1;(wait(T2.W2,z1);raise FAILURE;y1:=x1+z1),x1=1
                    rendz(T,W1,y1) except FAILURE⇒skip      x2=0
            ||T2::W2; (skip) rendz (T1,W2,x2)
            ||T::W; wait(T1.W1,y) except FAILURE⇒y:=3))
```

$\xrightarrow[4]{\varepsilon}$

```
task T1::W1;(
task T2::W2;(T1::W1;(wait(T2.W2,z1);raise FAILURE;y1:=x1+z1),x1=1
                    rendz(T,W1,y1) except FAILURE⇒skip      x2=0
            ||T2::W2;ack (T1,W2,x2)
            ||T::W; wait(T.W1,y) except FAILURE⇒y:=3))
```

$\xrightarrow[5]{\varepsilon}$

```
task T1::W1;(
task T2::W2;(T1::W1;(raise FAILURE; y1:=x1+z1)rendz(T,W1,y1),x1=1
                    except FAILURE⇒skip                     x2=0
            ||T2::W2; skip                                  z1=0
            ||T::W; wait(T1.W1,y) except FAILURE⇒y:=3))
```

$\xrightarrow[6]{\varepsilon}$

```
task T1::W1;(
task T2::W2;(T1::W1;(ack(T,W1,FAILURE); raise FAILURE)      ,x1=1
                    except FAILURE⇒skip                     x2=0
            ||T2::W2; skip                                  z1=0
            ||T::W; wait(T1.W1,y) except FAILURE⇒y:=3))
```

$\xrightarrow[7]{\varepsilon}$

```
task T1::W1;(
task T2::W2;(T1::W1;(raise FAILURE) except FAILURE⇒skip   x1=1
            ||T2::W2; skip                                ,x2=0
            ||T::W; raise FAILURE except FAILURE⇒y:=3))   z1=0
```

$\xrightarrow{\varepsilon^*}$

$$\left\{\begin{array}{ll}\underline{\text{task}}\ \text{T1::W1;(} & \text{x1=1}\\ \underline{\text{task}}\ \text{T2::W2;(T1::W1; }\underline{\text{skip}} & \text{,x2=0}\\ \qquad\qquad \|\text{T2::W2; }\underline{\text{skip}} & \text{z1=0}\\ \qquad\qquad \|\text{T::W; y:=3))} & \end{array}\right\}$$

$\xrightarrow{\varepsilon}$

$$\left\{\begin{array}{ll}\underline{\text{task}}\ \text{T1::W1;} & \text{x1=1}\\ \underline{\text{task}}\ \text{T2::W2;(T1::W1; }\underline{\text{skip}} & \text{,x2=0}\\ \qquad\qquad \|\text{T2::W2; }\underline{\text{skip}} & \text{z1=0}\\ \qquad\qquad \|\text{T::W; }\ \underline{\text{skip}}\text{))} & \text{y=3}\end{array}\right\}$$

$\xrightarrow{\varepsilon^*}$(x1=1,x2=0,z1=0,y=3)

Let us justify transition step 3. It is obtained by applying the D-task rule twice and by the multitask rule since the following two transition relations are valid:

a. $$\left\{\begin{array}{ll}\text{T1::W1;(}\underline{\text{wait}}\text{(T2.W2,z1); y1:=x1+z1)}\underline{\text{rendz}}\text{(T,W1,y1)} & \text{,s}\\ \qquad \underline{\text{except}}\ \text{FAILURE}\Rightarrow\underline{\text{skip}} & \end{array}\right\}$$

$\xrightarrow{\text{(T1,T2)?f1}}$

$$\left\{\begin{array}{ll}\text{T1::W1;(}\underline{\text{wait}}\text{(T2.W2,z1); }\underline{\text{raise}}\ \text{FAILURE; y1:=x1+z1)} & \text{,s}\\ \qquad \underline{\text{rendz}}\text{(T,W1,y1) }\underline{\text{except}}\ \text{FAILURE}\Rightarrow\underline{\text{skip}} & \end{array}\right\}$$

b . $$\left\{\begin{array}{ll}\text{T2::W2;(}\underline{\text{select}}\ \text{x2=0}\Rightarrow\underline{\text{traise}}\ \text{T1 or x2}\neq\text{0}\Rightarrow\underline{\text{skip}} & \text{,x2=0}\\ \qquad \underline{\text{end}}\text{)}\underline{\text{rendz}}\ \text{(T1,W2,x2)} & \end{array}\right\}$$

$\xrightarrow{\text{(T2,T1)!f1}}$

$\langle$T2::W2,(skip)rendz(T1,W2,x2) ,(x2=0)$\rangle$

Transition (a) is by the B-task rule since

$$\left\{\begin{array}{ll}\text{(}\underline{\text{wait}}\text{(T2.W2,z1); y1:=x1+z1)}\underline{\text{rendz}}\text{(T,W1,y1)} & \text{,s}\\ \underline{\text{except}}\ \text{FAILURE}\Rightarrow\underline{\text{skip}} & \end{array}\right\}$$

$\xrightarrow{\text{(*,T2)?f1}}$

$$\left\langle \begin{array}{l} (\underline{\text{wait}}(T2.W2,z1);\ \underline{\text{raise}}\ \text{FAILURE};\ y1:=x1+z1) \\ \underline{\text{rendz}}(T,W1,y1)\ \underline{\text{except}}\ \text{FAILURE} \Rightarrow \underline{\text{skip}} \end{array} ,s \right\rangle$$

and this is obtained from the S-except rule 3 since

$$\langle (\underline{\text{wait}}(T2.W2,z1);y1:=x1+z1)\ \underline{\text{rendz}}(T,W1,y1)\ ,s\rangle$$

$$\xrightarrow{(*,T2)?f1}$$

$$\left\langle \begin{array}{l} (\underline{\text{wait}}(T2.W2,z1);\ \underline{\text{raise}}\ \text{FAILURE};\ y1:=x1+z1) \\ \underline{\text{rendz}}(T,W1,y1) \end{array} ,s \right\rangle$$

This follows from the rendz rule 1 since

$$\langle \underline{\text{wait}}(T2.W2,z1);y1:=x1+z1\ ,s\rangle \xrightarrow{(*,T2)?f1}$$

$$\langle \underline{\text{wait}}(T2.W2,z1);\ \text{raise FAILURE};\ y1:=x1+z1\ ,s\rangle$$

which is by the composition rule since

$$\langle \underline{\text{wait}}(T2.W2,z1),s\rangle \xrightarrow{(*,T2)?f1} \langle \underline{\text{wait}}(T2.W2,z1);\underline{\text{raise}}\ \text{FAILURE}\ ,s\rangle$$

by the wait rule 3. Similarly, we can justify step (b) and the other transition steps. □

A criticism of the semantics may be that the motivation of the language designers was to introduce an asynchronous abortion and failure mechanism. Especially for the case of abortion the Ada manual says "An abort statement causes the unconditional asynchronous termination of the named task" (see [DoD 80] section 9.10). Our semantics given above is based on handshaking and is synchronous. For example, a tabort statement can be executed iff the named task is ready to receive the order to abort and this is neither unconditional nor asynchronous. The techniques used in section 2.5 may solve this asynchronous problem.

4. An operational semantics for Edison

The programming language Edison was invented by Brinch-Hansen ([Brinch-Hansen 81a]) for use both in teaching the principles of concurrent programming and in constructing reliable programs for multiprocessor systems. Edison was born after deep consideration of both successful and ill-fated experiences in the use and design of concurrent programming languages, especially Concurrent Pascal and Modula. The concepts of modularity, concurrency and synchronisation are separated in Edison by introducing (respectively) modules and procedures, concurrent statements, and when statements. This decision makes Edison simpler, more general and more flexible than Concurrent Pascal and Modula.

In contrast to CSP and Ada, communication in Edison is based on management of mutually exclusive access to shared variables (common data). This idea was developed independently by Brinch-Hansen ([Brinch-Hansen 73,75]) and Hoare ([Hoare 74]) as the monitor mechanism.

In general a system of communicating processes in Edison is obtained by properly arranging modules, procedures and when statements. Thus far it is the highest level design in this direction. To convince the reader of this let us consider a typical example --- a one-character buffer. In Concurrent Pascal (or Modula) this can be programmed as follows:

Example 4.1

```
type buffer
monitor
        var x: char; b: boolean;
        send-q, receive-q: queue;
        procedure entry send (c:char);
        begin  if b then delay(send-q);
                x:=c; b:=true;
                continue(receive-q)
        end;
        procedure entry receive(var c:char);
        begin  if not b then delay(receive-q);
                c:=x; b:=false;
                continue(send-q)
        end;
begin  b:=false end;
```

The shared variable x is a slot to store a character for exchanging, where the boolean b indicates whether the buffer is full or not. Two typed queue variables send-q and receive-q are used to delay the sending and receiving processes until the buffer is empty and full respectively.

The procedure send delays the calling process (if necessary) until the buffer is empty then puts the character (actual parameter) into x and activates the first receiving process waiting in the receive queue. The receive procedure is similar to the send procedure.

The body of the monitor sets the initial state of the buffer to empty. Since only one procedure must be performed at a time on the monitor variables the following rules apply to operations on queues:

When a procedure delays completion another operation can be performed by the other procedure. When a procedure activates a delayed procedure, the activating procedure automatically returns immediately after execution of the continue operation.

The above explanation shows that a monitor in Concurrent Pascal or Modula is an intricate combination of shared variables, procedures, process scheduling and modularity. Edison replaces this complicated scheduling by a simple statement for synchronization, the when statement. Thus, in Edison, a one-character buffer can be simply programmed as a module:

Example 4.2

```
module buffer
      var x: char; b: bool;
      *proc send(c: char)
       begin
            when not b do
                x:=c; b:=true
            end
       end
      *proc receive(var c: char)
       begin
            when b do
                c:=x; b:=false
            end
       end
begin b:=false end
```

Here x and b have the same meaning as in the previous example and the functions of the procedures send and receive are also the same.

The * preceding each proc declaration above indicates that the associated procedures are to be exported from the module. Note that Edison does not include the monitor concept and that there are no queues or queue operations here. The execution of all when statements will take place strictly one at a time. If several processes need to evaluate (re-evaluate) the guards simultaneously they will be able to do so one at a time. The when statement means:

1. Wait until no other process is executing the "body" of any other when statement. This is called the synchronisation phase.

2. Then evaluate the boolean expression b. If its value is true then execute the body of the when statement; otherwise execute this when statement again. This is called the critical phase.

The purpose of this chapter is to study the semantics of Edison. In section 4.1 the abstract syntax of a subset of Edison is given. We call this subset Edison.1. Since the primitive means of communication in Edison is by procedure call and modification of shared variables the declarations of variables, procedures and modules are introduced in the abstract syntax of Edison.1 as basic entities. In section 4.2 we discuss the static semantics of these declarations. And finally, in section 4.3 a structural operational semantics is given for Edison.1.

4.1 The syntax of Edison.1

The abstract syntax of Edison.1 is parameterised on the following disjoint sets:

Var - a countably infinite set of variables, ranged over by x.

Exp - a countably infinite set of expressions, ranged over by e.

Bexp - a countably infinite set of boolean expressions, ranged over by b.

Pnm - a countably infinite set of procedure names, ranged over by P.

Mnm - a countably infinite set of module names, ranged over by M.

The syntactic categories of guarded statements, statements and declarations are defined using a BNF-like notation as follows:

Gts - a set of Guarded statements, ranged over by GS and defined by:

GS ::= b⇒S | GS▯GS

Stm - a set of statements, ranged over by S and defined by:

S ::= skip | abort | x:=e | S;S | if GS fi | do GS od |
P(AP) | M.P(AP) | when GS end | S ‖ S

Dec - a set of declarations, ranged over by D and defined by:

D ::= empty | var x | proc P(FP) BS |
module M(EP) BS | D;D

Bstm - a set of block statements, ranged over by BS and defined by:

BS ::= D;S

Frm - a set of formal parameters, ranged over by FP and defined by:

FP ::= empty | val x | ref x | FP,FP

Act - a set of actual parameters, ranged over by AP and defined by:

AP ::= empty | e | loc x | AP,AP

Vis - a set of exported (visible) procedure heads, ranged over by EP and defined by:

EP ::= empty | P(FP) | EP;EP

Finally, a program of Edison is just a block statement.

Strictly speaking, the above syntax tells us that Edison.1 is only an "Edison-like" language. It differs from the original Edison in the following respects:

1. Types play an important part in Edison, but they are omitted in Edison.1 in order to focus our attention on parallelism and communication, as we did in previous chapters. Instead of using types we follow the traditional treatment of denotational semantics and use disjoint sets of variables, procedure names, module names and so on. In fact, types can be introduced directly into the abstract syntax and studied in the operational way without difficulties (see [Plotkin 81]).

2. In the Edison manual (see [Brinch-Hansen 81a]) the conditional statements are introduced as deterministic entities. Their abstract syntax should be:

CS ::= b do S | (b do S) else CS

Their execution consists of first evaluating the boolean conditions one at a time in the order written until one yielding the value true is found, whereupon the corresponding statement is executed, or until all the conditions have been found to be false. To simplify our later work of translating Ada to Edison see chapter 7) we use Dijkstra's guarded statements, as in CSP and Ada.1, as a replacement.

3. In a module of the original Edison the form of an exported declaration is a normal declaration with a prefix "*". For the sake of convenience in expressing the transition relations, we adopt the method used in Ada to describe the entries, listing all the exported procedure heads (procedure name together with formal parameters) after the module name in the module declaration. We only allow procedures to be exported entities of a module.

4. The original Edison allows (mutually) recursive procedures, but for simplicity in Edison.1 we allow non-recursive procedures only. In the original manual a formal parameter can be a procedure heading, but in Edison.1 a formal parameter can only be a value parameter (val x) or a variable parameter (ref x).

5. Finally, parallelism in Edison is achieved by concurrent statements, with the following abstract syntax:

PS ::= T do S | PS also PS

S ::= cobegin PS end

where PS is called a process statement and T is system-dependent (specifying e.g. that the process must be performed on a particular processor) In Edison a process is a dynamic concept and the use of

nested concurrent statements is forbidden. The manual says "if any of the concurrent processes reaches a concurrent statement the execution fails". Instead of this discipline we use a simple form S||S into Edison.1 and allow the use of nested concurrent statements.

4.2 Static semantics

Since several kinds of declarations are included in the syntax of Edison.1 the study of its static semantics becomes more interesting than those in CSP and Ada.1.

First of all we need for each formal or actual parameter π the sequence TY(π) containing the types occurring in π in the order written. TY(π) is defined by:

Formal parameters

	empty	val x	ref x	FP1,FP2
TY	$\emptyset$	val	ref	TY(FP1).TY(FP2)

Actual parameters

	empty	e	loc x	AP1,AP2
TY	$\emptyset$	val	ref	TY(AP1).TY(AP2)

For each syntactic entity Ω in Edison.1. We also need the following sets:

FV(Ω) is the set of free variables contained in Ω.

FPP(Ω) is a set of pairs. For each simple call statement P(AP) in Ω, FPP contains the pair (P,TY(AP)), consisting of the free procedure name together with the sequence of types occuring in AP.

FMP(Ω) is a set of triples. For each exported procedure call statement M.P(AP) in Ω, FMP contains the triple (M,P,TY(AP)).

For each declaration D we need the following sets:

DV(D) is the set of variable names defined by D.

DPP(D) is a set of pairs. For each procedure declaration proc P(FP) BS occurring in D, DPP contains the pair (P,TY(FP)).

DMP(D) is a set of triples. For each exported procedure head P(FP) of module M occurring in D, DMP contains the triple (M,P,TY(FP)).

We also need DV(FP), FV(AP) and DPP(EP) for each set FP of formal parameters, set AP of actual parameters and set EP of exported procedure heads. Finally, for a guarded statement GS we need the predicate:

Bool(GS) is the disjunction of guards occurring in GS.

All these sets and the predicate are defined inductively by the following tables:

Guarded statements

	b ⇒ S	GS1 ▯ GS2
FV	FV(b)∪FV(S)	FV(GS1)∪FV(GS2)
FPP	FPP(S)	FPP(GS1)∪FPP(GS2)
FMP	FMP(S)	FMP(GS1)∪FMP(GS2)
Bool	b	Bool(GS1)∨Bool(GS2)

Statements

	skip	abort	x:=e	P(AP)	M.P(AP)
FV	∅	∅	FV(e)∪{x}	FV(AP)	FV(AP)
FPP	∅	∅	∅	{(P,TY(AP))}	∅
FMP	∅	∅	∅	∅	{(M,P,TY(AP))}

	S1 ; S2	if GS fi	do GS od	when GS end	S1 ‖ S2
FV	FV(S1) ∪ FV(S2)	FV(GS)	FV(GS)	FV(GS)	FV(S1) ∪ FV(S1)
FPP	FPP(S1) ∪ FPP(S2)	FPP(GS)	FPP(GS)	FPP(GS)	FPP(S1) ∪ FPP(S2)
FMP	FMP(S1) ∪ FMP(S2)	FMP(GS)	FMP(GS)	FMP(GS)	FMP(S1) ∪ FMP(S2)

Declarations

	empty	var x	proc P(FP) BS	module M(EP) BS
DV	∅	{x}	∅	∅
FV	∅	∅	FV(BS)\DV(FP)	FV(BS)
DPP	∅	∅	{(P,TY(FP))}	∅
FPP	∅	∅	FPP(BS)	FPP(BS)
DMP	∅	∅	∅	{(M,K)|K∈DPP(EP)}
FMP	∅	∅	FMP(BS)	FMP(BS)

	D1;D2
DV	DV(D1) ∪ DV(D2)
FV	FV(D1) ∪ (FV(D2)\DV(D1))
DPP	DPP(D1) ∪ DPP(D2)
FPP	FPP(D1) ∪ (FPP(D2)\DPP(D1))
DMP	DMP(D1) ∪ DMP(D2)
FMP	FMP(D1) ∪ (FMP(D2)\DMP(D1))

Block statements

	D;S
DV	DV(D)
FV	FV(D) ∪ (FV(S)\DV(D))
DPP	DPP(D)
FPP	FPP(D) ∪ (FPP(S)\DPP(D))
DMP	DMP(D)
FMP	FMP(D) ∪ (FMP(S)\DMP(D))

Formal parameters

	empty	val x	ref x	FP1,FP2
DV	$\emptyset$	{x}	{x}	DV(FP1) $\cup$ DV(FP2)

Actual parameters

	empty	e	loc x	AP1,AP2
FV	$\emptyset$	FV(e)	{x}	FV(AP1) $\cup$ FV(AP2)

Exported entries

	empty	P(FP)	EP1;EP2
DPP	$\emptyset$	{(P,TY(FP))}	DPP(EP1) $\cup$ DPP(EP2)

To define the static semantics of Edison.1 the following notations are needed:

Notation 4.1

If D ε Dec, then $\mathbb{D}$(D) denotes the set of all variables, procedure names (associated with the types of their formal parameters) and exported procedure names (associated with the types of their formal parameters) defined by the declaration D:

$$\mathbb{D}(D) = DV(D) + DPP(D) + DMP(D)$$

where "+" denotes the disjoint union of sets (see [Gordon 79]). We define

$$\mathbb{D}_v = \{I \mid in_1(I) \varepsilon \mathbb{D}\}$$
$$\mathbb{D}_p = \{I \mid in_2(I) \varepsilon \mathbb{D}\}$$
$$\mathbb{D}_m = \{I \mid in_3(I) \varepsilon \mathbb{D}\}\ \square$$

The static semantics for a syntactic entity Ω is represented as:

$$\mathbb{D} \vdash \Omega \quad \text{or} \quad \vdash \Omega$$

The first form means that given a set $\mathbb{D}$, the syntactic entity Ω is valid; the second says that Ω is valid without any preconditions. Thus for guarded statements, statements, declarations, block statements and actual parameters we have:

$\mathbb{D} \vdash GS$ $\quad\quad$ $\mathbb{D} \vdash S$

$\mathbb{D} \vdash D$ $\quad\quad$ $\mathbb{D} \vdash BS$

$\mathbb{D} \vdash AP$

For formal parameters, exported entities and programs we have:

$\vdash FP$ $\quad\quad$ $\vdash EP$

$\vdash Prog$

The static semantics of Edison.1 is defined by the axioms and rules below:

Guarded statements

1. $\dfrac{\mathbb{D} \vdash S}{\mathbb{D} \vdash b \Rightarrow S}$ if $FV(b) \subseteq \mathbb{D}_V$

2. $\dfrac{\mathbb{D} \vdash GS1,\ \mathbb{D} \vdash GS2}{\mathbb{D} \vdash GS1 \ [] \ GS2}$

Rule 1 means that given $\mathbb{D}$ a guarded statement $b \Rightarrow S$ is valid if S is valid under $\mathbb{D}$ and all variables contained in the boolean expression b have already been declared. Rule 2 is similar to the corresponding rules in CSP and Ada.1.

Statements

1. $\mathbb{D} \vdash$ skip $\quad\quad$ 2. $\mathbb{D} \vdash$ abort

3. $\dfrac{\mathbb{D} \vdash S1,\ \mathbb{D} \vdash S2}{\mathbb{D} \vdash S1;S2}$ 4. $\dfrac{\mathbb{D} \vdash GS}{\mathbb{D} \vdash \underline{if}\ GS\ \underline{fi}}$

5. $\dfrac{\mathbb{D} \vdash GS}{\mathbb{D} \vdash \underline{do}\ GS\ \underline{od}}$ 6. $\dfrac{\mathbb{D} \vdash GS}{\mathbb{D} \vdash \underline{when}\ GS\ \underline{end}}$

7. $\dfrac{\mathbb{D} \vdash S1,\ \mathbb{D} \vdash S2}{\mathbb{D} \vdash S1 \parallel S2}$

8. $\mathbb{D} \vdash x:=e$ if $FV(e) \subseteq \mathbb{D}_v$, $x \in \mathbb{D}_v$

9. $\mathbb{D} \vdash P(AP)$ if $(P, TY(AP)) \in \mathbb{D}_p$, $FV(AP) \subseteq \mathbb{D}_v$

10. $\mathbb{D} \vdash M.P(AP)$ if $(M, P, TY(AP)) \in \mathbb{D}_m$, $FV(AP) \subseteq \mathbb{D}_v$

Rules 1 to 7 are easy to understand and their forms are similar to those in CSP and Ada.1. Rule 8 says an assignment statement x:=e is valid if both x and the free variables contained in the expression e have already been declared. Let us explain rule 10. It says that a procedure call M.P(AP) is valid if and only if the procedure P has been declared as an exported procedure by a declared module M, the free variables in the actual parameters AP have been declared, and the types and numbers of the actual parameters occurring in AP are exactly the same as the types and numbers of the formal parameters declared for P.

Declarations

1. $\mathbb{D} \vdash \underline{var}\ x$

2. $\dfrac{\vdash FP,\ \mathbb{D} \cup DV(FP) \vdash BS}{\mathbb{D} \vdash \underline{proc}\ P(FP)\ BS}$

$$3.\ \frac{\vdash EP,\ \mathbb{D} \vdash BS}{\mathbb{D} \vdash \underline{\text{module}}\ M(EP)\ BS} \qquad \text{if } DPP(EP) \subseteq DPP(BS)$$

$$4.\ \frac{\mathbb{D} \vdash D1,\ \mathbb{D} \cup \mathbb{D}(D1) \vdash D2}{\mathbb{D} \vdash D1;D2} \qquad \text{if } \mathbb{D}(D1) \cap \mathbb{D}(D2) = \emptyset$$

Rule 2 says a procedure declaration proc P(FP) BS is valid if and only if FP is valid and all variables, procedure and module names contained in BS are declared. Rule 3 means that a module declaration module M(EP) BS is valid if EP and BS are valid and all exported entities are declared in D. The condition $\mathbb{D}(D1) \cap \mathbb{D}(D2) = \emptyset$ in rule 4 means that a variable, procedure name or module name cannot be declared more than once in the same declaration area.

Block statements

$$1.\ \frac{\mathbb{D} \vdash D,\ \mathbb{D} \cup \mathbb{D}(D) \vdash S}{\mathbb{D} \vdash D;S}$$

Formal parameters

1. $\vdash$ empty 2. $\vdash$ val x

3. $\vdash$ ref x

$$4.\ \frac{\vdash FP1,\ \vdash FP2}{\vdash FP1;FP2} \qquad \text{if } DV(FP1) \cap DV(FP2) = \emptyset$$

Actual parameters

1. $\mathbb{D} \vdash$ empty 2. $\mathbb{D} \vdash e$ if $FV(e) \subseteq \mathbb{D}_V$

3. $\mathbb{D} \vdash \underline{\text{loc}}\ x$ if $x \in \mathbb{D}_v$

4. $\dfrac{\mathbb{D} \vdash AP1,\ \mathbb{D} \vdash AP2}{\mathbb{D} \vdash AP1, AP2}$

Exported procedure names

1. $\vdash \underline{\text{empty}}$ 2. $\vdash P(FP)$

3. $\dfrac{\vdash EP1,\ \vdash EP2}{\vdash EP1, EP2}$

Programs

1. $\dfrac{\emptyset \vdash D,\ \mathbb{D}(D) \vdash S}{\vdash D;S}$

The final rule means that if the declaration D is valid under the empty set and S is valid under the declaration then the program D;S is valid.

Let us now examine an example using this static semantics:

Example 4.3

Consider the typical communication mechanism in Edison.1 below (from example 4.2):

```
var x;
module M(send(val c),receive(ref c)),
      var x; var b;
      proc send(val c) when not b do x:=c; b:=true end;
      proc receive(ref c) when b do c:=x; b:=false end;
      b:=false;
M.send(5) || M.receive(loc x)
```

We are going to prove this program is valid. To do so let BS_m, BS_s and BS_r denote the bodies of the module M and the procedures send and receive respectively, and let D_m and D denote the declaration part of the module body and of the entire program respectively. According to the program rule we need to prove that the following two clauses are true:

a. $\emptyset \vdash D$

b. $\mathbb{D}(D) \vdash M.send(5) \parallel M.receive(\underline{loc}\ x)$

The first is true by declaration rule 4 since:

$\emptyset \vdash \underline{var}\ x$ by declaration rule 1 and

0. $\{x\}+\emptyset+\emptyset \vdash \underline{module}\ M(send,receive)\ BS_m$

the second follows by block statement rule and declaration rule 3 since

1. $\{x\}+\emptyset+\emptyset \vdash D_m$
2. $\{x\} \cup \mathbb{D}(D_m) \vdash b:=false$
3. $\{(send,\underline{val}), (receive,\underline{ref})\} \subseteq DPP(D_m)$

Clause 1 is true since

$\{x\}+\emptyset+\emptyset \vdash \underline{var}\ x$

$\{x\}+\emptyset+\emptyset \vdash \underline{var}\ b$

are true (by declaration rule 1) and

4. $\{x,b\}+\emptyset+\emptyset \vdash \underline{proc}\ send(\underline{val}\ c)\ BS_s;\ \underline{proc}\ receive(\underline{ref}\ c)\ BS_r$

by declaration rule 4 since

5. $\{x,b\}+\emptyset+\emptyset \vdash \underline{proc}\ send(\underline{val}\ c)\ BS_s$

and

6. $\{x,b\}+\{(send,\underline{val})\}+\emptyset \vdash \underline{proc}\ receive(\underline{ref}\ c)\ BS_r$

Clause 5 is true by declaration rule 2 since

7. $\vdash \underline{val}\ c$

by formal parameter rule 2 and

8. {x,b,c}+{(send,val)}+∅ ⊢ when not b do x:=c;b:=true end

Clause 8 follows from statement rules 3, 6 and 8 and guarded statement rule 1 since all free variables contained in the when statement are in {x,b,c}.

Similarly, clauses 2, 3 and 6 can be justified. We now have:

$$\mathbb{D}(D_m) = \{x,b\} + \{(send,val), (receive,ref)\} + \emptyset$$

and

$$\mathbb{D}(D) = \{x\} + \emptyset + \{(M,send,val), (M,receive,ref)\}$$

Finally, clause (b) is true by statement rule 7 since both M.send(5) and M.receive(loc x) are valid under $\mathbb{D}(D)$, by statement rule 10. □

4.3 Operational semantics

Successful research in the denotational approach tells us that to construct a semantics for a language including declarations of entities such as variables, procedures and modules, it is necessary to:

1. Distinguish the denotable values (locations, procedure and module abstractions) from the storable values (truth values, natural numbers and files).

2. Separate the concepts of store (mapping locations to storable values) and environment (mapping identifiers to denotable values).

For example, the elaboration of a declaration of variable x produces a new environment which maps x to a new location. The location can be thought of as an abstract address; note that we do not really want to commit ourselves to any machine architecture, but

only to the necessary logical properties. Similarly, a declaration of a procedure (or a module) maps the procedure name to a denotable procedure (or module) value. This is called a procedure abstraction (or respectively, a module abstraction). In general, the execution of a statement changes the store but not the environment, and the elaboration of a declaration changes the environment (and possibly the store). This approach provides a secure foundation for handling problems such as static binding (storage sharing), call-by-reference (aliasing problems), arrays (location expressions) and reference types.

In this section an operational semantics of Edison.1 is given using the above concepts. First of all, we introduce the following sets and functions:

Ide - the set of variables, procedure and module names, i.e. Ide=Var∪Pnm∪Mnm. These sets are disjoint. Let I range over Ide.

V - a given countably infinite set of storable values, ranged over by v and assumed to contain integers, truth values and character strings.

Loc - a set of locations, ranged over by a.

Pabs - a set of procedure abstractions (see [Tennent 81]), defined by:

$$Pabs = \{ \lambda FP.\ BS \mid BS \varepsilon Bstm,\ FP \varepsilon Frm \}$$

Mabs - a set of module abstractions, defined by:

$$Mabs = Env$$

Dval - a set of denotable values, ranged over by μ and defined by

$$Dval = Loc + Pabs + Mabs$$

Env - the set of environments, ranged over by ρ and defined by:

$$Env = \{\ \rho \varepsilon Ide \rightarrow Dval \mid \rho(Var) \underline{\subseteq} Loc,\ \rho(Pnm) \underline{\subseteq} Pabs,\ \rho(Mnm) \underline{\subseteq} Mabs\}$$

Note this is a recursive definition of Env. However this does not lead any problems, for details see [Plotkin 81]. We use $\mathbb{D}(\rho)$ to denote the argument domain of an environment ρ. For later discussion on environments some standard notation is necessary (see [Gordon 79]):

Notation 4.2 Store

A store s is a function s:Loc $\rightarrow$V; Stores is the set of all stores. A function new:Stores $\rightarrow$Loc is defined by:

for any sεStores s(new s) is unused $\square$

Notation 4.3 Environments

If ρ_1, ρ_2 ε Env and I ε Ide, then $\rho_1[\rho_2]$ ε Env is defined by:

$$\rho_1[\rho_2](I) = \begin{cases} \rho_1(I) & \text{if } \rho_2(I) \text{ is undefined} \\ \rho_2(I) & \text{if } \rho_2(I) \text{ is defined} \end{cases}$$

If μ_i ε Dval, I_i ε Ide, i=1,...,n, then $[\mu_1,\ldots,\mu_n/I_1,\ldots,I_n]$ ε Env denotes the "little" environment ρ defined by:

$$\rho(I) = \begin{cases} \mu_i & \text{if } I = I_i, \text{ for some } i=1,\ldots,n \\ \text{undefined} & \text{otherwise} \end{cases}$$

For xεVar let {x=a} be the little environment defined by:

$$\{x=a\}(I)=\begin{cases} a & \text{if } I=x \\ \\ \text{undefined} & \text{otherwise} \end{cases}$$

In order to describe the behaviours of the syntactic entities of Edison.1, we need to add the following extra syntactic clauses:

D ::= ... | ρ | FP=/.P

S ::= ... | crit S end

BS ::= ... | body::BS

Here FP=AP denotes the substitution of formal parameters by actual parameters; body::BS is a procedure body and denotes the execution of a procedure; crit S end is called a critical statement and denotes the execution of the critical phase of a when statement.

The static semantics of these new syntactic entities are given below:

$$\mathbb{D} \vdash \rho$$

$$\frac{\vdash FP,\ \mathbb{D} \vdash AP}{\mathbb{D} \vdash FP=AP} \quad \text{if } TY(FP)=TY(AP)$$

$$\frac{\mathbb{D} \vdash S}{\mathbb{D} \vdash \text{crit } S \text{ end}}$$

$$\frac{\mathbb{D} \vdash BS}{\mathbb{D} \vdash \text{body::BS}}$$

Finally, the operational semantics of Edison.1 is given by the following labelled transition systems:

$$\mathbb{T}_{gs}=\langle \Gamma_{gs}, T_{gs}, \Lambda_{gs}, \xrightarrow{gs} \rangle$$
$$\mathbb{T}_{s} =\langle \Gamma_{s}, T_{s}, \Lambda_{s}, \xrightarrow{s} \rangle$$
$$\mathbb{T}_{d} =\langle \Gamma_{d}, T_{d}, \Lambda_{d}, \xrightarrow{d} \rangle$$
$$\mathbb{T}_{bs}=\langle \Gamma_{bs}, T_{bs}, \Lambda_{bs}, \xrightarrow{bs} \rangle$$

where the sets of configurations are defined by:

$$\Gamma_{gs}=\{\langle GS,\rho,s\rangle \mid GS \in Gts,\ \rho \in Env,\ s \in Stores\} \cup \Gamma_{s}$$
$$T_{gs} =\Gamma_{s}$$

$$\Gamma_{s} =\{\langle S,\rho,s\rangle \mid S \in Stm,\ \rho \in Env,\ s \in Stores\} \cup T_{s}$$
$$T_{s} =\{\langle \rho,s\rangle \mid \rho \in Env,\ s \in Stores\} \cup \{\underline{abortion}\}$$

$$\Gamma_{d} =\{\langle D,\rho,s\rangle \mid D \in Dec,\ \rho \in Env,\ s \in Stores\} \cup T_{s}$$
$$T_{d} =T_{s}$$

$$\Gamma_{bs}=\{\langle BS,\rho,s\rangle \mid BS \in Bstm,\ \rho \in Env,\ s \in Stores\} \cup T_{s}$$
$$T_{bs}=T_{s}$$

In general, given a configuration $\langle \Omega,\rho,s\rangle$, Ω denotes the current syntactic entity to be executed and ρ and s denote the current environment and store. A configuration $\langle \rho,s\rangle$ stands for a normal termination where ρ and s denote the environment and store at termination. As in CSP and Ada.1, abortion denotes an abnormal termination.

The set of transition labels (actions) is

$$\Lambda_{gs}=\Lambda_{s}=\Lambda_{d}=\Lambda_{bs}=\{\varepsilon\}.$$

This is much simpler than in CSP and Ada.1 since communication in Edison.1 is not achieved by handshaking but by mutually exclusive access to shared variables. We will see this later. In fact we can omit the action ε since it is the only action in the above transition systems, but here we prefer to have it because it may be useful for proving properties of the semantics.

Finally, we define the transition relations in the above systems. As usual, we will omit the gs, s, d, and bs under the arrows when there is no confusion in the context.

In general, given a syntactic entity Ω a transition relation in the above transition systems takes one of the following three forms:

a. $\langle \Omega,\rho,s\rangle \xrightarrow{\varepsilon} \langle \Omega',\rho',s'\rangle$

meaning that given an environment ρ and a store s, the execution of Ω produces a new configuration where Ω becomes Ω', and ρ, s are changed to ρ' and s' respectively.

b. $\langle \Omega,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho',s'\rangle$

meaning that given an environment ρ and a store s the execution of Ω results in a normal termination $\langle \rho',s'\rangle$.

c. $\langle \Omega,\rho,s\rangle \xrightarrow{\varepsilon}$ abortion

meaning that given an environment ρ and a store s the execution of Ω leads to an abnormal termination.

As usual we assume that any expression $e \in Exp$ (or boolean expression $b \in Bexp$) can be evaluated properly and results in either a value $v \in V$ or an error. The form $[\![e]\!]_{\rho s}$ denotes the value of a expression e under the environment ρ and the store s. A predefined boolean variable $\tilde{x}$ (different from all other identifiers) is introduced to implement the control of mutually exclusive access to shared variables; the initial value of $\tilde{x}$ is assumed to be ff.

All the transition relations (rules) of $\xrightarrow[o]{}$ are given below:

Guarded statements

Guards 1. $$\frac{[\![b]\!]_{\rho s} = tt,\ \langle S,\rho,s\rangle \xrightarrow{g} r}{\langle b \Rightarrow S,\rho,s\rangle \xrightarrow{g} r}$$

2. $$\frac{[\![b]\!]_{\rho s} = \underline{error}}{\langle b \Rightarrow S,\rho,s\rangle \xrightarrow{g} \underline{abortion}}$$

Alternative

1. $$\frac{\langle GS1,\rho,s\rangle \xrightarrow{g} \langle S1,\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \underline{abortion}}{\langle GS1 \,[]\, GS2,\rho,s\rangle \xrightarrow{g} \langle S1,\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \underline{abortion}}$$

2. $$\frac{\langle GS2,\rho,s\rangle \xrightarrow{g} \langle S2,\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \underline{abortion}}{\langle GS1 \,[]\, GS2,\rho,s\rangle \xrightarrow{g} \langle S2,\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \underline{abortion}}$$

The rules for the guarded statements are similar to those in CSP and Ada.1.

Statements

skip $$\langle \underline{skip},\rho,s\rangle \xrightarrow{g} \langle \rho,s\rangle$$

abort $$\langle \underline{abort},\rho,s\rangle \xrightarrow{g} \underline{abortion}$$

assign 1. $$\frac{[\![e]\!]_{\rho s} = v}{\langle x:=e,\rho,s\rangle \xrightarrow{g} \langle \underline{skip},\rho,s[v/\rho(x)]\rangle}$$

2. $$\frac{[\![e]\!]_{\rho s} = \underline{error}}{\langle x:=e,\rho,s\rangle \xrightarrow{g} \underline{abortion}}$$

S-composition

$$\frac{\langle S_1,\rho,s\rangle \xrightarrow{g} \langle S_1',\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \underline{abortion}}{\langle S_1;S_2,\rho,s\rangle \xrightarrow{g} \langle S_1';S_2,\rho',s'\rangle \mid \langle S_2,\rho',s'\rangle \mid \underline{abortion}}$$

condition 1. $$\frac{\langle GS,\rho,s\rangle \xrightarrow{\varepsilon} \langle S,\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \underline{abortion}}{\langle \underline{if}\ GS\ \underline{fi},\rho,s\rangle \xrightarrow{\varepsilon} \langle S,\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \underline{abortion}}$$

2. $$\frac{[\![Bool(GS)]\!]_{\rho s}=ff}{\langle \underline{if}\ GS\ \underline{fi},\rho,s\rangle \xrightarrow{\varepsilon} \underline{abortion}}$$

repetitive 1. $$\frac{\langle GS,\rho,s\rangle \xrightarrow{\varepsilon} \langle S,\rho',s'\rangle}{\langle \underline{do}\ GS\ \underline{od},\rho',s\rangle \xrightarrow{\varepsilon} \langle S;\underline{do}\ GS\ \underline{od},\rho',s'\rangle}$$

2. $$\frac{\langle GS,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho',s'\rangle \mid \underline{abortion}}{\langle \underline{do}\ GS\ \underline{od},\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{do}\ GS\ \underline{od},\rho',s'\rangle \mid \underline{abortion}}$$

3. $$\frac{[\![Bool(GS)]\!]_{\rho s}=ff}{\langle \underline{do}\ GS\ \underline{od},\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho,s\rangle}$$

when 1. $$\frac{[\![\tilde{x}]\!]_{\rho s}=ff,\ \langle GS,\rho,s\rangle \xrightarrow{\varepsilon} \langle S,\rho',s'\rangle}{\langle \underline{when}\ GS\ \underline{end},\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{crit}\ S\ \underline{end},\rho',s'[tt/\rho(\tilde{x})]\rangle}$$

2. $$\frac{[\![\tilde{x}]\!]_{\rho s}=ff,\ \langle GS,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho',s'\rangle \mid \underline{abortion}}{\langle \underline{when}\ GS\ \underline{end},\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho',s'\rangle \mid \underline{abortion}}$$

crit 1. $$\frac{\langle S,\rho,s\rangle \xrightarrow{\varepsilon} \langle S',\rho',s'\rangle}{\langle \underline{crit}\ S\ \underline{end},\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{crit}\ S'\ \underline{end},\rho',s'\rangle}$$

2. $$\frac{\langle S,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho',s'\rangle \mid \underline{abortion}}{\langle \underline{crit}\ S\ \underline{end},\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{skip},\rho',s'[ff/\rho(\tilde{x})]\rangle \mid \underline{abortion}}$$

When rule 1 means that if no other process is executing the critical phase of a *when* statement (the value of $\tilde{x}$ in the store s is ff) and at least one guard of the guarded statement in this *when* statement is satisfied, then this *when* statement gains exclusive access (the value of $\tilde{x}$ in the store s' is changed to tt) and enters its critical phase (the corresponding crit statement). When rule 2 deals with skip or abort actions of guarded statements. The crit rules model the execution of the critical phase of a *when* statement. For example, crit rule 2 says that if the execution of the body of a

critical statement terminates normally then the execution of this critical statement is finished and exclusive access is given up (the value of $\tilde{x}$ in the store s' is changed to ff); if the execution of the body of the critical statement aborts then so does the critical statement.

concurrent 1. $$\frac{\langle S_1,\rho,s\rangle \xrightarrow{\varepsilon} \langle S_1',\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \underline{\text{abortion}}}{\langle S_1 \| S_2,\rho,s\rangle \xrightarrow{\varepsilon} \langle S_1' \| S_2,\rho',s'\rangle \mid \langle S_2,\rho',s'\rangle \mid \underline{\text{abortion}}}$$

2. $$\frac{\langle S_2,\rho,s\rangle \xrightarrow{\varepsilon} \langle S_2',\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \underline{\text{abortion}}}{\langle S_1 \| S_2,\rho,s\rangle \xrightarrow{\varepsilon} \langle S_1 \| S_2',\rho',s'\rangle \mid \langle S_1,\rho',s'\rangle \mid \underline{\text{abortion}}}$$

The concurrent rules are very much simpler than those in the semantics of CSP and Ada.1 since there is no handshake communication between concurrent processes in Edison. The rules mean that a concurrent statement is executed by interleaving the execution of its component processes.

call 1. $\langle P(AP),\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{\text{body}}::FP=AP;BS,\ \rho,s\rangle$

if $\rho(P) = \lambda$ FP. BS

2. $\langle M.P(AP),\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{\text{body}}::FP=AP;BS,\ \rho,s\rangle$

if $\rho(M)(P)=\lambda$ FP. BS

We now give the transitions for declarations:

Declaration

empty $\langle \underline{\text{empty}},\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho,s\rangle$

D-var $\langle \underline{\text{var}}\ x,\rho,s\rangle \xrightarrow{\varepsilon} \langle \{x=new(s)\},\rho,s[\underline{\text{undefined}}/new(s)]\rangle$

D-proc $\quad \langle \underline{\text{proc}}\ P(FP)\ BS,\ \rho, s\rangle \xrightarrow{\varepsilon} \langle \{P=\lambda FP.\rho_p; BS\},\ \rho, s\rangle$

$$\rho_p(I) = \begin{cases} \text{undefined} & \text{if } I \text{ is DV}(FP) \\ \rho(I) & \text{otherwise} \end{cases}$$

The D-proc rule means that the elaboration of a procedure declaration forms a little environment $\{P=\lambda FP.\rho_p; BS\}$. All free variables, procedure names and module names are bound to their values at procedure-declaration time, i.e. only static binding (where bindings of all free variables, procedure names and module names are determined by their textual occurrences) is considered in Edison.1.

D-module 1. $$\frac{\langle BS,\rho,s\rangle \xrightarrow{\varepsilon} \langle BS',\rho',s'\rangle}{\langle \underline{\text{module}}\ M(EP)\ BS,\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{\text{module}}\ M(EP)\ BS',\rho',s'\rangle}$$

2. $$\frac{\langle BS,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho',s'\rangle \mid \text{abortion}}{\langle \underline{\text{module}}\ M(EP)\ BS,\rho,s\rangle \xrightarrow{\varepsilon} \langle \{M=\rho'\},\rho,s'\rangle \mid \underline{\text{abortion}}}$$

Rule 1 says that all local and exported entities declared in M are created one at a time in the order written and then the body of that module is executed. Rule 2 says that when the elaboration of a module declaration terminates normally a new module abstraction is created; otherwise the elaboration of the module results in abortion. These two rules model the following requirement from the Edison manual:

"The initialization of a module M takes place in three steps:

1. The local and exported entities declared in M are created and added to the current context.
2. The modules declared in M are initialized one at a time in the order written.
3. The initial operation of M is performed."

D-composition

$$1.\ \frac{\langle D_1,\rho,s\rangle \xrightarrow{\varepsilon} \langle D_1',\rho,s\rangle \mid \underline{\text{abortion}}}{\langle D_1;D_2,\rho,s\rangle \xrightarrow{\varepsilon} \langle D_1';D_2,\rho,s\rangle \mid \underline{\text{abortion}}}$$

$$2.\ \frac{\langle D,\rho[\rho_0],s\rangle \xrightarrow{\varepsilon} \langle D',\rho[\rho_0],s'\rangle \mid \underline{\text{abortion}}}{\langle \rho_0;D,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho_0;D',\rho,s'\rangle \mid \underline{\text{abortion}}}$$

$$3.\ \langle \rho_1;\rho_2,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho_1[\rho_2],\rho,s\rangle$$

$$4.\ \langle \rho_1,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho[\rho_1],s\rangle$$

Parameter bindings

$$1.\ \frac{[\![e]\!]_{\rho s} = v}{\langle \underline{\text{val}}\ x = e,\rho,s\rangle \xrightarrow{\varepsilon} \langle \{x=\text{new}(s)\},\ \rho,s[v/\text{new}(s)]\rangle}$$

$$2.\ \frac{[\![e]\!]_{\rho s} = \underline{\text{error}}}{\langle \underline{\text{val}}\ x=e,\rho,s\rangle \xrightarrow{\varepsilon} \underline{\text{abortion}}}$$

$$3.\ \langle \underline{\text{ref}}\ x = \underline{\text{loc}}\ y,\ \rho,s\rangle \xrightarrow{\varepsilon} \langle \{x=\rho(y)\},\rho,s\rangle$$

$$4.\ \frac{\langle FP_1=AP_1,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho_1,\rho,s'\rangle \mid \underline{\text{abortion}}}{\langle (FP_1,FP_2=AP_1,AP_2),\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho_1;FP_2=AP_2,\rho,s'\rangle \mid \underline{\text{abortion}}}$$

Parameter rule 1 models a call-by-value binding mechanism. It says that an actual parameter gives a new (little) environment $\{x=\text{new}(s)\}$ and a new store where the (new) location new(s) is

assigned the value obtained by evaluating the actual expression e under the environment ρ and store s. Rule 3 models a call-by-reference binding mechanism, binding the formal parameter in the environment to the location in the store occupied by the actual parameter. Rule 4 says that the bindings of the parameters of a procedure are created one at a time in the order listed.

Finally, we define transitions for block statements:

Block statements

$$1.\ \frac{\langle D_1,\rho,s\rangle \xrightarrow{\varepsilon} \langle D_1',\rho,s\rangle \mid \underline{\text{abortion}}}{\langle D_1;S_2,\rho,s\rangle \xrightarrow{\varepsilon} \langle D_1';S_2,\rho,s\rangle \mid \underline{\text{abortion}}}$$

$$2.\ \frac{\langle S,\rho[\rho_0],s\rangle \xrightarrow{\varepsilon} \langle S',\rho[\rho_0],s'\rangle \mid \langle \rho[\rho_0],s'\rangle \mid \underline{\text{abortion}}}{\langle \rho_0;S,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho_0;S',\rho,s'\rangle \mid \langle \rho[\rho_0],s'\rangle \mid \underline{\text{abortion}}}$$

procedure body

$$\frac{\langle BS,\rho,s\rangle \xrightarrow{\varepsilon} \langle BS',\rho',s'\rangle \mid \langle \rho',s'\rangle \mid \text{abortion}}{\langle \underline{\text{body}}::BS,\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{\text{body}}::BS',\rho',s'\rangle \mid \langle \rho,s'\rangle \mid \underline{\text{abortion}}}$$

Note that the second case says that after a procedure call the environment is restored to what it was before the call.

Now we have given an operational semantics for Edison.1.

It should be mentioned that the D-composition and block statement rules are similar to those given in [Plotkin 81]. Let us work through two simple examples to see how these rules work. For simplicity, in the following examples we will use $\{x=a_x\}$ to denote a little environment in which we assign an unused location a_x to x, and will omit the action ε.

Example 4.4

Given the initial environment ρ and store s. Let

$$s'=s[\underline{undefined}/a_x] \quad \text{and} \quad s''=s'[\underline{undefined}/a_b].$$

Consider the elaboration of a declaration as follows:

$\langle \underline{var}\ x; \underline{var}\ b; \underline{proc}\ P(FP)\ BS, \rho, s\rangle$

$\xrightarrow[1]{} \langle \{x=a_x\}; \underline{var}\ b; \underline{proc}\ P(FP)\ BS, \rho, s'\rangle$

$\xrightarrow[2]{} \langle \{x=a_x\}; \{b=a_b\}; \underline{proc}\ P(FP)\ BS, \rho, s''\rangle$

$\xrightarrow[3]{} \langle \{x=a_x\}[\{b=a_b\}]; \underline{proc}\ P(FP)\ BS, \rho, s''\rangle$

$\xrightarrow[4]{} \langle \{x=a_x, b=a_b\}; \{P=\lambda FP.BS\}, \rho, s''\rangle$

$\xrightarrow[5]{} \langle \{x=a_x, b=a_b\}[\{P=\lambda FP.BS\}], \rho, s''\rangle$

$\xrightarrow[6]{} \langle \rho[\{x=a_x, b=a_b, P=\lambda FP.BS\}], s''\rangle$

It is easy to see that step 1 is by D-composition rule 1 since

$$\langle \underline{var}\ x, \rho, s\rangle \longrightarrow \langle \{x=a_x\}, \rho, s'\rangle$$

Step 2 is by D-composition rules 1 and 2 since

$$\langle \underline{var}\ b, \rho[\{x=a_x\}], s'\rangle \longrightarrow \langle \{b=a_b\}, \rho[\{a=a_x\}], s''\rangle$$

and step 3 is by D-composition rule 2 since

$$\langle \{x=a_x\}; \{b=a_b\}, \rho, s''\rangle \longrightarrow \langle \{x=a_x\}[\{b=a_b\}], \rho, s''\rangle$$

by D-composition rule 3. Similarly we can examine transition steps 4 to 6. □

When we understand the rules for declarations the block statement rules become very easy to comprehend. Let us consider the following example.

Example 4.5

Consider the computation of the block statement below:

$\langle \underline{\text{var}}\ x; x:=1, \rho, s\rangle$

$\xrightarrow[1]{} \langle \{x=a_x\}; x:=1, \rho, s[\underline{\text{undefined}}/a_x]\rangle$

$\xrightarrow[2]{} \langle \{x=a_x\}; \underline{\text{skip}}, \rho, s[1/a_x]\rangle$

$\xrightarrow[3]{} \langle \rho[\{x=a_x\}], s[1/a_x]\rangle$

Step 1 is by block rule 1 since
$\langle \underline{\text{var}}\ x, \rho, s\rangle \longrightarrow \langle \{x=a_x\}, \rho, s[\underline{\text{undefined}}/a_x]\rangle$

Step 2 is by the first case of block statement rule 2 since
$\langle x:=1, \rho[\{x=a_x\}], s[\underline{\text{undefined}}/a_x]\rangle \longrightarrow \langle \underline{\text{skip}}, \rho, s[1/a_x]\rangle$.

Step 3 is by the second case of block rule 2 since
$\langle \underline{\text{skip}}, \rho[\{x=a_x\}], s[1/a_x]\rangle \longrightarrow \langle \rho[\{x=a_x\}], s[1/a_x]\rangle$. □

As in the case of CSP and Ada, we can prove the following properties of this semantics:

Lemma 4.1

1. Let Ω, $\Omega' \in Gts \cup Stm$.
   If $\langle \Omega, \rho, s\rangle \xrightarrow{\varepsilon} \langle \Omega', \rho', s'\rangle$ or $\langle \Omega, \rho, s\rangle \xrightarrow{\varepsilon} \langle \rho', s'\rangle$ then $\rho' = \rho$.

2. Let Ω, $\Omega' \in Dec \cup Bstm$.
   If $\langle \Omega, \rho, s\rangle \xrightarrow{\varepsilon} \langle \Omega', \rho', s'\rangle$ then $\rho' = \rho$.

Lemma 4.2

a. If $\Omega \in Gts \cup Stm$ and $\langle \Omega, \rho, s\rangle \xrightarrow{\varepsilon} \langle \Omega', \rho, s'\rangle$ then $\Omega' \in Stm$ and
   $FV(\Omega') \subseteq FV(\Omega)$ and $FPP(\Omega') \subseteq FPP(\Omega)$ and $FMP(\Omega') \subseteq FMP(\Omega)$

b. If $\langle D,\rho,s\rangle \xrightarrow{\varepsilon} \langle D',\rho,s'\rangle$ then

$FV(D') \subseteq FV(D)$ and $FPP(D') \subseteq FPP(D)$ and $FMP(D') \subseteq FMP(D)$

$DV(D') \subseteq DV(D)$ and $DPP(D') \subseteq DPP(D)$ and $DMP(D') \subseteq DMP(D)$

c. If $\langle BS,\rho,s\rangle \xrightarrow{\varepsilon} \langle BS',\rho,s'\rangle$ then

$FV(BS') \subseteq FV(BS)$ and $FPP(BS') \subseteq FPP(BS)$ and $FMP(BS') \subseteq FMP(BS)$

$DV(BS') \subseteq DV(BS)$ and $DPP(BS') \subseteq DPP(BS)$ and $DMP(BS') \subseteq DMP(BS)$

Theorem 4.1

a. Let $\Omega \in Gts \cup Stm$. If $\mathbb{D}(\rho) \vdash \Omega$ and $\langle \Omega,\rho,s\rangle \xrightarrow{\varepsilon} r$, then either $r=\langle S,\rho,s'\rangle$ $S \in Stm$ and $\mathbb{D}(\rho) \vdash S$ or r is one of $\langle \rho,s'\rangle$ or abortion.

b. If $\mathbb{D}(\rho) \vdash D$ and $\langle D,\rho,s\rangle \xrightarrow{\varepsilon} r$, then either $r=\langle D',\rho,s'\rangle$ and $\mathbb{D}(\rho) \vdash D'$ or r is one of $\langle \rho',s'\rangle$ or abortion.

c. If $\mathbb{D}(\rho) \vdash BS$ and $\langle BS,\rho,s\rangle \xrightarrow{\varepsilon} r$, then either $r=\langle BS',\rho,s'\rangle$ and $\mathbb{D}(\rho) \vdash BS'$ or r is one of $\langle \rho',s'\rangle$ or abortion.

Let us examine the example given in section 4.1 using the above semantics.

Example 4.6

Let $\{a_x=v\}$ denote a little store which means assign a value v to a_x.

Firstly, the elaboration of the local declaration contained in the module body is:

$$\left\{\begin{array}{l} \underline{\text{var}}\ x;\ \underline{\text{var}}\ b; \\ \underline{\text{proc}}\ \text{send}(\underline{\text{val}}\ c), BS_s; \\ \underline{\text{proc}}\ \text{receive}(\underline{\text{ref}}\ c), BS_r; \end{array}\right. , \{\tilde{x}=a_{\tilde{x}}, x=a_x\}, \{a_{\tilde{x}}=ff\}$$

$$\xrightarrow{*}\left|\begin{array}{l}\{x=a'_x, b=a_b\};\\ \underline{\text{proc}}\ \text{send}(\underline{\text{val}}\ c), BS_s;\\ \underline{\text{proc}}\ \text{receive}(\underline{\text{ref}}\ c), BS_r;\end{array}\right.\quad\left.\begin{array}{l}\\ ,\{\tilde{x}=a_{\tilde{x}}, x=a_x\},\ \{a_{\tilde{x}}=ff\}\\ \end{array}\right|$$

(these transitions are by the D-composition rule and the D-var rule)

$$\xrightarrow{*}\left|\begin{array}{l}\{x=a'_x, b=a_b,\\ \text{send}=\lambda\ \underline{\text{val}}\ c.\ \{\tilde{x}=a_{\tilde{x}}, x=a'_x, b=a_b\}, BS_s,\\ \text{receive}=\lambda\ \underline{\text{ref}}\ c.\ \{\tilde{x}=a_{\tilde{x}}, x=a'_x, b=a_b, \text{send}=---\}, BS_r\}\end{array}\right.\quad\left.\begin{array}{l}\\ ,\{\tilde{x}=a_{\tilde{x}}, x=a_x\},\ (a_{\tilde{x}}=ff)\\ \end{array}\right|$$

$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\rho'_m}$

This is by the D-proc and D-composition rules. Let ρ'_m denote the elaboration of the declarations contained in the module M. The execution of the initial statement of the module is:

$$\langle b:=\underline{\text{false}},\ \{\tilde{x}=a_{\tilde{x}}, x=a_x\}[\rho'_m],\ (a_{\tilde{x}}=ff)\rangle$$
$$\xrightarrow{*}\langle\{\tilde{x}=a_{\tilde{x}}, x=a_x\}[\rho'_m],\ (a_{\tilde{x}}=ff, a_b=ff)\rangle$$

This is by the assign and skip rules. Thus according to the block statement rule the execution of the module body is:

$$\langle Dm;\ b:=\underline{\text{false}},\ \{\tilde{x}=a_{\tilde{x}}, x=a_x\},\ (a_{\tilde{x}}=ff)\rangle$$
$$\xrightarrow{*}\langle\rho'_m;\ b:=\underline{\text{false}},\ \{\tilde{x}=a_{\tilde{x}}, x=a_x\},\ (a_{\tilde{x}}=ff)\rangle$$
$$\xrightarrow{*}\langle\{\tilde{x}=a_{\tilde{x}}, x=a_x\}[\rho'_m],\ (a_{\tilde{x}}=ff,\ a_b=ff)\rangle$$

Let $\rho_m=\{\tilde{x}=a_{\tilde{x}}, x=a_x\}[\rho'_m]$. Applying the D-module rule the elaboration of the module gives:

$$\langle\underline{\text{module}}\ M(\text{send}, \text{receive})\ BS_m\ ,\ \{\tilde{x}=a_{\tilde{x}}, x=a_x\},\ (a_{\tilde{x}}=ff)\rangle$$
$$\xrightarrow{*}\langle\{M=\rho_m\},\ \{\tilde{x}=a_{\tilde{x}}, x=a_x\},\ (a_{\tilde{x}}=ff, a_b=ff)\rangle$$

Thus the elaboration of the declaration of this program is:

$$\langle D,\emptyset,s\rangle \xrightarrow{*} \langle\{\tilde{x}=a_{\tilde{x}},x=a_x,M=\rho_m\},\emptyset,(a_{\tilde{x}}=ff,a_b=ff)\rangle$$

Let ρ denote $\{\tilde{x}=a_{\tilde{x}},x=a_x,M=\rho_m\}$. Applying the block statement rule the computation of the program becomes:

$$\langle D;(M.send(5)\,\|M.receive(loc\ x))\ ,\ \emptyset,\ (a_{\tilde{x}}=ff)\rangle$$
$$\xrightarrow{*}\langle \rho;(M.send(5)\,\|M.receive(loc\ x))\ ,\ \emptyset,\ (a_{\tilde{x}}=ff,a_b=ff)\rangle$$

Consider a computation of the concurrent statement in which the execution of the procedure call M.send(5) is followed by the execution of the procedure call M.receive(loc x). If we notice that:

$$\left|\begin{array}{l}\underline{val}\ c=5;\{\tilde{x}=a_{\tilde{x}},x=a'_x,b=a_b\};\\ \underline{when}\ \underline{not}\ b\Rightarrow(x:=c;b:=true)\ \underline{end}\end{array}\right.\ ,\rho_m,(a_{\tilde{x}}=ff,a_b=ff)\Big|$$
$$\xrightarrow{*}\left|\begin{array}{l}\{c=a_c,\tilde{x}=a_{\tilde{x}},x=a'_x,b=a_b\};\\ \underline{when}\ \underline{not}\ b\Rightarrow(x:=c;b:=true)\ \underline{end}\end{array}\right.\ ,\rho_m,\ (a_{\tilde{x}}=ff,a_b=ff,a_c=5),$$

(by the actual parameter and D-composition and block statement rules)

$$\xrightarrow{*}\left|\begin{array}{l}\{c=a_c,\tilde{x}=a_{\tilde{x}},x=a'_x,b=a_b\};\\ \underline{crit}\ (\underline{skip};b:=true)\ \underline{end}\end{array}\right.\ ,\rho_m,\ (a_{\tilde{x}}=tt,a_b=ff,a_c=5,a'_x=5)\Big|$$

(by the the when ruleand and block statement rule)

$$\xrightarrow{*}\langle\{c=a_c,\tilde{x}=a_{\tilde{x}},x=a'_x,b=a_b\};\underline{skip},\ \rho_m,\ (a_{\tilde{x}}=ff,a_b=tt,a_c=5,a'_x=5)\rangle$$

Then we see the computation of the procedure call M.send(5) is:

$$\langle M.send(5),\ \rho,\ (a_{\tilde{x}}=ff,a_b=ff)\rangle$$
$$\xrightarrow{*}\langle \underline{body}::(val\ c=5);\rho_{send};BS_s\ ,\ \rho,\ (a_{\tilde{x}}=ff,a_b=ff)\rangle$$
$$\xrightarrow{*}\langle \underline{body}::\{c=a_c,\rho_{send}\},\ \rho,\ (a_{\tilde{x}}=ff,a_b=tt,a_c=5,a'_x=5)\rangle$$

$\xrightarrow{\varepsilon}\langle\rho,(a_{\tilde{x}}=ff,a_b=tt,a_c=5,a'_x=5)\rangle$

Therefore applying the concurrent statement rule the computation of the concurrent statement becomes:

$$\langle\rho;(M.send(5)\,\|\,M.receive(loc\ x))\ ,\emptyset,(a_{\tilde{x}}=ff,a_b=ff)\rangle$$
$$\xrightarrow{\varepsilon}\langle\rho;\ M.receive(loc\ x)\ ,\emptyset,(a_{\tilde{x}}=ff,a_b=tt,a_c=5,a'_x=5)\rangle$$

Similarly, the complete computation is:

$$\langle\rho;M.receive(loc\ x)\ ,\ \emptyset,(a_{\tilde{x}}=ff,a_b=tt,a_c=5,a'_x=5)\rangle$$
$$\xrightarrow{\varepsilon}\langle\rho,\ (a_{\tilde{x}}=ff,a_b=ff,a_x=5,a'_x=5)\rangle$$

It is easy to check that if the procedure call M.receive(loc x) is executed first then its computation will be suspended when the execution reaches the corresponding when statement (the initial value of the guard b is false) until the execution of the procedure call M.send(5) is complete. □

Finally, similar to Hoare (see [Hoare 74]) we can introduce a more general form of when statement:

with t when GS end

where t is a boolean variable. The statement says to wait until no other process is executing a when statement with the same boolean t, then execute the body of the when statement. Similarly we need a new crit statement with t crit GS end to describe the execution of the when statement. The semantics of these statements are:

with-when

$$1.\ \frac{[\![t]\!]_{\rho s}=ff,\ \langle GS,\rho,s\rangle\xrightarrow{\varepsilon}\langle S,\rho',s'\rangle}{\langle \underline{with}\ t\ \underline{when}\ GS\ \underline{end},\rho,s\rangle\xrightarrow{\varepsilon}\langle \underline{with}\ t\ \underline{crit}\ S\ \underline{end},\rho',s'[tt/\rho(t)]\rangle}$$

$$2.\ \frac{[\![t]\!]_{\rho s} = ff,\ \langle GS,\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho,s'\rangle \mid \underline{abortion}}{\langle \underline{with}\ t\ \underline{when}\ GS\ \underline{end},\rho,s\rangle \xrightarrow{\varepsilon} \langle \rho,s'\rangle \mid \underline{abortion}}$$

with-crit

$$1.\ \frac{\langle S,\rho,s\rangle \xrightarrow{\varepsilon} \langle S',\rho',s'\rangle}{\langle \underline{with}\ t\ \underline{crit}\ S\ \underline{end},\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{with}\ t\ \underline{crit}\ S'\ \underline{end},\rho',s'\rangle}$$

$$2.\ \frac{\langle S,\rho,s\rangle' \xrightarrow{\varepsilon} \langle \rho',s'\rangle \mid \underline{abortion}}{\langle \underline{with}\ t\ \underline{crit}\ S\ \underline{end},\rho,s\rangle \xrightarrow{\varepsilon} \langle \underline{skip},\rho,s'[ff/\rho(t)]\rangle \mid \underline{abortion}}$$

The execution of all the new when statements with the same t take place strictly one at a time. So in this sense we can say that the statement when GS end is a "global" when statement and the statement with t when GS end is a "local" when statement.

5. An operational translation theory

Between a high level programming language and the "bare" machines on which it runs there are normally several layers represented by intermediate languages. Between each pair of consecutive levels there is a translation of high-level objects into lower-level objects. In this sense the general subject of translation is a very pervasive and important part of computer science today. The examples are too numerous to mention, but it is worth noticing that, recently, several proposals for implementing tasking in Ada use this approach. In [Luckham et al 81], the implementation of multitasking facilities in Ada is by translation into a lower level intermediate language called Adam. In another ambitious project [Bjørner and Oest 80], a semantics is given for Ada by translation into the language META+CSP. A similar approach is also taken in [Belz et al 80] where preliminary Ada has been translated into SEMANOL+Semaphores+Forking.

In theory the translation problem may be formalised in the following way: As we know, a semantics for a language L can be given by

1. a semantic domain SD(L)

2. a semantic mapping M_L from the objects L (usually programs) to SD(L)

A translation can then be viewed as a mapping $[\![\ ]\!]:L_1 \rightarrow L_2$, and its correctness can be investigated by considering the diagram

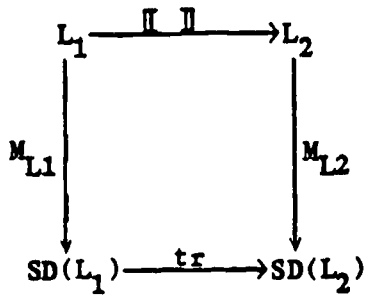


Figure 5.1

where tr is some predefined mapping between the semantic domains. Thus by the correctness of a translation we mean that the diagram commutes, i.e

$$⟦\ ⟧ \circ M_{L2} = M_{L1} \circ tr$$

Here ⟦ ⟧ is a mapping between syntactic categories and tr is a mapping between semantic domains. In our later work semantic domains will be constructed from syntactic entities, so there will be little distinction between the two and we will use the term "translation" to describe these two kinds of mapping, where the exact meaning will be understood from the context.

This kind of approach was announced first by McCarthy and Painter ([McCarthy and Painter 67]) and then Burstall and Landin ([Burstall and Landin 70]) with the goal of making compilers for high-level programming languages completely trustworthy by proving their correctness. Morris stated his belief that the compiler correctness problem is much less general and better structured than the unrestricted program correctness problem and gave the above diagram ([Morris 73]) treating ⟦ ⟧ as a compiler. Using a denotational approach he proved the correctness of a compiler for a small sequential language. Later ADJ studied this problem using an

algebraic approach [ADJ 79].

It should be mentioned that all these authors are concerned only with sequential programming languages and use either denotational semantics or algebraic semantics. A problem arises when the languages include parallelism and communication as in Ada, CCS, CSP and Edison. The reason is that no satisfactory formal semantics of such a language in the denotational or algebraic style has so far been produced, though research is progressing in this area (see [Hennessy and Plotkin 80], [Plotkin 82]).

We study this problem using the structural operational approach. Roughly speaking, the basic idea of our approach is that:

1. Any syntactic translation between languages L_1 and L_2 induces a semantic translation between the transition systems which define the operational semantics of L_1 and L_2.

2. The execution of a program can be represented by a finite transition sequence ending in a terminal configuration (successful computation), by a finite transition sequence ending in a nonterminal configuration (deadlock computation), or by an infinite transition sequence. Saying that a translation is correct amounts to saying that all these three kinds of computations for a program in the object system and in the target system correspond to each other. In particular, the possible final configurations of the translation of a program should be just the translations of the final configurations of the possible computations of the program.

3. Once we have formalised the idea of the the correctness of a translation, the next problem is to set up some sufficient conditions which guarantee the correctness and which can be used to prove a particular translation is correct. We call this the adequacy

problem. A first attempt at sufficient conditions for a correct translation may be expressed by

$$r \xrightarrow{\lambda} r' \quad \text{iff} \quad tr(r) \xrightarrow{tr(\lambda)} tr(r')$$

where $tr(\lambda)$ may be a sequence. This means that if a translation is adequate then any program and its translation should have the "same behaviour" in the respective transition systems. Intuitively, the phrase "same behaviour" includes at least that any transition of a program in the object transition system can be simulated by its translation in the target system, and any finite transition sequence from a translation of a program must be a simulation of a transition of that program. We will see later that this requirement is not enough. In fact, when a transition system describes a language with nondeterminism, parallelism and communication, the conditions which a correct translation must satisfy are very complicated. Investigating these properties is the goal of this chapter.

In [Hennessy, Li and Plotkin 81] the correctness problem in the operational approach was first studied in a concrete manner, i.e. the correctness of a translation from a simple CSP (without nested parallel structures) to CCS was studied and proved. In [Hennessy and Li 82] the adequacy problem was studied in a more general setting but the conditions found were not sufficient to prove correctness. It should also be mentioned that Jensen and Priese studied the problem of simulation between transition systems in a similar way but they only considered binary transition relations without transition labels and their conditions are too strong (see [Jensen 80] and [Priese 80]).

In section 5.1 the definition of a correct translation is given and the preservation of correctness under composition is proved. In

section 5.2 we deal with congruence relations on transition systems. In section 5.3 the adequacy problem is studied and a set of sufficient conditions for the correctness is given. Some examples are studied and help us to show why these conditions were chosen.

5.1 Translation and its correctness

To formalise and prove the correctness of a translation we need the following definitions:

Definition 5.1

Given a transition system $\Pi=\langle \Gamma, T, \Lambda, \rightarrow \rangle$, and recollecting that D is the set of deadlock configurations of Π (see section 1.2), we define:

$$\mathbb{R}(r)=\{r' \mid r \xrightarrow{*} r',\ r' \in T\}$$

$$\mathbb{K}(r)=\{r' \mid r \xrightarrow{*} r',\ r' \in D\}$$

Informally, $\mathbb{R}(r)$ is the set of reachable terminal configurations from r, and $\mathbb{K}(r)$ is the set of reachable deadlock configurations from r.

Definition 5.2 Semi-D

Given a transition system $\Pi=\langle \Gamma, T, \Lambda, \rightarrow \rangle$, the set Semi-D is the least set such that:

1. $D \subseteq$ Semi-D

2. For any $r \in \Gamma$ if $\Delta(r) \subseteq$ Semi-D then $r \in$ Semi-D.

A configuration r is semi-deadlocked if $r \in$ Semi-D.

This definition means that if $r \in$ Semi-D then all complete computations from r are finite with final configurations in D; that

is, r must eventually deadlock. A translation between two transition systems is defined as follows:

Definition 5.3 Translation

Given two labelled transition systems $\mathbb{T}_1=\langle \Gamma_1, T_1, \Lambda_1, \xrightarrow[1]{} \rangle$ and $\mathbb{T}_2=\langle \Gamma_2, T_2, \Lambda_2, \xrightarrow[2]{} \rangle$, a translation tr from $\mathbb{T}_1$ into $\mathbb{T}_2$ is any pair of mappings (we use the same name for both):

$$tr : \Gamma_1 \to \Gamma_2 \quad \text{and} \quad tr : \Lambda_1 \to \Lambda_2$$

where the second is an injective mapping. The transition system $\mathbb{T}_1$ is called the <u>object system</u> and $\mathbb{T}_2$ is called the <u>target system</u>. A configuration $x \in \Gamma_2$ is called a <u>translated configuration</u> if for some $r \in \Gamma_1$ $tr(r)=x$; a transition label $\rho \in \Lambda_2$ is called a <u>translated label</u> if for some $\lambda \in \Lambda_1$ $tr(\lambda)=\rho$. □

Having introduced the necessary notation, we can now consider the correctness problem of a translation. Since any complete computation from a configuration r is either finite or infinite and when the computation is finite then the final configuration can only be a terminal or deadlock configuration, there is a natural definition:

Definition 5.4 Correct translation

A translation tr as given above is <u>correct</u> iff for any $r \in \Gamma_1$

A1. $tr(\mathbb{R}(r))=\mathbb{R}(tr(r))$

A2. $tr(\mathbb{K}(r)) \subseteq \text{Semi-}D_2$ and $\mathbb{K}(tr(\mathbb{K}(r)))=\mathbb{K}(tr(r))$

A3. $r\uparrow$ iff $tr(r)\uparrow$ □

Condition A1 means that terminal configurations reachable from the translation of a configuration are just the translations of terminal configurations reachable from the original configuration.

The first formula of condition A2 means that the translations of deadlocked configurations must eventually deadlock. If we omit the leftmost $\mathbb{K}$, then the second formula of condition A2 becomes

$$tr(\mathbb{K}(r))=\mathbb{K}(tr(r))$$

and this means that deadlocked configurations reachable from the translation of a configuration are just the translations of deadlocked configurations reachable from the original configuration. However since $tr(\mathbb{K}(r))$ is in Semi-D2 and is Semi-deadlocked, the $\mathbb{K}$ is added to the left part of the formula. Condition A3 says that the infinite computations in $\mathbb{T}_1$ correspond with those in $\mathbb{T}_2$. The following theorem partly expresses this idea:

Definition 5.5 Input-output function
Given a transition system $\mathbb{T}=\langle \Gamma, T, \Lambda, \rightarrow \rangle$, the input-output function $[\![\ ]\!]: \Gamma \rightarrow T \cup \{\delta, \perp\}$ is defined by:

$$[\![r]\!]=\mathbb{R}(r) \cup \{\delta \mid \exists r' \in D,\ r \xrightarrow{*} r'\} \cup \{\perp \mid r\uparrow\}$$

where $r \in \Gamma$, and δ and $\perp$ are distinguished symbols to denote deadlock and divergence respectively.

In particular, if $\Gamma=(Syn \times States) \cup States$ where Syn denotes the set of syntactic entities of a language and T=States, then the above input-output function can be written as

$$[\![\ ]\!]: Syn \times States \rightarrow States \cup \{\delta, \perp\}$$

and $$[\![\Omega]\!]_s=\{s' \mid \langle \Omega, s\rangle \xrightarrow{*} s'\}$$
$$\cup \{\delta \mid \exists \langle \Omega', s'\rangle \in D,\ \langle \Omega, s\rangle \xrightarrow{*} \langle \Omega', s'\rangle\}$$
$$\cup \{\perp \mid \langle \Omega, s\rangle \uparrow\}$$

and this is the usual form of the input-output function. The following general result holds for any correct translation:

Theorem 5.1

If the translation $tr:\mathbb{T}_1 \rightarrow \mathbb{T}_2$ is correct then putting $tr(\dot{\perp})=\dot{\perp}$ and $tr(\delta)=\delta$, for any $r \in \Gamma_1$

$$tr(\llbracket r \rrbracket) = \llbracket tr(r) \rrbracket .$$

Proof. Since tr is correct by condition A1 we have $tr(\mathbb{R}(r))=\mathbb{R}(tr(r))$. We only need to prove:

1. $\delta \varepsilon \llbracket r \rrbracket$ iff $\delta \varepsilon \llbracket tr(r) \rrbracket$.
2. $\dot{\perp} \varepsilon \llbracket r \rrbracket$ iff $\dot{\perp} \varepsilon \llbracket tr(r) \rrbracket$.

For the first clause, if $\delta \varepsilon \llbracket r \rrbracket$ then by definition 5.5 $\mathbb{K}(r) \neq \emptyset$. By condition A2, $tr(\mathbb{K}(r)) \subseteq Semi\text{-}D_2$ so $\mathbb{K}(tr(r))=\mathbb{K}(tr(\mathbb{K}(r))) \neq \emptyset$. That is $\delta \varepsilon \llbracket tr(r) \rrbracket$. Conversely, if $\delta \varepsilon \llbracket tr(r) \rrbracket$ then $\mathbb{K}(tr(r)) \neq \emptyset$ so by A2 we have $\mathbb{K}(r) \neq \emptyset$, and this means $\delta \varepsilon \llbracket r \rrbracket$.

The second clause is immediate from A3. □

We can see that the converse of theorem 5.1 is not true, i.e. if theorem 5.1 holds then the translation need not be correct. This is because theorem 5.1 does not distinguish the deadlock computations but definition 5.4 does. Consider the example below:

Example 5.1

Consider the two transition systems associated with the graphs:

(a) (b)

$\mathbb{T}_1=\langle \Gamma_1, T_1, \Lambda_1, \xrightarrow[1]{} \rangle$ where $\Gamma_1=\{0,1,e\}$, $T_1=\{1\}$, $\Lambda_1=\{\tau\}$ and the transition relation $\xrightarrow[1]{}$ is given as graph (a).

$\mathbb{T}_2=\langle \Gamma_2, T_2, \Lambda_2, \xrightarrow{}_2 \rangle$ where $\Gamma_2=\{0,1,e,e'\}$, $T_2=\{1\}$, $\Lambda_2=\{\tau\}$ and the transition relation is shown as graph (b).

Let the translation $tr:\mathbb{T}_1 \to \mathbb{T}_2$ be the identity mapping. It is not difficult to see that:

$$tr([\![r]\!]) = [\![tr(r)]\!] = \{1, \delta\}$$

but $tr(\mathbb{K}(0))=\{e\} \neq \{e,e'\}=\mathbb{K}(tr(0))$

□

In fact we can take theorem 5.1 as a definition of the correctness of a translation. But in this thesis we prefer definition 5.4, because it is stronger, and sometimes distinguishing the different deadlock computations may be important to find run-time errors in practice.

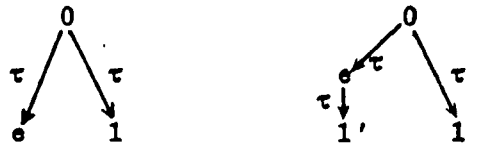
One might ask why we do not use the set $tr(\mathbb{K}(r))$ to replace $\mathbb{K}(tr(\mathbb{K}(r)))$ in condition A2. The reason is that in most cases when we translate a high-level system into a low-level system, the equation

$$tr(\mathbb{K}(r))=\mathbb{K}(tr(r))$$

is not true. Let us examine the following example:

Example 5.2

Consider the transition systems associated with the graphs:



(a) (b)

Figure 5.2

$\mathbb{T}_1=\langle\Gamma_1,T_1,\Lambda_1,\xrightarrow[1]{}\rangle$, where $\Gamma_1=\{0,e,1\}$, $T_1=\{1\}$, $\Lambda_1=\{\tau\}$, and the transition relation $\xrightarrow[1]{}$ is shown as graph (a).

Let $\Gamma_2=\{o,1,e,1'\}$, $T_2=\{1\}$, $\Lambda_2=\{\tau,\tau'\}$ and $\xrightarrow[2]{}$ is defined by graph (b).

Let the translation $tr:\mathbb{T}_1\rightarrow\mathbb{T}_2$ be the identity mapping. It is easy to see that:

$$\mathbb{K}(tr(0))=\{1'\}\neq\{e\}=tr(\mathbb{K}(0))$$

but $\mathbb{K}(tr(0))=\{1'\}=\mathbb{K}(tr(\mathbb{K}(0)))$

Clearly however the translation is correct in an intuitive sense since e is a deadlocked configuration in $\mathbb{T}_1$ and the complete computation from tr(e) in $\mathbb{T}_2$ terminates with a deadlock configuration. □

As we claimed at the beginning of this chapter, between a high level programming language and a "bare" machine there are often several translations between consecutive layers. It is therefore natural to ask whether the composition of two consecutive translations is correct if both translation are correct. The answer is in the affirmative.

Definition 5.6

Given transition systems $\mathbb{T}_1$, $\mathbb{T}_2$, $\mathbb{T}_3$ and translations

$$tr1:\mathbb{T}_1\rightarrow\mathbb{T}_2 \quad \text{and} \quad tr2:\mathbb{T}_2\rightarrow\mathbb{T}_3$$

The composite translation $tr2\circ tr1:\mathbb{T}_1\rightarrow\mathbb{T}_3$ is defined by:

$$tr2\circ tr1(r)=tr2(tr1(r)) \quad \text{for} \quad r\in\Gamma_1$$

$$tr2\circ tr1(\lambda)=tr2(tr1(\lambda)) \quad \text{for} \quad \lambda\in\Lambda_1$$

Lemma 5.1

If $tr:\mathbb{T}_1 \rightarrow \mathbb{T}_2$ is correct then $tr(\text{Semi-}D_1) \subseteq \text{Semi-}D_2$.

Proof. Let $r \in \text{Semi-}D_1$. We prove that $tr(r) \in \text{Semi-}D_2$.

Firstly, there is no infinite computation from $tr(r)$; otherwise by A3 there is an infinite computation from r and this contradicts the assumption $r \in \text{Semi-}D_1$.

Secondly, $\mathbb{R}(tr(r)) \neq \emptyset$ otherwise by A1 we have

$$tr(\mathbb{R}(r)) = \mathbb{R}(tr(r)) \neq \emptyset \quad \text{so} \quad \mathbb{R}(r) \neq \emptyset$$

and this also contradicts the assumption $r \in \text{Semi-}D_1$. □

For the composition of translations we have the following result:

Theorem 5.2

Let tr1, tr2 and tr2∘tr1 be given as above. If tr1 and tr2 are correct then so is tr2∘tr1.

Proof We prove that tr2∘tr1 satisfies A1 to A3.

For A1, take $r \in \Gamma_1$, and calculate:

$$\begin{aligned}
\mathbb{R}(tr2 \circ tr1(r)) &= \mathbb{R}(tr2(tr1(r))) \\
&= tr2(\mathbb{R}(tr1(r))) && \text{(tr2 is correct)} \\
&= tr2(tr1(\mathbb{R}(r))) && \text{(tr1 is correct)} \\
&= tr2 \circ tr1(\mathbb{R}(r))
\end{aligned}$$

Thus A1 is proved.

For A2, we first calculate that for any $r \in \Gamma_1$,

$$tr2(tr1(\mathbb{K}(r))) \subseteq tr2(\text{Semi-}D_2) \subseteq \text{Semi-}D_3 \quad \text{by lemma 5.1}$$

and then that

$$\begin{aligned}\mathbb{K}(tr2\circ tr1(r)) &= \mathbb{K}(tr2(tr1(r))) \\ &= \mathbb{K}(tr2(\mathbb{K}(tr1(r)))) \quad (tr2 \text{ is correct}) \\ &= \mathbb{K}(tr2(\mathbb{K}(tr1(\mathbb{K}(r))))) \quad (tr1 \text{ is correct}) \\ &= \mathbb{K}(tr2(tr1(\mathbb{K}(r))))\end{aligned}$$

So $tr2\circ tr1$ satisfies A2.

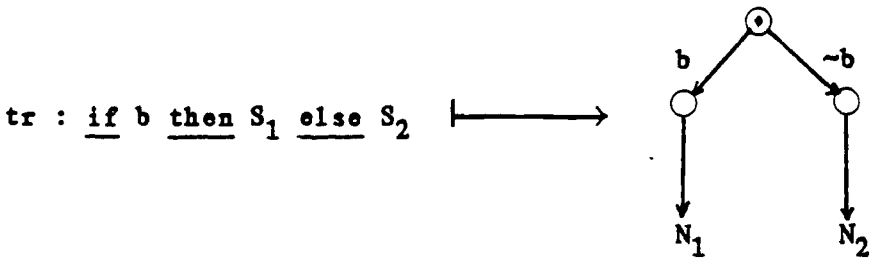
For A3, we just note that $tr2\circ tr1(r)\uparrow$ iff $tr1(r)\uparrow$ iff $r\uparrow$. Thus the theorem is proved. □

5.2 Congruence relations on labelled transition systems

When we study the correctness problem we need to understand the notion of a congruence relation on labelled transition systems. The reason is that usually the target language is more primitive than the object language; therefore one transition step of the object system will be modelled by a number of steps of the target system. To organise this simulation properly some "housekeeping" must be done and this inevitably introduces some "debris" into the target system. The following example will clarify this issue.

Example 5.3 Translating into nets

Consider a translation of a simple programming language into predicated nets. For example the statement *if* b *then* S_1 *else* S_2 would be mapped into the net



where N_i represents the net which results from the translation of S_i

i=1,2. In the case when the predicate b is true, we would have the transition

$$\underline{\text{if}}\ \text{true}\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2 \longmapsto S_1$$

However in the target system, we would have

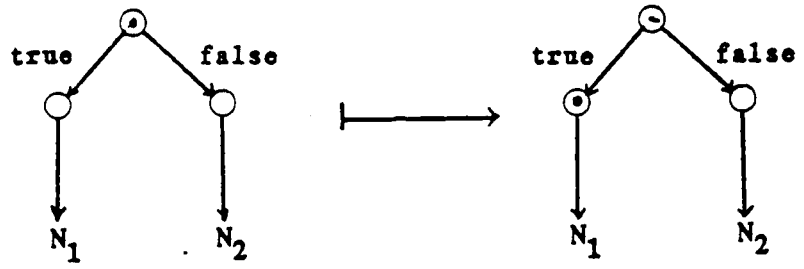


Figure 5.3

and the right hand side is different from $tr(S_1)=N_1$. However net theorists would agree that the nets

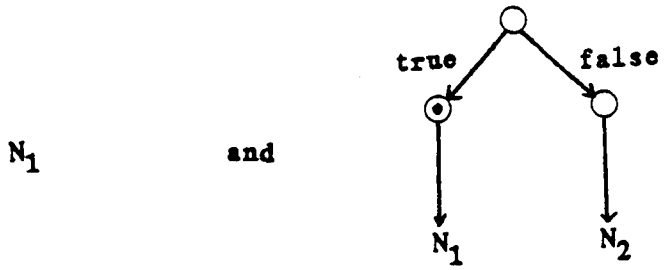


Figure 5.4

are equivalent and therefore the diagram is "almost" satisfied. The degree to which the qualifier "almost" is used depends on which nets we agree to say are equivalent. For this reason we introduce the following concept.

Definition 5.7 Congruence relations on labelled transition systems
Let $\mathbb{T}=\langle \Gamma, T, \Lambda, \rightarrow \rangle$ be a transition system. A relation $\sim \subseteq \Gamma \times \Gamma$ is called a congruence relation on $\mathbb{T}$ iff $\sim$ is a congruence relation

and whenever $p \sim q$, $\lambda \in \Lambda$ and $p' \in \Gamma$

a. if $p \xrightarrow{\lambda} p'$ then for some $q' \in \Gamma$ $q \xrightarrow{\lambda} q'$ and $p' \sim q'$.

b. if $p \in T$ then $q \in T$. □

From the definition we can see that the degree to which the qualifier "almost" is used depends on the congruence relation $\sim$, i.e. which configurations we agree to say are equivalent. For a congruence relation on labelled transition system, we have the following result.

Theorem 5.3

A relation $\sim \subseteq \Gamma \times \Gamma$ is a congruence relation on T iff $\sim$ is a congruence relation and whenever $p \sim q$, $w \in \Lambda^*$ and $p' \in \Gamma$

a. if $p \xrightarrow{w} p'$ then for some $q' \in \Gamma$ $q \xrightarrow{w} q'$ and $p' \sim q'$.

b. if $p \in T$ then $q \in T$

Proof. We prove (a) only. The "if" part is trivial. We prove the "only if" part by induction on length(w).

If length(w)=0, then the result is obvious. If length(w)=1 then the result follows by definition 5.7

Suppose the case length(w)=k is proved. Consider the case length(w)=k+1. Then $p \xrightarrow{hd(w)} p'' \xrightarrow{tl(w)} p'$, thus for some q" we have $q \xrightarrow{hd(w)} q''$ and $p'' \sim q''$. Since length(tl(w))=k, by the induction hypothesis we have that there must exist $q' \in \Gamma$ such that $q'' \xrightarrow{tl(w)} q'$ and $p' \sim q'$. So $q \xrightarrow{w} q'$ and $p' \sim q'$. □

Example 5.4 A congruence relation $\sim$ on CCS

Consider CCS terms $\alpha x.Nil \mid \alpha(5,Nil)$ and $\gamma(5,\alpha x.Nil)[\beta/\gamma]$. According to the transition rules in section 1.3.3 we have

$$\alpha x.Nil \mid \alpha(5,Nil) \xrightarrow{\tau} Nil \mid Nil$$

$$\gamma(5,\alpha x.Nil)[\beta/\gamma] \xrightarrow{\beta 15} \alpha x.Nil[\beta/\gamma]$$

In general, an execution of a CCS term produces a lot of redundant Nils and renamings, for example, Nil in the first transition and $[\beta/\gamma]$ in the second; they are 'debris". The "housekeeping" which we should do is to let $Nil \mid Nil \sim Nil$ and $\alpha x.Nil[\beta/\gamma] \sim \alpha x.Nil$. To formalise this, we introduce the following relation ~:

Given a renaming ϕ let $CD(\phi)$ denote the set of line names which are really changed or restricted by ϕ, that is

$$CD(\phi)=\{\alpha \mid \phi(\lambda) \text{ is undefined or } \phi(\alpha)\neq\alpha\}$$

Then ~ is defined by:

1. $t \sim t \mid Nil$
2. $t \sim t[\phi]$ where $FL(t) \cap CD(\phi)=\emptyset$
3. $t \sim t$
4. If $t \sim u$ then $u \sim t$
5. If $t \sim u$ and $u \sim v$ the $t \sim v$
6. If $t \sim u$ then $C[t] \sim C[u]$ for any context $C[\ ]$ of CCS

where a context is merely a term with a "hole" in it in which any other term may be placed.

The first two clauses do the housekeeping: the first means that we can always omit the term Nil and the second means that we can always remove the renaming ϕ from $t[\phi]$ if all lines contained in t cannot be changed or restricted by ϕ. Clause 3 to 5 say that ~ is an equivalence relation. Together with clause 6 they say that ~ is a congruence relation on Π. This can be proved by structural induction on the syntax of the terms. Thus we have $Nil \mid Nil \sim Nil$ and

$\alpha x.Nil[\beta/\gamma] \sim \alpha x.Nil$. □

Remarks If ~ is a congruence relation on the transition system $\mathbb{T}$, we will simply use the transition $r \xrightarrow{\lambda} r'$ to indicate that there is an r'' in Γ such that $r \xrightarrow{\lambda} r''$ and $r' \sim r''$. This means that r can evolve to r' via the transition action λ to within the accuracy of the congruence relation ~. Finally, in the remainder of this chapter we sometimes study transition systems without mentioning any congruence relation. In this case we mean that the congruence relation is the identity relation.

5.3 Adequate translation

In this section we invent some sufficient conditions for proving that a translation is correct. The difficulties involved in languages with parallel and nondeterministic constructs make the right conditions hard to find. For example, the following properties may at the first seem to be sufficient:

X1. $r \xrightarrow[1]{*} r'$ implies $tr(r) \xrightarrow[2]{*} tr(r')$.

X2. $tr(r) \xrightarrow[2]{*} tr(r')$ implies $r \xrightarrow[1]{*} r'$.

X3. $r \varepsilon T_1$ implies $tr(r) \varepsilon T_2$.

Unfortunately, there are many instances of translations which satisfy these properties and which are not correct. Let us consider a simple example:

Example 5.5 A bad translation

Consider the following translation:

Figure 5.5

Let $T_1:\langle \Gamma_1, T_1, \Lambda_1, \xrightarrow[1]{} \rangle$ where $\Gamma_1=\{1,2\}$, $T_1=\{2\}$, $\Lambda_1=\{\tau\}$ and $\xrightarrow[1]{}$ is defined by

$$1 \xrightarrow{\tau} 2$$

Let $T_2:\langle \Gamma_2, T_2, \Lambda_2, \xrightarrow[2]{} \rangle$ where $\Gamma_2=\{1,2,2'\}$, $T_2=T_1$, $\xrightarrow[2]{}$ is defined by

1. $1 \xrightarrow{\tau} 2$
2. $1 \xrightarrow{\tau} 2'$
3. $2' \xrightarrow{\tau} 2'$

Let the translation tr be defined by

$$tr(n)=n \quad \text{and} \quad tr(\tau)=\tau \qquad n=1,2$$

This translation satisfies X1, X2 and X3, but is clearly not a correct translation. □

To eliminate such phenomena we could introduce a condition to the effect that every infinite transition sequence starting with tr(r) must contain a configuration other than the initial one, of the form tr(r'). But it is easy to find an example to show that this requirement is too strong. In fact, in certain instances we need to allow infinite sequences in the target system which lie outside the range of tr. See the following example:

Example 5.6

Let $T_1=\langle \Gamma_1, T_1, \Lambda_1, \xrightarrow[1]{} \rangle$ be defined by

1. $\Gamma_1=N\times N$ (where N denotes the natural numbers)

2. $T_1=\emptyset$ and $\Lambda_1=\{\tau\}$

3. $\xrightarrow[1]{}$ is defined by

$$\langle n,m\rangle \xrightarrow[1]{\tau} \langle n+1,m\rangle \qquad \text{for all } n,m\in N$$
$$\langle n,m\rangle \xrightarrow[1]{\tau} \langle n,m+1\rangle \qquad " \quad "$$

The intuition is the interleaving of two parallel processes and we could draw a net:

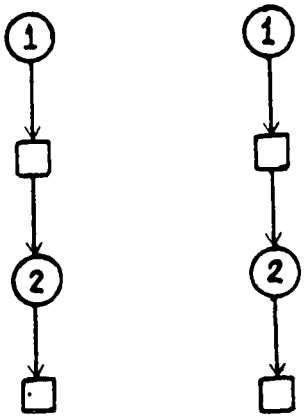


Figure 5.6

Let $\Pi_2:\langle \Gamma_2, T_2, \Lambda_2, \xrightarrow[2]{}\rangle$ be defined by

1. $\Gamma_2=N_1 \times N_1$ (where $N_1=N\cup N'=N\cup\{n' \mid n\in N\}$)

2. $T_2=\emptyset$, $\Lambda_2=\{\tau\}$

3. let $n,m\in N$, $n',m'\in N'$ and $z\in N_1$. $\xrightarrow[2]{}$ is defined by

$$\langle n,z\rangle \xrightarrow[2]{\tau} \langle n',z\rangle$$
$$\langle z,m\rangle \xrightarrow[2]{\tau} \langle z,m'\rangle$$
$$\langle n',z\rangle \xrightarrow[2]{\tau} \langle n+1,z\rangle$$
$$\langle z,m'\rangle \xrightarrow[2]{\tau} \langle z,m+1\rangle$$

The intuition is as before but we include some intermediate

configurations such as $\langle n',m\rangle$ and $\langle n,m'\rangle$ in T_2. Thus each transition in T_1 is split into two transitions in T_2, i.e.
$\langle n,m\rangle \xrightarrow[1]{\tau} \langle n+1,m\rangle$ in T_1 becomes $\langle n,m\rangle \xrightarrow[2]{\tau} \langle n',m\rangle \xrightarrow[2]{\tau} \langle n+1,m\rangle$ in T_2.

We could draw a net:

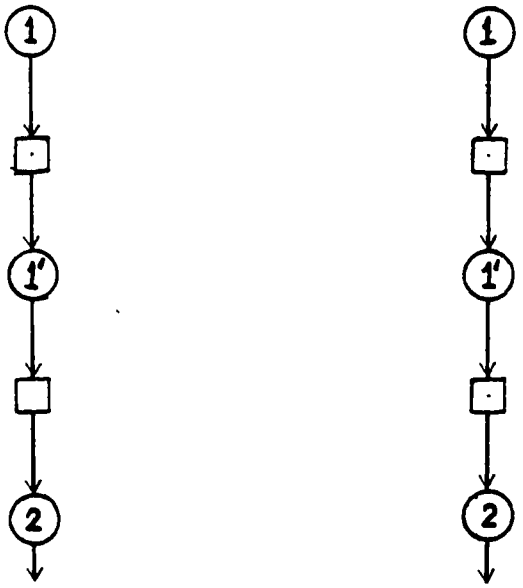


Figure 5.7

Let a translation tr be defined by

$$tr(\langle n,m\rangle)=\langle n,m\rangle \quad \text{and} \quad tr(\tau)=\tau$$

Then tr is correct and it of course satisfies X1, X2 and X3, but it does not satisfy the suggested condition. A typical infinite sequence lying outside the range of tr is:

$$\langle 1,1\rangle \xrightarrow[2]{\tau} \langle 1',1\rangle \xrightarrow[2]{\tau} \langle 1',1'\rangle \xrightarrow[2]{\tau} \langle 2,1'\rangle \xrightarrow[2]{\tau} \langle 2',1'\rangle \xrightarrow[2]{\tau} \ldots$$

Such a phenomenon is called an "accident of interleaving" and occurs naturally if we wish to translate a parallel language with assignment statements into an assembly language in which transfer of data can only be made between the memory and registers and not from memory to memory. In the next chapter when we translate CSP into CCS

this phenomenon will occur. □

The above two examples demonstrate that for a language involving parallelism and nondeterminism and even for a very simple case the execution of a program is rather complicated and finding sufficient conditions for the correctness of a translation is quite difficult. In the rest of this section we focus our attention on discovering appropriate conditions. As a first step we introduce the following useful notations and concepts.

For sequences of transition labels we need a partial function rt in the direction opposite to the translation tr.

Definition 5.8 function rt
The function $rt:\Lambda_2^* \rightarrow \Lambda_1^*$ is defined recursively by

$$rt(u)=\begin{cases}\emptyset & \text{if } u=\emptyset \\ \lambda.rt(tl(u)) & \text{if } \lambda\in\Lambda_1 \text{ and } tr(\lambda)=hd(u) \\ rt(tl(u)) & \text{if } hd(u)\notin tr(\Lambda_1)\end{cases}$$

Note that since $tr:\Lambda_1 \rightarrow \Lambda_2$ is an injection the function rt is well-defined. For the configurations in the target system reachable from a translated configuration we introduce the following notation:

Definition 5.9 Ex-translated configurations
The set of <u>Ex-translated</u> configurations $Ex\text{-}tr(\Gamma_1)$ is the least set such that

if $x\in\Gamma_2$ and for some $r\in\Gamma_1$ $u\in\Lambda_2^*$, $tr(r)\xrightarrow{u}_2 x$ then $x\in Ex\text{-}tr(\Gamma_1)$. □

We are now ready to characterise adequate translations.

Definition 5.10 Adequate translation

Let $\sim_1$, $\sim_2$ be the two congruence relations on the transition systems $\mathbb{T}_1$ and $\mathbb{T}_2$ respectively. A translation $tr:\mathbb{T}_1 \to \mathbb{T}_2$ is said to be <u>M-adequate</u> if it satisfies the following properties where $M \subseteq \Lambda_2 \backslash tr(\Lambda_1)$.

P0. If $r \in T_1$ then $tr(r) \in T_2$, and for $x \in T_2$ if $tr(r) \xrightarrow[2]{u} x$ and $rt(u)=\emptyset$ then $r \in T_1$.

P1. Whenever $r, r' \in \Gamma_1$, $\lambda \in \Lambda_1$, if $r \xrightarrow[1]{\lambda} r'$ then for some $u \in \Lambda_2^*$ $tr(r) \xrightarrow[2]{u} tr(r')$ and $rt(u)=\lambda$.

P2. If $tr(r) \xrightarrow[2]{u} x$, $rt(u)=\lambda$ then there exist $r' \in \Gamma_1$, $x' \in \Gamma_2$ and u_1 with $rt(u_1)=\emptyset$ and $u_2 \in M^*$ such that $r \xrightarrow[1]{\lambda} r'$, $tr(r') \xrightarrow[2]{u_1} x'$ and $x \xrightarrow[2]{u_2} x'$ (see Figure 5.8).

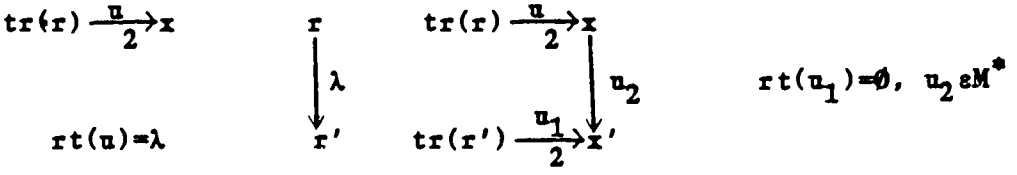
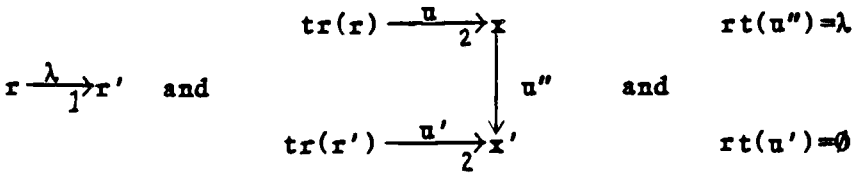


Figure 5.8

P3. If $tr(r)\uparrow$ then there exist $r' \in \Gamma_1$ and $\lambda \in \Lambda_1$ such that $r \xrightarrow[1]{\lambda} r'$ and $tr(r')\uparrow$.

P4. $\xrightarrow[2]{}$ in $Ex\text{-}tr(\Gamma_1)$ is M-commutative.

P5. If r is active ($r \notin D_1 \cup T_1$), $tr(r) \xrightarrow{u} x$ and $rt(u)=\emptyset$ then there exist $r' \in \Gamma_1$, $x' \in \Gamma_2$, $\lambda \in \Lambda_1$, and $u', u'' \in \Lambda_2^*$ such that



We now have two concepts to evaluate the quality of a translation: correctness and adequacy. The concept of correctness focuses on the result (especially the input-output relation) of a program and its translation. The concept of adequacy pays more attention to the detailed simulation of the behaviour of a program by its translation. Loosely, clause P0 says that a terminal configuration of $\mathbb{T}_1$ must be translated into a terminal configuration of $\mathbb{T}_2$ and conversely. Clause P1 ensures that any transition of the object system can be simulated by the target system; clauses P2 and P4 imply a weak version of the converse. Clauses P1 and P3 cover the simulation of infinite transition sequences. Finally, clause P5 implies that deadlocked configurations in $\mathbb{T}_1$ and $\mathbb{T}_2$ correspond with each other.

It should be pointed out that the set M mentioned in the definition may be a proper subset of $\Lambda_2 \backslash tr(\Lambda_1)$. For some reasonable translations, the transition relations in $Ex\text{-}tr(\Gamma_1)$ are only commutative in proper subsets of $\Lambda_2 \backslash tr(\Lambda_1)$.

The condition P3, i.e. "if $tr(r)\uparrow$ then there exist $r' \in \Gamma_1$ and $\lambda \in \Lambda_1$ such that $r \xrightarrow[1]{\lambda} r'$ and $tr(r')\uparrow$" is necessary for a translation to be correct. See the following example:

Example 5.7

Let $\mathbb{T}_1 : \langle \Gamma_1, T_1, \Lambda_1, \xrightarrow[1]{} \rangle$ be defined by $\Gamma_1 = N \times N$, $T_1 = \emptyset$, $\Lambda_1 = \{\tau\}$, where $\xrightarrow[1]{}$ is defined by

$$\langle n,m \rangle \xrightarrow[1]{\tau} \langle n+1,m \rangle \qquad \text{if } n \neq m$$
$$\langle n,m \rangle \xrightarrow[1]{\tau} \langle n,m+1 \rangle \qquad \text{if } n \neq m$$

Let $\mathbb{T}_2 : \langle \Gamma_2, T_2, \Lambda_2, \xrightarrow[2]{} \rangle$ be defined by

$\Gamma_2 = \Gamma_1$, $T_2 = T_1$, $\Lambda_2 = \{\tau, \tau'\}$, and $\xrightarrow[2]{}$ is defined by adding to

$\xrightarrow[1]{}$ the pair

$$\langle n,n\rangle \xrightarrow[2]{\tau'} \langle n+1,n+1\rangle$$

Finally, let tr be defined by

$$tr(\langle n,m\rangle)=\langle n,m\rangle$$

$$tr(\tau)=\tau$$

The translation tr satisfies all conditions of definition 5.10 except P3. However it is easy to see that $tr(\langle n,n\rangle)$ has an infinite computation

$$\langle n,n\rangle \xrightarrow[2]{\tau'} \langle n+1,n+1\rangle \xrightarrow[2]{\tau'} \ldots$$

while in the object transition system $\langle n,n\rangle$ does not. The translation tr is not correct because it introduces the ability to diverge. □

It also should be mentioned that property P5 is also necessary for a translation to be correct; otherwise the stuck computations may not be introduced, as in the following example:

Example 5.8

Let $T_1:\langle \Gamma_1, T_1, \Lambda_1, \xrightarrow[1]{}\rangle$ be defined by $\Gamma_1=N\times N$, $T_1=\emptyset$, $\Lambda_1=\{\tau\}$, and $\xrightarrow[1]{}$ is defined by

$$\langle n,m\rangle \xrightarrow[1]{\tau} \langle n+1,m\rangle$$

$$\langle n,m\rangle \xrightarrow[1]{\tau} \langle n,m+1\rangle$$

Let $T_2:\langle \Gamma_2, T_2, \Lambda_2, \xrightarrow[2]{}\rangle$ be defined by

$\Gamma_2=\Gamma_1 \cup \{\langle 1',1'\rangle\}$, $\Lambda_2=\{\tau,\tau'\}$, and $\xrightarrow[2]{}$ is defined by adding to $\xrightarrow[1]{}$ the following relation

$\langle n,m\rangle \xrightarrow[2]{\tau'} \langle 1',1'\rangle$

Let the translation tr be defined by

$tr(r)=r$

$tr(\tau)=\tau$

It is easy to see that the translation tr satisfies P0 to P4 but not P5 since $\langle 1',1'\rangle$ is a deadlock configuration. It is of course not adequate since $\langle n,m\rangle$ can deadlock in $\mathbb{T}_2$ but can never deadlock in $\mathbb{T}_1$! □

We now prove that if a translation is adequate then it must be correct. In the following lemmas, theorems and their proofs we consider a fixed adequate translation $tr:\mathbb{T}_1 \rightarrow \mathbb{T}_2$, we use r, x (maybe with superscripts or subscripts) to range over elements in Γ_1 and Γ_2 respectively, and λ and ρ (maybe with superscripts and subscripts) to range over labels in Λ_1 and Λ_2 respectively, and use w and u to range over sequences in Λ_1^* and Λ_2^* respectively.

The first lemma is a "many-step" generalisation of P1.

Lemma 5.2

$r \xrightarrow[1]{w} r'$ implies that there exists u such that $tr(r) \xrightarrow[2]{u} tr(r')$ and $rt(u)=w$.

Proof. By induction on the length of w.

If length(w)=$\emptyset$, then choose u=$\emptyset$ and the result is trivial.

Suppose the case of length(w)=k is proved, consider the case length(w)=k+1. Since $r \xrightarrow[1]{hd(w)} r'' \xrightarrow[1]{tl(w)} r'$, according to P1 there exists $u_1 \in \Lambda_2^*$ such that $rt(u_1)=hd(w)$ and $tr(r) \xrightarrow[2]{u_1} tr(r'')$. Since

length(tl(w))=k we have by the induction hypothesis that there exists $u_2 \varepsilon \Lambda_2^*$ such that $rt(u_2)=tl(w)$ and

$$tr(r'') \xrightarrow[2]{u_2} tr(r')$$

Thus choose $u=u_1.u_2$ and the result has been proved. □

Corollary 5.1

$$tr(\mathbb{R}(r)) \subseteq \mathbb{R}(tr(r))$$

Proof. We need to prove that if $r' \varepsilon \mathbb{R}(r)$ then $tr(r') \varepsilon \mathbb{R}(tr(r))$. Since $r \xrightarrow{w} r'$ by lemma 5.2 we have $tr(r) \xrightarrow{u} tr(r')$ for some u and by P0 we have $tr(r') \varepsilon T_2$, showing $tr(r') \varepsilon \mathbb{R}(tr(r))$. □

The next lemma is a "many-step" generalisation of P2.

Lemma 5.3

$tr(r) \xrightarrow[2]{u} x$ implies that there exist $r' \varepsilon \Gamma_1$, $x' \varepsilon \Gamma_2$, $rt(u_1)=\emptyset$ and $u_2 \varepsilon M^*$ such that

$$r \xrightarrow[1]{rt(u)} r' \qquad \text{and} \qquad \begin{array}{ccc} tr(r') & & x \\ & {\scriptstyle u_1}\searrow \quad \swarrow{\scriptstyle u_2} & \\ & x' & \end{array}$$

Figure 5.9

Proof. By induction on length(rt(u)).

If length(rt(u))=∅, then there is nothing to prove as we can take $r'=r$, $x'=x$, $u_1=u$ and $u_2=\emptyset$. If length(rt(u))=1 then the result is just P2.

Now suppose the case length(rt(u))=n is proved and consider the case length(rt(u))=n+1.

Let u=t.v with length(rt(t))=1 and length(rt(v))=n. Now we have, for some y, $tr(r) \xrightarrow{t} y \xrightarrow{v} x$. By P2 we have that there exist r'', y'

t1 with $rt(t1)=\emptyset$ and $t2 \in M^*$ such that

$$r \xrightarrow{rt(t)} r'' \quad \text{and} \quad \begin{array}{ccc} & & y \\ & & \downarrow t2 \\ tr(r'') & \xrightarrow{t1} & y' \end{array}$$

Now consider the diagram:

$$\begin{array}{ccc} y & \xrightarrow{v} & x \\ {\scriptstyle t2}\downarrow & & \downarrow{\scriptstyle t3} \\ y' & \xrightarrow{v'} & x'' \end{array} \qquad \text{with } t2 \in M^*$$

By theorem 1.1 this fills in as indicated with $t3 \in M^*$ and $rt(v')=rt(v)$ since $fil_M(v')=fil_M(v)$ and $M \subseteq \Lambda_2 \backslash tr(\Lambda_2)$. Then we have

$tr(r'') \xrightarrow{t1.v'} x''$ with $rt(t1.v')=rt(t1).rt(v')=rt(v')=rt(v)$,

and so the induction hypothesis applies and we get $r' \in \Gamma_1$, such that

$r'' \xrightarrow{rt(t1.v')} r'$ and x', u1 and u2 such that

$$\begin{array}{ccc} tr(r') & & x'' \\ & {\scriptstyle u1}\searrow \quad \swarrow{\scriptstyle u2} & \\ & x' & \end{array} \qquad \text{with } rt(u1)=\emptyset \text{ and } u2 \in M^*$$

But now we have:

a. $r \xrightarrow{rt(t)} r'' \xrightarrow{rt(t1.v')} r'$ and so $r \xrightarrow{rt(t.v)} r'$ and so $r \xrightarrow{rt(u)} r'$.

b. $tr(r') \xrightarrow{u1} x'$ with $rt(u1)=\emptyset$.

c. $x \xrightarrow{t3} x'' \xrightarrow{u2} x'$ and so $x \xrightarrow{t3.u2} x'$ with $t3.u2 \in M^*$.

as required. □

This lemma together with P0 now enables us to prove that for any terminating computation from tr(r) in Π_2 there must exist a

terminating computation from r in T_1 with corresponding initial and terminal configurations.

Lemma 5.4

$tr(r) \xrightarrow[2]{u} x$, $x \varepsilon T_2$ implies that there exists $r' \varepsilon T_1$ such that $r \xrightarrow[1]{rt(u)} r'$ and $tr(r')=x$.

Proof. Applying the above lemma and noticing that $x \varepsilon T_2$, we have that there exist $r' \varepsilon \Gamma_1$, $rt(u_1)=\emptyset$ such that $r \xrightarrow[1]{rt(u)} r'$ and $tr(r') \xrightarrow[2]{u_1} x$. By P0 we have $r' \varepsilon T_1$ and so $tr(r') \varepsilon T_2$, thus $u_1=\emptyset$ and $tr(r')=x$. □

Corollary 5.2

$$R(tr(r)) \subseteq tr(R(r)).$$

Proof. We need to prove that if $tr(r) \xrightarrow{*} x$, $x \varepsilon T_2$ then there must be an $r' \varepsilon T_1$ such that $r \xrightarrow{*} r'$ and $tr(r')=x$, and this is immediate from lemma 5.4.

We have proved that if tr is adequate then condition A1 given in the definition of correct translation holds. Now we can show that condition A3 holds.

Lemma 5.5

There exists an infinite computation from tr(r) iff there exists an infinite computation from r.

Proof. For the "if" part: Suppose

$$r \xrightarrow[1]{\lambda_1} r_1 \xrightarrow[1]{\lambda_2} r_2 \longrightarrow \ldots$$

is an infinite computation from r. According to P1 we have

$$tr(r) \xrightarrow[2]{u_1} tr(r_1) \xrightarrow[2]{u_2} tr(r_2) \longrightarrow \ldots$$

where $rt(u_i)=\lambda_i$ $i=1,2,\ldots$. Since the first computation is infinite,

the computation from tr(r) is infinite.

For the "only if" part: Suppose there exists an infinite computation from tr(r). According to P3 there exist $\lambda_1 \varepsilon \Lambda_1$, $r_1 \varepsilon \Gamma_1$ such that $r \xrightarrow[1]{\lambda_1} r_1$ and $tr(r')\uparrow$. Similarly for tr(r') we get $\lambda_2 \varepsilon \Lambda_1$, $r_2 \varepsilon \Gamma_1$ and $r_1 \xrightarrow[1]{\lambda_2} r_2$, $tr(r_2)\uparrow$. By repeating this procedure an infinite computation from r can be constructed. □

Finally, we turn to proving A2.

Lemma 5.6

If $r \varepsilon D_1$ then $tr(r) \varepsilon$ Semi-D_2.

Proof. It is easy to see that all computations from tr(r) are finite, otherwise according to condition P3 we could find $r' \varepsilon \Gamma_1$, $\lambda \varepsilon \Lambda_1$ such that $r \xrightarrow{\lambda} r'$ and this contradicts $r \varepsilon D_1$.

We prove that if $tr(r) \xrightarrow[2]{u} x$ is a complete computation then $x \not\varepsilon T_2$. Suppose that this is not true; since tr is adequate, either $rt(u) = \emptyset$ then $r \varepsilon T_1$ by P0 or $rt(u) \neq \emptyset$ and there must exist $r' \varepsilon T_1$ such that $r \xrightarrow[1]{rt(u)} r'$ by lemma 5.3. Both of these contradict the condition $r \varepsilon D_1$, so therefore $x \varepsilon D_2$ and $tr(r) \varepsilon$ Semi-D_2. □

Lemma 5.7

$$tr(\mathbb{K}(r)) \subseteq \text{Semi-}D_2 \quad \text{and} \quad \mathbb{K}(tr(\mathbb{K}(r))) \subseteq \mathbb{K}(tr(r))$$

Proof. The first is obvious from lemma 5.6 since $\mathbb{K}(r) \subseteq D_1$. For the second assume $x \varepsilon \mathbb{K}(tr(\mathbb{K}(r)))$, that is there is $r' \varepsilon D_1$ such that $r \xrightarrow{*} r'$ and $tr(r') \xrightarrow{*} x$ and $x \varepsilon D_2$. By lemma 5.1 we have $tr(r) \xrightarrow{*} tr(r') \xrightarrow{*} x$, and this means $x \varepsilon \mathbb{K}(tr(r))$. □

Lemma 5.8

If $tr(r) \varepsilon$ Semi-D_2 then $r \varepsilon$ Semi-D_1.

Proof. Suppose $tr(r) \varepsilon$ Semi-D_2. First, there is no infinite

computation from r; otherwise by lemma 5.5 there would be an infinite computation from tr(r).

Second, we prove that there is no complete computation $r \xrightarrow[1]{w} r'$ where $r' \varepsilon T_1$. Suppose this is not true; by lemma 5.2 $tr(r) \xrightarrow[2]{u} tr(r')$ for some $u \varepsilon \Lambda_2^*$ and by P0 $tr(r') \varepsilon T_2$. Thus there is at least one computation where tr(r) does not deadlock, and this contradiction assures that $r \varepsilon$ Semi-D_1. □

Lemma 5.9

$$\mathbb{K}(tr(r)) \subseteq \mathbb{K}(tr(\mathbb{K}(r)))$$

Proof. Suppose $tr(r) \xrightarrow[2]{u} x$ where $x \varepsilon D_2$. We need to prove that there exist $r' \varepsilon D_1$ such that $r \xrightarrow{*} r'$ and $tr(r') \xrightarrow{u'} x$. According to lemma 5.3 there exist $r' \varepsilon \Gamma_1$ and $rt(u_1) = \emptyset$ such that $r \xrightarrow{rt(u)} r'$ and $tr(r') \xrightarrow[2]{u_1} x$.

By P0, r' is not in T_1 since otherwise tr(r') is in T_2 and so $u_1 = \emptyset$ and x is in T_2 contradicting the fact that x is in D_2.

By P5, r' cannot be active, otherwise for some x' and u" we have $x \xrightarrow{u''} x'$ and $rt(u'') \neq \emptyset$ again contradicting $x \varepsilon D_2$. So r' is in D_1 and the proof is finished. □

Having proved these lemmas and corollaries we prove our main theorem:

Theorem 5.4

If the translation tr is adequate, then it is correct.

Proof. The proof follows directly from corollaries 5.1, 5.2 and lemmas 5.5, 5.7 and 5.9. □

Sometimes the following lemma can be very helpful for proving a

translation satisfies condition P3.

Lemma 5.10

If a translation tr satisfies P0, P2 and P4 and every infinite computation from tr(r) contains at least one translated label, then tr satisfies P3.

Proof. We need to prove for any tr(r) such that tr(r)↑ there exist r' and λ such that $r \xrightarrow{\lambda} r'$ and tr(r')↑. Let

$$tr(r)=x_0 \xrightarrow{\rho_1} x_1 \xrightarrow{\rho_2} x_2 \xrightarrow{\rho_3} \cdots$$

be an infinite computation from tr(r), and let ρ_k be a translated label, i.e. $\rho_k=tr(\lambda)$ for some $\lambda \varepsilon \Lambda_1$. Applying P2 we find r', x'_k, u_1 (with $rt(u_1)=\emptyset$) and $u''_k \varepsilon M^*$ (see figure 5.10), such that $r \xrightarrow{\lambda} r'$ and

$$tr(r') \xrightarrow{u_1} x'_k \quad \text{and} \quad x_k \xrightarrow{u''_k} x'_k$$

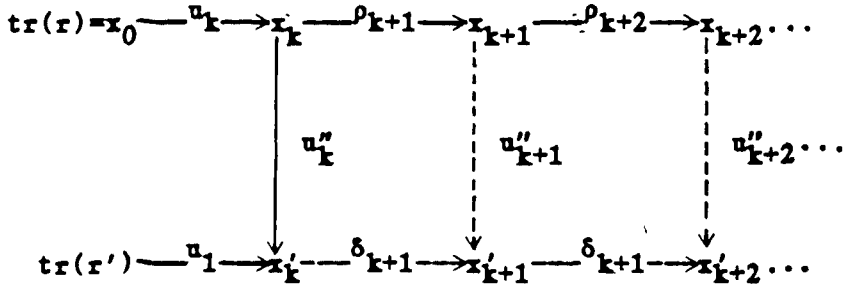


Figure 5.10

Now for

$$x_k \xrightarrow{\rho_{k+1}} x_{k+1} \quad \text{and} \quad x_k \xrightarrow{u''_2} x'_k$$

by P4 and lemma 1.1 we can find x'_{k+1} and u''_{k+1} and δ_{k+1} such that

$$x'_k \xrightarrow{\delta_{k+1}} x'_{k+1} \quad \text{and} \quad x_{k+1} \xrightarrow{u''_{k+1}} x'_{k+1}$$

where $fil_M(\rho_{k+1})=fil_M(\delta_{k+1})$ and $\delta_{k+1} \neq \emptyset$ and u''_{k+1} is a subsequence of u''_k so $u''_{k+1} \varepsilon M^*$ (see figure 5.10). By repeating this procedure we have

an infinite computation from tr(r') and so the lemma is proved. □

Finally, we might expect that an adequate translation would satisfy the following property:

P6. If $tr(r) \xrightarrow[2]{u} tr(r')$ for $u \in \Lambda_2^*$ where $rt(u) \neq \emptyset$ then $r \xrightarrow[1]{rt(u)} r'$.

Unfortunately, this is not the case as the following example shows:

Example 5.9

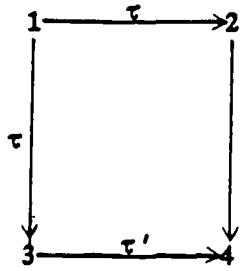
Let $\Pi_1 : \langle \Gamma_1, T_1, \Lambda_1, \xrightarrow[1]{} \rangle$ where $\Gamma_1 = \{1, 2, 3\}$, $T_1 = \emptyset$, $\Lambda_1 = \{\tau\}$ and $\xrightarrow[1]{}$ is defined by

$$1 \xrightarrow[1]{\tau} 2$$

Let $\Pi_2 : \langle \Gamma_2, T_2, \Lambda_2, \xrightarrow[2]{} \rangle$ be defined by

$\Gamma_2 = \{1, 2, 3, 4\}$, $T_2 = \emptyset$, $\Lambda_2 = \{\tau, \tau'\}$, and $\xrightarrow[2]{}$ is defined by

$$1 \xrightarrow[2]{\tau} 2$$
$$1 \xrightarrow[2]{\tau} 3$$
$$2 \xrightarrow[2]{\tau'} 4$$
$$3 \xrightarrow[2]{\tau'} 4$$



Let $tr : \Pi_1 \to \Pi_2$ be defined by

$$tr(1) = 1$$
$$tr(2) = 2$$
$$tr(3) = 3$$
$$tr(\tau) = \tau$$

Let $M = \{\tau'\}$. It is easy to see that tr is an M-adequate translation but the following transition in Π_2

$$1 \xrightarrow[2]{\tau} 3$$

shows that it does not satisfy P6 since the corresponding transition is not true in T_1. However we have already seen that P6 is not an important condition for a translation to be correct. □

Now we have set up the beginnings of the operational translation theory which will be used in the next two chapters. The theory tells us that if we want to prove that a translation is correct we only need to prove that it is adequate, i.e. that the translation satisfies the conditions P0 to P5. In the next chapter we will study a important translation from CSP to CCS and use our theory to prove that it is correct.

6. Translating CSP to CCS

The purpose of this chapter is to demonstrate some techniques for proving that a translation algorithm is correct. To do so, we study a useful example --- a syntax directed translation algorithm from CSP to CCS and prove it is correct. More precisely,we will consider a restricted subset of the CSP studied in chapter 2 (but still including all the essential features of Hoare's original proposal) and a slightly amended CCS with transition system $\mathbb{T}_c=\langle \Gamma_c, T_c, \Lambda_c, \xrightarrow[c]{} \rangle$ where T_c will be subset of Γ_c chosen to correspond to the possible final CSP computations and Λ_c remains to be specified.

Our strategy used here is that we first introduce an intermediate CCS called CCS'_A, then give a translation from CSP to CCS'_A and a translation from CCS'_A to CCS, and composing the two translations we obtain a translation from CSP to CCS; furthermore we prove the first translation is correct indirectly by proving its adequacy and prove the second is correct directly. Then the composition theorem of chapter 5 ensures the composition is correct.

Generally speaking, our proof process consists of the following four steps:

1. Given two languages L_1 and L_2 define a (syntactic) translation

   $[\![\ ]\!]: L_1 \rightarrow L_2$

2. Define two transition systems (operational semantics) $\mathbb{T}_1$ and $\mathbb{T}_2$ for L_1 and L_2 respectively.

3. From $[\![\ ]\!]$ generate a (semantic) translation between two transition systems:

   $tr: \mathbb{T}_1 \rightarrow \mathbb{T}_2$

4. Prove the translation tr is adequate.

In section 6.1 an intermediate CCS called CCS'_A is introduced and a translation from CCS'_A to CCS is given and proved to be correct. In section 6.2 we deal with both the syntactic and semantic translation from CSP to CCS'_A. In section 6.3 we prove this translation is adequate.

Finally it should be mentioned that in [Hennessy, Li and Plotkin 81] a translation from a simple CSP (without nested parallel structure) is given and is proved correct. In this chapter we study a more general case, e.g. nested parallel structure is allowed.

6.1 An intermediate CCS

Rather than translate directly to CCS we consider a language CCS'_A called intermediate CCS, where $A \subseteq \Delta$ (for our purposes A will be the set of translated labels). The syntax of CCS'_A is just the same as CCS but with a new kind of term $\tau'.t$.

The transition system $\mathbb{T}_t$ for CCS'_A is defined by:

$$\mathbb{T}_t = \langle \Gamma_t, T_t, \Lambda_t, \xrightarrow[t]{} \rangle$$

where Γ_t is the set of CCS'_A terms and $T_t = T_c$ and $\Lambda_t = \Lambda_c \cup \{\tau'\}$.

The transition relation $\xrightarrow[t]{}$ is the same as $\xrightarrow[c]{}$ with the following changes (see section 1.4):

action 3. $\tau.t \xrightarrow{\tau} t$

4. $\tau'.t \xrightarrow{\tau'} t$

composition

$$3.\ \frac{t \xrightarrow{\lambda} t',\ u \xrightarrow{\bar{\lambda}} u'}{t|u \xrightarrow{\tau'} t'|u',\ u|t \xrightarrow{\tau'} u'|t'}$$

if $\lambda \neq \tau$ and the line name of λ is in A

$$4.\ \frac{t \xrightarrow{\lambda} t',\ u \xrightarrow{\bar{\lambda}} u'}{t|u \xrightarrow{\tau} t'|u',\ u|t \xrightarrow{\tau} u'|t'} \quad \text{otherwise.}$$

Rule 3 means that a composition is transformed by a marked τ' if the line name of the transition label (therefore its complement) of its component is in A. Rule 4 is the same as rule 3 in section 1.4.3. The motivation for introducing these rules is to distinguish translated actions from actions which denote interaction with simulated memory. We will see this point later.

The only difference between T_c and T_t is that $\xrightarrow{}_t$ involves a marked transition label τ'. Define the obvious translation $tr': T_t \rightarrow T_c$, which just removes the dashes. We assume that for any t in Γ_t $tr'(t) \in T_c$ implies $t \in T_t$ and will check this later. We have the following result:

Lemma 6.1

1. If $t1 \xrightarrow{\lambda} t2$ then $tr'(t1) \xrightarrow{tr'(\lambda)} tr'(t2)$.

2. If $tr'(t1) \xrightarrow{\rho} x$ then there exist λ and t2 such that $t1 \xrightarrow{\lambda} t2$ and $x = tr'(t2)$ and $tr'(\lambda) = \rho$.

3. $t \in T_t$ iff $tr'(t) \in T_c$ and $t = tr'(t)$.

Proof. The proof is by structural induction on t1. □

Lemma 6.2

Let $u \in \Lambda_c^*$.

1. If $t1 \xrightarrow{u} t2$ then $tr'(t1) \xrightarrow{tr'(u)} tr'(t2)$.

2. If $tr'(t1) \xrightarrow{v} x$ then there are u and t2 such that $t1 \xrightarrow{u} t2$ and $x = tr'(t2)$ and $tr'(u) = v$.

3. t is deadlocked iff tr'(t) is deadlocked.

Proof. Clauses 1 and 2 are from lemma 6.1 by induction on the lengths of u and v respectively. Clause 3 is clear. □

Theorem 6.1

The translation tr' is correct.

Proof. A1. Suppose $x \in tr'(\mathbb{R}(t))$. Then $t \xrightarrow{*} t'$ and $t' \in T_t$ and $tr'(t') = x$. By lemma 6.2 we have $tr(t) \xrightarrow{*} tr'(t') = x \in T_c$ and so $x \in \mathbb{R}(tr'(t))$. Therefore $tr'(\mathbb{R}(t)) \subseteq \mathbb{R}(tr'(t))$.

Conversely, suppose $x \in \mathbb{R}(tr'(t))$. Then $tr'(t) \xrightarrow{*} x \in T_c$. By lemma 6.2 $t \xrightarrow{*} t'$ and $x = tr'(t')$ for some t'. But $t' = tr'(t') = x \in T_t$ so $t' \in T_c$. So $x \in tr'(\mathbb{R}(t))$. Thus $\mathbb{R}(tr'(t)) \subseteq tr'(\mathbb{R}(t))$.

A2. As anything in $\mathbb{K}(t)$ is deadlocked so is anything in $tr'(\mathbb{K}(t))$ by lemma 6.2. So indeed $tr'(\mathbb{K}(t)) \subseteq \text{semi-}D_t$ and indeed $\mathbb{K}((tr'(\mathbb{K}(t))) = tr'(\mathbb{K}(t))$.

A3. This is straightforward using lemma 6.2. □

This theorem tells us that given a language if we want to find a correct translation from L to CCS we only need to give a translation tr from L to CCS'_A for some A and prove it is correct. For then the composition $tr' \circ tr$ defines a translation from L to CCS and the composition theorem ensures it is correct. This is the strategy we shall adopt in this chapter.

6.2 Translation from CSP to CCS

6.2.1 Useful notation

First we introduce some useful terms which are used to describe a translation algorithm, tr. As we go along we shall introduce various constraints on the set Δ, of the line names needed to fix exactly which CCS'_A is intended as the range of our translation. It will be convenient to assume that all the CSP programs under consideration use variables in $X=\{x_1,\ldots,x_n\}$ (by making n large enough we can include any finite set of programs).

A. A store

For each variable $x_i \varepsilon Var$ a term Reg_i is defined to model a register which is used to hold the (current) value of x. Such registers are defined in [Milner 80], using parameterised recursive definitions. Reg_i contains two lines: α_i for writing a new value into the register and β_i for reading this valuc. Thus if $Reg_i(v)$ represents the register which contains the value v then we should have:

$$Reg_i(v) \xrightarrow{\beta_i!v} Reg_i(v)$$
$$Reg_i(v) \xrightarrow{\alpha_i?v'} Reg_i(v')$$

and these are the only transitions. To obtain this effect let $Reg_i(e)$ be the term:

$$[\mu R(v).(\beta_i(v,R(v))+\alpha_i v'.R(v'))](e)$$

We may assume that $Stores=[X\rightarrow V]$ and for any store s in Stores let M_s denote the term

$$\Big|_{i=1..n} \mathrm{Reg}_i(s(x_i))$$

The term M_s simulates the store s; from now on we assume that α_i, β_i i=1,...,n are distinguished line names.

Let $X_j=\{x_{j1},x_{j2},\ldots,x_{jk}\}\subseteq X$ and tεTerms. The notation

$RS_{X_j}[\ t\ ]$ is used to denote $\beta_{j1}x_{j1}.(\beta_{j2}x_{j2}.(\ldots(\beta_{jk}x_{jk}.t)\ldots))$

The use of the form RS_{X_j} is for reading the set X_j from the simulated store M_s.

B. Auxilliary terms

Let σ, δ be distinguished line names in Δ. The following terms and constructs will prove useful

$$\underline{\mathrm{done}} = \sigma(0,\mathrm{Nil})$$

$$\underline{\mathrm{fail}} = \delta(0,\mathrm{Nil})$$

$$t\ \underline{\mathrm{before}}\ u = (t[\pi]\,|\,\gamma_1x.u + \gamma_2x.\underline{\mathrm{fail}})\backslash\{\gamma_1,\gamma_2\} \quad (x \text{ is not in } u)$$

$$t\ \underline{\mathrm{par}}\ u = (t[\pi]\,|\,u[\pi]\,|\,\mathrm{parx})\backslash\{\gamma_1,\gamma_2\}$$

where

$$\mathrm{parx} = \gamma_1x.(\gamma_1x.\tau'.\underline{\mathrm{done}} + \gamma_2x.\underline{\mathrm{fail}}) + \gamma_2x.(\gamma_1x.\tau'.\underline{\mathrm{fail}} + \gamma_2x.\underline{\mathrm{fail}})$$

and

$$\pi = \{\gamma_1/\sigma,\gamma_2/\delta\}$$

and γ_1, γ_2 $(\gamma_1\neq\gamma_2)$ do not occur freely in t and u. Strictly speaking, we should insist on a particular choice of γ_1 and γ_2 and

include α-conversion in the congruence, ~, to allow for the fact that the choice of γ_1 and γ_2 will depend on t and u.

Note that τ' appears in the term par and will correspond to ε given in CSP. The lines σ and δ in the terms before and par are restricted, thus the term t before u means that if the execution of t is finished then a signal is sent through the (restricted) line σ and the term t before u is transformed to u via an internal action τ. If the execution of t leads to fail then a signal is sent through the (restricted) line δ and t before u is transformed to fail via an internal action. Similarly, for the term t par u if one of t and u is done then t par u becomes the other followed by τ'.done; if one of t and u is failed then the behaviour of t par u is the same as the other followed by a τ'.fail. All these explanations will be described formally in the next section.

6.2.2 Hoare's CSP

To translate Hoare's CSP, as we explained in chapter 2, we need to restrict the static semantics for guarded commands. To do so, we introduce the set, Bsc, of basic commands ranged over by BC and defined as follows:

Bsc = { skip, P?W(x), P!W(e) }

That is a basic command can only be skip or an input or output command. Then the syntax and static semantics for Hoare's guarded commands commands are defined by:

GC ::= b⇒BC;C | GC▯GC

The forms of guarded commands now can only be

b⇒skip;C or b⇒P?W(x);C or b⇒P!W(e);C or GC▯GC

as we discussed in section 3.1 they are the same as

$$b \rightarrow C \quad \text{or} \quad b, P?W(x) \rightarrow C \quad \text{or} \quad b, P!W(e) \rightarrow C \quad \text{or} \quad GC \,[]\, GC.$$

Given a guarded command GC, the set GV(GC) of free guard variables contained in the guard of GC is defined by:

	$b \Rightarrow BC; C$	$GC1 \,[]\, GC2$
GV	$FV(b) \cup FV(BC)$	$GV(GC1) \cup GV(GC2)$

The static semantics of these guarded commands are given below:

$$\frac{\vdash_c BC, \ \vdash_c C}{\vdash_{gc} b \Rightarrow BC; C} \quad \text{if } GV(b \Rightarrow BC; C) = \emptyset$$

$$\frac{\vdash_{gc} GC1, \ \vdash_{gc} GC2}{\vdash_{gc} GC1 \ [] \ GC2}$$

Note the static semantics for other commands are the same as before.

Remarks We assume that all CSP commands concerned in this chapter are syntactically correct.

Notation 6.1

Let s be a state.

The forms s(e) and s(Bool(GC)) denote the substitution of free variables in e and Bool(GC) by s respectively.

The forms s(BC) and s(GC) denote the substitution of free guarded variables by s respectively. []

Finally, to prove the translation is correct we need to modify the parallel rule given in chapter 2 as follows:

$$\underline{\text{parallel-B}}\ 1.3.\ \frac{\langle C_1, s\rangle \xrightarrow{\lambda} s' \mid \underline{\text{abortion}}}{\langle C_1 \| C_2, s\rangle \xrightarrow{\lambda} \langle C_2; \underline{\text{skip}}, s'\rangle \mid \langle C_2; \underline{\text{abort}}, s\rangle}$$

$$2.3.\ \frac{\langle C_2, s\rangle \xrightarrow{\lambda} s' \mid \underline{\text{abortion}}}{\langle C_1 \| C_2, s\rangle \xrightarrow{\lambda} \langle C_2; \underline{\text{skip}}, s'\rangle \mid \langle C_1; \underline{\text{abort}}, s\rangle}$$

The difference from the parallel rules as given in section 2.3 is that if one of the constituents aborts, the parallel command may not abort immediately as the other may not terminate.

6.2.3 Syntactic translation

For convenience we use the suggestive line names $[N,N']W$ throughout the translation, where $N, N' \in Plab \cup \{*\}$ and $W \in Ptn$. Intuitively, $[N,N']W$ denotes the name of a communication line through which process N sends a value to process N' with a pattern W. For simplicity it is also assumed that all expressions can be evaluated properly, i.e, the case $[\![e]\!]_s = \underline{\text{error}}$ can not happen.

Having introduced the necessary notation and assumptions we now study the syntactic translation from CSP to CCS'_A. This translation consists of the two types of translations:

$(\!|\ |\!) : Bsc \rightarrow Terms$

$[\![\ ]\!] : Syn \rightarrow Terms$

defined below:

Basic commands

1. $(\!|\underline{\text{skip}}|\!) = [\![\underline{\text{skip}}]\!]$

2. $(\!|P?W(x_i)|\!) = [\![P?W(x_i)]\!]$

3. $(\!|Q!W(e)|\!) = [*,Q]W(e,[\![\underline{skip}]\!])$

Guarded commands

1. $[\![b \Rightarrow BC;C]\!] = \underline{if}\ b\ \underline{then}\ (\!|BC|\!)\underline{before}[\![C]\!]\ \underline{else}\ Nil$

2. $[\![GC_1 [] GC_2]\!] = [\![GC_1]\!] + [\![GC_2]\!]$

Commands

1. $[\![\underline{skip}]\!] = \tau'.\underline{done}$

2. $[\![\underline{abort}]\!] = \tau'.\underline{fail}$

3. $[\![x_i:=e]\!] = \tau'.RS_{FV(e)}[\alpha_i(e, [\![\underline{skip}]\!])]$

4. $[\![C_1;C_2]\!] = [\![C_1]\!]\underline{before}[\![C_2]\!]$

5. $[\![P?W(x_i)]\!] = [P,*]Wx.\alpha_i(x,[\![\underline{skip}]\!])$

6. $[\![Q!W(e)]\!] = RS_{FV(e)}[[*,Q]W(e,[\![\underline{skip}]\!])]$

7. $[\![\underline{if}\ GC\ \underline{fi}]\!] = RS_{GV(GC)}[\underline{if}\ Bool(GC)\ \underline{then}\ [\![GC]\!]\ \underline{else}\ [\![\underline{abort}]\!]]$

8. $[\![\underline{do}\ GC\ \underline{od}]\!] = \mu P.RS_{GV(GC)}[\underline{if}\ Bool(GC)\ \underline{then}\ ([\![GC]\!]\underline{before}\ P)\ \underline{else}\ [\![\underline{skip}]\!]]$

9. $[\![C_1 \| C_2]\!] = [\![C_1]\!]\ \underline{par}\ [\![C_2]\!]$

10. $[\![R::C]\!] = [\![C]\!][\phi_R]$

where ϕ_R is the renaming defined by:

$$\phi_R(\alpha)=\begin{cases}[R,P]W & \text{if } \alpha=[*,P]W \text{ and } R\neq P\\ [P,R]W & \text{if } \alpha=[P,*]W \text{ and } R\neq P\\ \alpha & \text{if } \alpha \text{ is one of } \tau',\ \tau,\ \alpha_i!v,\ \beta_i?v,\ \sigma!0 \text{ and } \delta!0\end{cases}$$

11. $[\![\underline{\text{process}}\ R;C]\!] = [\![C]\!][\eta_{R,FPL(C)}]$

where $\eta_{R,L}$ is the renaming defined by:

$$\eta_{R,L}(\alpha)=\begin{cases}[*,P]W & \text{if } \alpha=[R,P]W \text{ and } P\notin L\\ [P,*]W & \text{if } \alpha=[P,R]W \text{ and } P\notin L\\ \text{undefined} & \text{if } \alpha=[R,P]W \text{ and } P\in L\\ \text{undefined} & \text{if } \alpha=[P,R]W \text{ and } P\in L\\ \alpha & \text{if } \alpha \text{ is } [P,N]W \text{ or } [N,P]W \text{ where } R\neq N, R\neq P\\ \alpha & \text{if } \alpha \text{ is one of } \tau',\ \tau,\ \alpha_i!v,\ \beta_i?v,\ \sigma!0 \text{ and } \delta!0\end{cases}$$

Let us explain the meaning of $\eta_{R,L}$ informally:

Suppose $\alpha=[R,P]W$. Together with the translations of R::C and P!W(e) and R?W(x_i) we know that there are only two possibilities:

a. A command R::C' occurs in C and a command P!W(e) occurs in C'. In this case if $P\notin FPL(C)$, i.e. the command P::C'' does not occur in C, then from the outside of $[\![\underline{\text{process}}\ R;C]\!]$ this action must be [*,P]W!v since process R;C defines the scope of R. Thus the line [R,P]W becomes [*,P]W and this is the first case of the definition. If $P\in FPL(C)$, i.e. a command P::C'' does occur in C, then by the scope rule the action [R,P]W!v must be invisible from the outside of $[\![\underline{\text{process}}\ R;C]\!]$. So $\eta_{R,L}([R,P]w)$ is undefined and this is the third case of the definition.

b. A command P::C' occurs in C and a command R?W(x_i) occurs in

C'. In this case P≤FPL(C) and according to the scope rule the action [R,P]W?v must be not visible from the outside of [[process R;C]]. This is also the third case of the definition.

Similarly we can check the remaining cases of the definition. To make sure the translation is syntactically correct we need the following notation:

Definition 6.1

Let $M_\alpha=\{ \alpha_i \mid i=1,\ldots,n\}$ and $M_\beta=\{ \beta_i \mid i=1,\ldots,n\}$.

The function read:$X\rightarrow M_\beta$ is defined by:

$$read(x_i)=\beta_i \quad i=1,\ldots,n$$

The function write:$X\rightarrow M_\alpha$ is defined by:

$$write(x_i)=\alpha_i$$

and $M=M_\alpha \cup M_\beta \cup \{\sigma,\delta\}$ □

By structural induction we can prove the following results:

Lemma 6.3

a. If $\vdash \Omega$ then

$$read(RV(\Omega))=FL([\![\Omega]\!]) \cap M_\beta$$

$$write(WV(\Omega))=FL([\![\Omega]\!]) \cap M_\alpha$$

b. If $\vdash C_1 \| C_2$ then

$$FL([\![C_1]\!]) \cap FL([\![C_2]\!]) \cap M_\alpha=\emptyset$$

c.
$$FV([\![BC]\!]) = RV(BC)$$

$$FV([\![GC]\!]) = GV(GC)$$

$$FV([\![C]\!]) = \emptyset$$

$$[\![s(GC)]\!] = s[\![GC]\!] \quad \square$$

6.2.4 Semantic translation

We now study the adequacy of our proposed translation tr and

begin with its definition using the notation introduced above.

Firstly, we need to set up the transition systems for CSP and CCS'_A. To simplify the later proofs we take the union of the transition systems $\mathbb{T}_{gc}$ and $\mathbb{T}_c$ given in chapter 2, (of course with the constraints given above) defining

$$\mathbb{T}_p=\langle \Gamma_{gc} \cup \Gamma_c,\ T_c, \Lambda_c, \xrightarrow[gc]{} \cup \xrightarrow[c]{} \rangle$$

and this is the transition system for CSP. We will just use the identity congruence.

The transition system $\mathbb{T}_t$ of CCS'_A is defined as:

$$A = \{[N,N']W \mid N,N' \in Plab \cup \{*\},\ W \in Ptn \text{ and } N \neq * \text{ or } N' \neq * \}$$

and

$$T_t=\{(\underline{done} \mid M_s \mid R)\backslash M,\ (M_{s0} \mid \underline{fail})\backslash M \mid s \in Stores\}$$

where $s0(x_i)=0$ for all i, and R is the reset term given as:

$$R = \delta x.a_1(0,a_2(0,\ldots,a_n(0,\underline{fail})\ldots)$$

i.e. if R receives a signal from line δ then it sets the simulated state M_s to M_{s0}.

The congruence relation ~ on the transition system $\mathbb{T}_t$ is just as given in section 5.2.

The translation $tr:\mathbb{T}_p \rightarrow \mathbb{T}_t$ is defined by:

$$tr(r)=\begin{cases}(\llbracket \Omega \rrbracket \mid M_s \mid R)\backslash M & \text{if } r=\langle \Omega, s\rangle \\ (\underline{done} \mid M_s \mid R)\backslash M & \text{if } r=s \\ (M_{s0} \mid \underline{fail})\backslash M & \text{if } r=\underline{abortion}\end{cases}$$

and

$$tr(\lambda)=\begin{cases}[N,P]W!v & \text{if } \lambda=(N,P)!W(v) \\ [P,N]W?v & \text{if } \lambda=(N,P)?W(v) \\ \tau' & \text{if } \lambda=\varepsilon\end{cases}$$

Note tr on labels is an injective mapping; and we see that τ' is the translation of ε, it was the need to keep track of the ε transitions that led to the introduction of CCS'_A.

Remarks The purpose of introducing the term R is just to give injective mapping for abortion. In practice, when a program is terminated abnormally, the store (memory) is very important for users to find run-time errors. From this point of view, we should introduce $\langle\downarrow,s\rangle$ and $\langle\uparrow,s\rangle$ to replace s and abortion in the semantics of CSP respectively; and then the translation would become:

$$tr(r)=\begin{cases}(\llbracket\Omega\rrbracket \mid M_s)\backslash M & \text{if } r=\langle\Omega,s\rangle\\ (\underline{done} \mid M_s)\backslash M & \text{if } r=\langle\downarrow,s\rangle\\ (\underline{fail} \mid M_s)\backslash M & \text{if } r=\langle\uparrow,s\rangle\end{cases}$$

In fact this will make the later proofs simpler.

Finally, it should be mentioned that if we omit the restriction for the guards (see section 6.2.2) and use general Plotkin's guards then our translation is not adequate. Let us check the following example:

Example 6.1

Consider the CSP command C' with Plotkin's guards:

```
C'= R:: C ‖ P:: R?x3
```

where the command C is given by:

```
C = if
      true⇒P!0
    □
      true⇒( if x1⇒Q!0 □ not x1⇒Q?x2 fi)
    fi
```

Let the initial state be $s=\{x_1=0,x_2=0,x_3=0\}$. We know that the command C' can never deadlock. The translation of C is:

$[\![C]\!]$ = if true

then($[\![P!0]\!]+\beta 1x.$(if true then($[\![Q!0]\!]+[\![Q?x3]\!]$)else[abort])

else[abort]

and consider the following computation:

$tr(\langle C,s\rangle)=([\![C]\!]\,|M_s|\,R)\backslash M$

$\xrightarrow{\tau}$((if tt then($[\![Q!0]\!]+[\![Q?x3]\!]$)else[abort])$|M_s|$R)\M

and is deadlocked. So the computation from $tr(\langle C',s\rangle)$ can deadlock. □

6.3 Proving the adequacy

In this section we prove the translation $tr:\Pi_p\rightarrow\Pi_t$ is adequate. According to chapter 5 we need to prove tr satisfies P0 to P5. To do so, we first introduce some useful results about the terms t before u and t par u, and then prove tr satisfies P0 to P5 one by one. The method used to prove adequacy is mainly structural induction.

6.3.1 Useful lemmas

As we have already seen, in the syntactic translation we introduced the terms t before u and t par u, and in the semantic translation we introduced the form $(t|M_s|R)\backslash M$. In this subsection we will introduce some notation to help us study the properties of these terms. The proofs of these properties are given in appendix 1. First of all we need the following definitions:

Definition 6.3

Given $\gamma\varepsilon\Delta$ and $\lambda\varepsilon\Lambda_t$ and $u\varepsilon\Lambda_t^*$,

1. $\gamma \varepsilon \lambda$ iff $\lambda = \gamma!v$ or $\lambda = \gamma?v$ for some v.

2. $\lambda \varepsilon u$ iff $\lambda \varepsilon hd(u)$ or $\lambda \varepsilon tl(u)$ recursively.

Definition 6.4 Well-formed terms

A term t is <u>well-formed</u> iff whenever $t \xrightarrow{u} t' \xrightarrow{\lambda} t''$ where neither σ nor δ occur in u, and $\sigma \varepsilon \lambda$ or $\delta \varepsilon \lambda$, then t' is <u>done</u> or <u>fail</u> and $\lambda = \sigma!0$ or $\lambda = \delta!0$. (Of course then if $\sigma \varepsilon \lambda$ then t',t'' must be <u>done</u> and Nil respectively and λ must be $\sigma!0$; and if $\delta \varepsilon \lambda$ then t',t'' must be <u>fail</u> and Nil respectively and λ must be $\delta!0$.)

By structural induction we can prove the following lemma:

Lemma 6.31 Let $t \varepsilon \Gamma$ and C be a command of Hoare's CSP. If $[\![C]\!] \xrightarrow{*} t$ then t is a well-formed term.

Definition 6.5 Merge

Let u, v, w be in Λ_t^*. Then w is a <u>merge</u> of u and v (written as Merge(u,v,w)) iff u, v and w can be written for some $n \geq 0$ as

$$u = u_1 \ldots u_n$$

$$v = v_1 \ldots v_n$$

$$w = w_1 \ldots w_n$$

and for any j with $1 \leq j \leq n$ one of the following holds:

1. $w_j = u_j = \lambda$ and $v_j = \emptyset$
2. $w_j = v_j = \lambda$ and $u_j = \emptyset$
3. For some λ with label in A, $w_j = \tau'$, $u_j = \lambda$, $v_j = \bar{\lambda}$.
4. For some λ with label in $\{\sigma, \delta, \alpha_i, \beta_i, \gamma_i\}$, $w_j = \tau$, $u_j = \lambda$, $v_j = \bar{\lambda}$. □

The motivation of merge is that for a transition sequence of a composition $t_1 \| t_2 \xrightarrow{w} t_1' \| t_2'$ if we consider the "projected" transition sequences of its components we will have

$$t_1 \xrightarrow{u} t_1' \quad \text{and} \quad t_2 \xrightarrow{v} t_2'$$

Then these three transition sequences must satisfy Merge(u,v,w).

Definition 6.5

Given $u \varepsilon \Lambda_t^*$ and $s \varepsilon States$, the state $su \varepsilon States$ is defined by:

$$su = \begin{cases} s & u = \emptyset \\ su' & u = \lambda u' \quad \alpha_i \notin \lambda \text{ and } \beta_i \notin \lambda \\ su' & u = (\beta_i ? s(x_i))u' \\ s[v/x_i]u' & u = (\alpha_i ! v)u' \end{cases}$$

The motivation of the definition is that su defines the final state of a computation via the transition sequence u from the initial state s.

Definition 6.6

Given $u \varepsilon \Lambda_t$, the transition sequence $\bar{u} \varepsilon \Lambda_t$ is defined by:

$$\bar{u} = \begin{cases} \emptyset & u = \emptyset \\ \lambda \bar{u}' & u = \lambda u' \quad \alpha_i \notin \lambda \text{ and } \beta_i \notin \lambda \\ \tau \bar{u}' & u = \lambda u' \quad \alpha_i \varepsilon \lambda \text{ or } \beta_i \varepsilon \lambda \end{cases}$$

The motivation is that

$$t \xrightarrow{u} t' \quad \text{iff} \quad m(t,s) \xrightarrow{\bar{u}} m(t', su).$$

Notation 6.2

From now on we always use the following notation:

$$m(t,s) = (t | M_s | R) \backslash M$$

Notation 6.3

Given two states s_1, s_2 and finite subsets L_1, $L_2 \varepsilon \Delta$ with $L_1 \cap L_2 \cap M_\alpha = \emptyset$, we define the operation Θ on s_1 and s_2 as follows: If that s_1 and s_2 agree on those x_i for which α_i not in L_1 or L_2 then

$$s_1 \Theta s_2(x_i) = \begin{cases} s_1(x_i) & \text{if } \alpha_i \varepsilon L_1 \\ s_2(x_i) & \text{otherwise} \end{cases}$$

Let

$w=\tau\tau'\tau$ $\quad u=\tau P!0\emptyset$ $\quad v=\emptyset P?0\tau$

$w'=\beta_1?0.\tau'.\alpha_2!0$ $\quad u'=\beta_1?0.P!0.\emptyset$ $\quad v'=\emptyset.P?0.\alpha_2!0$

Then have

$w=\bar{w}'$ $\quad u=\bar{u}'$ $\quad v=\bar{v}'$

and

Merge(u,v,w) and Merge(u',v',w')

and

$su'=s$ $\quad sv'=s'$ and $s'=su'\theta sv'$ □

Lemma 6.4

Let t be well-formed and $w\varepsilon\Lambda_t^*$. Then $m(t,s)\xrightarrow{w}x$ iff one of the following holds:

a. For some t', u and s' we have $t\xrightarrow{u}t'$ and $s'=su$ and $w=\bar{u}$ and $x=m(t',s')$ and neither $\sigma\varepsilon u$ nor $\delta\varepsilon u$.

b. For some u, u'and s' we have σ, $\delta\notin u$ and
$t\xrightarrow{u}\underline{fail}$ and $w=\bar{u}u'$ and $s'=su$ and
$m(t,s)\xrightarrow{\bar{u}}m(\underline{fail},s')\xrightarrow{u'}x$.

(and of course $u'\varepsilon\{\tau\}^*$) □

Lemma 6.5

Let t_1 and t_2 be well-formed with no free occurrences of γ_i. Then $m(t_1\underline{\text{before}}\ t_2,s)\xrightarrow{w}x$ iff one of the following holds:

a. For some t_1' and s' we have:
$m(t_1,s)\xrightarrow{w}m(t_1',s')$ and $x=m(t_1'\underline{\text{before}}\ t_2,s')$

b. For some and u, u' and s' we have:
$m(t_1,s)\xrightarrow{u}m(\underline{done},s')$ and $m(t_2,s')\xrightarrow{u'}x$ and $w=u\tau u'$.

c. For some u, u' and s' we have:
$m(t_1,s)\xrightarrow{u}m(\underline{fail},s')\xrightarrow{u'}x$ and $w=u\tau u'$. □

Lemma 6.6

Let t_1 and t_2 be well-formed with no free occurrences of γ_i and $FL(t_1) \cap FL(t_2) \cap M_\alpha = \emptyset$. Then $m(t_1 \underline{\text{par}}\ t_2, s) \xrightarrow{w} x$ iff one of the following holds:

a. For some u, v, and w' we have: $\bar{w}' = w$ and

$x = m(t_1' \underline{\text{par}}\ t_2', sw')$ and $\text{Merge}(u,v,w')$ and $sw' = su\theta sv$ and $m(t_1, s) \xrightarrow{\bar{u}} m(t_1', su)$ and $m(t_2, s) \xrightarrow{\bar{v}} m(t_2', sv)$.

b. For some w', w" and t_2' we have $w = \bar{w}'\tau w"$ and:

$m(t_1 \underline{\text{par}}\ t_2, s) \xrightarrow{\bar{w}'} m(\underline{\text{done}} \underline{\text{par}}\ t_2', sw')$ and
$m(t_2' \underline{\text{before}}\ \underline{\text{done}}, sw') \xrightarrow{w"} x$
and for some u, v we have $\text{Merge}(u,v,w')$ and $sw' = su\theta sv$ and
$m(t_1, s) \xrightarrow{\bar{u}} m(\underline{\text{done}}, su)$ and $m(t_2, s) \xrightarrow{\bar{v}} m(t_2', sv)$

c. For some w', w" and t_1', s' we have $w = \bar{w}'\tau w"$ and

$m(t_1 \underline{\text{par}}\ t_2, s) \xrightarrow{\bar{w}'} m(t_1' \underline{\text{par}}\ \underline{\text{done}}, sw')$ and
$m(t_1' \underline{\text{before}}\ \underline{\text{done}}, sw') \xrightarrow{w"} x$
and for some u, v we have $\text{Merge}(u,v,w')$ and $sw' = su\theta sv$ and
$m(t_1, s) \xrightarrow{\bar{u}} m(t_1', su)$ and $m(t_2, s) \xrightarrow{\bar{v}} m(\underline{\text{done}}, sv)$

d. For some w', w" and t_2' we have $w = \bar{w}'\tau w"$ and

$m(t_1 \underline{\text{par}}\ t_2, s) \xrightarrow{\bar{w}'} m(\underline{\text{fail}} \underline{\text{par}}\ t_2', sw')$ and
$m(t_2' \underline{\text{before}} [\![\underline{\text{abort}}]\!], sw') \xrightarrow{w"} x$
and for some u, v we have $\text{Merge}(u,v,w')$ and $sw' = su\theta sv$
and $m(t_1, s) \xrightarrow{\bar{u}} m(\underline{\text{fail}}, su)$ and $m(t_2, s) \xrightarrow{\bar{v}} m(t_2', sv)$

e. For some w', w" and t_1' we have $w = \bar{w}'\tau w"$ and

$m(t_1 \ \underline{par}\ t_2, s) \xrightarrow{\bar{w}'} m(t_1' \ \underline{par}\ \underline{fail}, sw')$ and

$m(t_1' \ \underline{before}[\![\underline{abort}]\!], sw') \xrightarrow{w''} x$

and for some u, v we have Merge(u,v,w') and $sw' = su \Theta sv$

and $m(t_1, s) \xrightarrow{\bar{u}} m(t_1', su)$ and $m(t_2, s) \xrightarrow{\bar{v}} m(\underline{fail}, sv)$

Lemma 6.7

Suppose $u \neq \emptyset$. Then $m(t_1 + t_2, s) \xrightarrow{u} x$ iff

either $m(t_1, s) \xrightarrow{u} x$ or $m(t_2, s) \xrightarrow{u} x$.

Lemma 6.8

Suppose $u \neq \emptyset$. Then

$m(\underline{if}\ b\ \underline{then}\ t\ \underline{else}\ u, s) \xrightarrow{u} x$ iff

either $[\![b]\!] = tt$ and $m(t, s) \xrightarrow{u} x$ or $[\![b]\!] = ff$ and $m(u, s) \xrightarrow{u} x$

Lemma 6.9

Let t be well-formed with no free occurrences of γ_i and $w \in \Lambda_t$ and let ϕ be any renaming which is defined on w and is the identity on σ, δ, α_i and β_i. Then $m(t[\phi], s) \xrightarrow{w} x'$ iff one of the following holds:

a. For some t' and s' we have $x' = m(t'[\phi], s')$ and there is a w' with $\phi(\bar{w}') = w$ and $m(t, s) \xrightarrow{\bar{w}'} m(t', s')$ and $s' = sw'$.

b. For some u, u' and v we have $\delta, \sigma \notin u$ and $w = uv$ and $\phi(\bar{u}') = u$ and $m(t, s) \xrightarrow{\bar{u}'} m(\underline{fail}, su')$ and $m(\underline{fail}, su') \xrightarrow{v} x'$.

6.3.2 Proving the adequacy

In this section we prove the translation $tr:\mathbb{T}_p \rightarrow \mathbb{T}_t$ is adequate with $M=\{\tau\}^*$, to do so we prove that tr satisfies P0 to P5. First of all, for convenience of later proofs we give the complete transitions for the following simple translations:

$$tr(\langle \underline{skip}, s\rangle) = m(\tau'.\underline{done}, s)$$
$$\xrightarrow{\tau'} m(\underline{done}, s) = tr(s)$$

$$tr(\langle \underline{abort}, s\rangle) = m(\tau'.\underline{fail}, s)$$
$$\xrightarrow{\tau'} m(\underline{fail}, s)$$
$$\xrightarrow{\tau^+} (M_{s0} \,|\, \underline{fail}) \backslash M = tr(\underline{abortion})$$

Here τ^+ denotes a deterministic transition sequence.

$$tr(\langle x_i = e, s\rangle) = m(\tau'.RS_{FV(e)}[\alpha_i(e, [\![\underline{skip}]\!])], s)$$
$$\xrightarrow{\tau'} m(RS_{FV(e)}[\alpha_i(e, [\![\underline{skip}]\!])], s)$$
$$\xrightarrow{\tau^*} m(\alpha_i(s(e), [\![\underline{skip}]\!]), s)$$
$$\xrightarrow{\tau} m([\![\underline{skip}]\!], s[[\![s(e)]\!]/x_i])$$
$$\xrightarrow{\tau'} m(\underline{done}, s[[\![s(e)]\!]/x_i])$$

$$tr(\langle P?W(x_i), s\rangle) = m([P,*]W?x.\alpha_i(x, [\![\underline{skip}]\!]), s)$$
$$\xrightarrow{[P,*]W?v} m(\alpha_i(v, [\![\underline{skip}]\!]), s)$$
$$\xrightarrow{\tau} m([\![\underline{skip}]\!], s[v/x_i])$$
$$\xrightarrow{\tau'} m(\underline{done}, s[v/x_i])$$

$$tr(\langle Q!W(e), s\rangle) = m(RS_{FV(e)}[[*,Q]W(e, [\![\underline{skip}]\!])], s)$$
$$\xrightarrow{\tau^*} m([*,Q]W(s(e), [\![\underline{skip}]\!])], s)$$
$$\xrightarrow{[*,Q]W![\![s(e)]\!]} m(\underline{skip}, s)$$
$$\xrightarrow{\tau'} m(\underline{done}, s)$$

The renamings ϕ_R and $\eta_{R,L}$ have the following properties:

Lemma 6.10

For any $\lambda \varepsilon \Lambda_t$ we have

$$rt(\phi_R(\lambda)) = \phi_R(rt(\lambda))$$

with one side being defined iff the other is.

Proof. The proof is by case analysis on the definition.

Lemma 6.11

Let L be a finite subset of Plab. For any $\lambda \varepsilon \Lambda_t$ we have

$$rt(\eta_{R,L}(\lambda)) = \eta_{R,L}(rt(\lambda))$$

with one side being defined iff the other is.

Proof. The proof is by case analysis on the definition.

We assume that the definition of the functions ϕ_R and $\eta_{R,L}$ are extended to the sequences of transition actions in both CSP and CCS'_A. As usual, we use Syn to denote the set of syntactic entities of CSP.

Theorem 6.2

Given $s \varepsilon States$, it is impossible that for any $\Omega \varepsilon Syn$ $tr(\langle \Omega, s \rangle) \xrightarrow{u} x$, where x is in T_t and $rt(u)=\emptyset$.

Proof The proof is by structural induction on Ω.

case 1. Ω is skip, abort, x:=e, P!W(e) and Q?W(x).

Looking at the complete computations from $tr(\langle \Omega, s \rangle)$ given above the result is obvious.

case 2. Ω is $b \Rightarrow BC;C$.

Then $[\![b]\!]_s = tt$ and $FV(b)=\emptyset$ and

$tr(\langle b \Rightarrow BC;C,s\rangle) = m(\underline{\text{if}}\ b\ \underline{\text{then}}\llbracket BC\rrbracket\underline{\text{before}}\llbracket C\rrbracket\underline{\text{else}}\ Nil,s)$

$= m(\underline{\text{if}}\ b\ \underline{\text{then}}[\![BC]\!]\underline{\text{before}}[\![C]\!]\underline{\text{else}}\ Nil,s)$

(for as $GV(b \Rightarrow BC;C) = \emptyset$ we have $\llbracket BC\rrbracket = [\![BC]\!]$)

$= m(\underline{\text{if}}\ b\ \underline{\text{then}}[\![BC;C]\!]\underline{\text{else}}\ Nil,s)$

Thus the computation from $tr(\langle b \Rightarrow BC;C,s\rangle)$ is the same as the computation from $tr(\langle BC;C,s\rangle)$ therefore by the induction hypothesis the result is true.

case 3. Ω is GC1[]GC2.

Computation from $tr(\langle GC1 [] GC2,s\rangle)$ is the same as computation from $tr(\langle GCi,s\rangle)$ i=1 or 2. By the induction hypothesis on GCi the result is therefore true.

case 4. Ω is C1;C2.

$tr(\langle \Omega,s\rangle) = m([\![C1]\!]\underline{\text{before}}[\![C2]\!],s)$

Then according to lemma 6.5 we have only three possibilities:

a. For some t' and s' we have

$x = m(t'\underline{\text{before}}[\![C2]\!],s')$ and $tr(\langle C1,s\rangle) \xrightarrow{u} m(t',s')$

Since x is not in T_t this is not the case.

b. For some u1, u2 and s'

$tr(\langle C1,s\rangle) \xrightarrow{u1} m(\underline{\text{done}},s')$ and $tr(\langle C2,s'\rangle) \xrightarrow{u2} x$

and $u = u1 \tau u2$. By the induction hypothesis to C_1 the case is impossible.

c. For some u1 and s'

$tr(\langle C1,s\rangle) \xrightarrow{u1} m(\underline{\text{fail}},s') \xrightarrow{\tau^+} tr(\underline{\text{abortion}})$

By the induction hypothesis to C1 the case is impossible.

case 5. Ω is do GC od.

According to the translation any computation from $tr(\langle \Omega,s\rangle)$ begins as follows:

$tr(\langle\Omega,s\rangle)=m(\mu P.(RS_{GV}[\underline{\text{if}}\ Bool(GC)\ \underline{\text{then}}(\llbracket GC\rrbracket\underline{\text{before}}\ P)\underline{\text{else}}\llbracket\underline{\text{skip}}\rrbracket])\ ,s)$

$\xrightarrow{\tau^*}m(\underline{\text{if}}\ s(Bool(GC))\ \underline{\text{then}}\ \llbracket s(GC)\rrbracket\underline{\text{before}}\llbracket\Omega\rrbracket\ \underline{\text{else}}\ \llbracket\underline{\text{skip}}\rrbracket,s)$

So for some u1 with $rt(u1)=\emptyset$ we have one of the following possibilities:

a. $m(\langle\llbracket s(GC)\rrbracket\underline{\text{before}}\llbracket\Omega\rrbracket,s\rangle)\xrightarrow{u1}x$

b. $tr(\langle\underline{\text{skip}},s\rangle)\xrightarrow{u1}x$

Similar to the proof of case 4 the first is impossible and by case 1 the second is also impossible.

case 6. Ω is $\underline{\text{if}}$ GC $\underline{\text{fi}}$.

The proof is similar to case 5.

case 7. Ω is $C1\|C2$.

According to lemma 6.6 we have five possibilities:

a. For some t_1, t_2 and s' we have $x=(t1\ \underline{\text{par}}\ t2,s')$. Then x is not in T_t so this is not the case.

b. For some u1 and u2 and t2 we have

$tr(\langle C1\|C2,s\rangle)\xrightarrow{\bar{u1}}m(\underline{\text{done}}\ \underline{\text{par}}\ t2,s')$ and

$m(t2\ \underline{\text{before}}\ \underline{\text{done}},s')\xrightarrow{u2}x$ and $u=u1\tau u2$

and for some u11, u12 we have Merge(u11,u12,u1) and $s'=s(u11)\oplus s(u12)$

and $tr(\langle C2,s\rangle)\xrightarrow{\bar{u12}}m(t2,s(u12))$

and $tr(\langle C1,s\rangle)\xrightarrow{\bar{u11}}m(\underline{\text{done}},s(u11))$

Since $rt(u)=\emptyset$ so $rt(u1)=\emptyset$ thus according to the definition of Merge $rt(u11)=\emptyset$. So by the induction hypothesis the last transition is impossible, and so case (b) is impossible.

c. For some u1, u2 and t2 we have:

$tr(\langle C1\|C2,s\rangle)\xrightarrow{\bar{u1}}m(\underline{\text{fail}}\ \underline{\text{par}}\ t2,s')$ and

$m(t2\ \underline{\text{before}}\llbracket\underline{\text{abort}}\rrbracket,s')\xrightarrow{u2}x$ and $u=u1\tau u2$

and for some u11, u12, we have Merge(u11,u12,u1) and $s'=s(u11)\oplus s(u12)$ and

and $tr(\langle C2,s\rangle) \xrightarrow{\bar{u}12} m(t2,s(u12))$

and $tr(\langle C1,s\rangle) \xrightarrow{\bar{u}11} m(\underline{fail},s(u11))$

Similar to case (b) $rt(u11)=\emptyset$, so by the induction hypothesis the last transition is impossible and case (c) is impossible.

d. Exchange the positions of C1 and C2 in cases (b) and (c). Then the cases are still impossible and the proofs are similar to cases (b) and (c).

case 8. Ω is R::C.

As $rt(u)=\emptyset$ we have $\phi_R(u)=u$ (by lemma 6.10) therefore by lemma 6.9 $tr(\langle C,s\rangle) \xrightarrow{u} x$, $x\varepsilon T_2$ and $rt(u)=\emptyset$. By the induction hypothesis this is impossible so case 8 is impossible.

case 9. Ω is process R;C. The proof is similar to case 8. □

Corollary 6.1

The translation tr satisfies P0.

Proof. If $r\varepsilon T_c$ it is either s or abortion. Then tr(r) is either m(done,s) or $(M_{s0}\,|\,\underline{fail})\backslash M$ and so in T_t in either case.

If $tr(r) \xrightarrow{u} x$ with x in T_t and $rt(u)=\emptyset$ then by the previous theorem r cannot be of the form $\langle\Omega,s\rangle$ so $r\varepsilon T_c$. □

To prove tr satisfies P1 we need the following lemma which is easily proved by structural induction.

Lemma 6.12

If $\langle\Omega,s\rangle \xrightarrow{\varepsilon}$ abortion then

$$tr(\langle\Omega,s\rangle) \xrightarrow{u} m(\underline{fail},s) \xrightarrow{\tau^+} tr(\underline{abortion}).$$

and $rt(u)=\varepsilon$.

Theorem 6.3

The translation tr satisfies P1. That is for any $r=\langle\Omega,s\rangle$, $\Omega\in Syn$ and $\lambda\in\Lambda_p$, $\langle\Omega,s\rangle\xrightarrow[p]{\lambda}r$ implies that for some $u\in\Lambda_t^*$ $tr(\langle\Omega,s\rangle)\xrightarrow[t]{u}tr(r)$ and $rt(u)=\lambda$.

Proof The proof is by structural induction on Ω. We need to examine the following cases:

Case 1. Ω is skip, abort and x:=e. By looking at their complete computations the result is obvious.

Case 2. Ω is $P?W(x_i)$.

Then r must be $\langle skip,s\rangle$ and λ must be $(*,P)?W(v)$. So

$$tr(\langle P?W(x_i),s\rangle)=m([*,P]Wx.\alpha_i(x,[\![skip]\!]),s)$$

$$\xrightarrow[t]{[*,P]W?v}m(\alpha_i(v,[\![skip]\!]),s)$$

$$\xrightarrow[t]{\tau}m([\![skip]\!],s[v/x_i])$$

The last term is just $tr(\langle skip,s[v/x_i]\rangle)$.

Case 3. Ω is $P!W(e)$.

The proof is similar to case 2.

Case 4. Ω is $b\Rightarrow BC;C$.

According to the guard rule in section 2.3 $[\![b]\!]_s=tt$ and $\langle BC;C,s\rangle\xrightarrow[p]{\lambda}r$. By the induction hypothesis we have

$$tr(\langle BC;C,s\rangle)\xrightarrow[t]{u}tr(r) \text{ and } rt(u)=\lambda$$

So since $[\![b]\!]_s=tt$ and $FV(b)=\emptyset$ and

$$tr(\langle b\Rightarrow BC;C,s\rangle)=m(\underline{if}\ b\ \underline{then}(\!|BC|\!)\underline{before}[\![C]\!]\underline{else}\ Nil,s)$$

$$=m(\underline{if}\ b\ \underline{then}[\![BC]\!]\underline{before}[\![C]\!]\underline{else}\ Nil,s)$$

(for as $GV(b\Rightarrow BC;C)=\emptyset$ we have $(\!|BC|\!)=[\![BC]\!]$)

$$=m(\underline{if}\ b\ \underline{then}[\![BC;C]\!]\underline{else}\ Nil,s)$$

Thus the computation from $tr(\langle b \Rightarrow BC; C, s\rangle)$ is the same as the computation from $tr(\langle BC; C, s\rangle)$ therefore

$$tr(\langle b \Rightarrow BC; C, s\rangle) \xrightarrow[t]{u} tr(r).$$

Case 5. Ω is $GC_1 [] GC_2$.

According to the alternative rule in section 2.3 we know that

$$\langle GC_i, s\rangle \xrightarrow[p]{\lambda} r \quad \text{where} \quad i=1 \text{ or } 2$$

By the induction hypothesis we have

$$tr(\langle GC_i, s\rangle) \xrightarrow[p]{u} tr(r) \text{ and } rt(u)=\lambda.$$

Thus by lemma 6.8

$$tr(\langle GC_1 [] GC_2, s\rangle) = m([\![GC_1]\!] + [\![GC_2]\!], s) \xrightarrow[t]{u} tr(r)$$

Case 6. Ω is do GC od.

According to the repetition rule in section 2.3 we need to examine the following subcases:

1. $[\![Bool(GC)]\!]_s = tt$ and $\langle GC, s\rangle \xrightarrow[p]{\lambda} \langle C, s'\rangle | s' |$ abortion.

Let us just check the first one:

$$\langle GC, s\rangle \xrightarrow[p]{\lambda} \langle C, s'\rangle \text{ and so } \langle s(GC), s\rangle \xrightarrow[p]{\lambda} \langle C, s'\rangle.$$

By the induction hypothesis

$tr(\langle s(GC), s\rangle) \xrightarrow[t]{u} m([\![C]\!], s')$ and $rt(u)=\lambda$ therefore

$tr(\langle$ do GC od$, s\rangle)$

$= m(\mu P.(RS_{GV(GC)}[$ if Bool(GC) then$([\![GC]\!]$ before P) else $[\![$skip$]\!]), s)$

$\xrightarrow[t]{r^*} m($ if $s($Bool(GC)$)$ then$([\![s(GC)]\!]$ before $[\![\Omega]\!])$ else $[\![$skip$]\!], s)$

(by the recursive rule and lemma 6.4)

$\xrightarrow[t]{u} m([\![C]\!]$ before $[\![\Omega]\!], s')$

(by lemma 6.8)

The last form is just tr(⟨C;do GC od, s'⟩).

2. $[\![\text{Bool(GC)}]\!]_s$=ff, therefore

$$tr(\langle \Omega, s\rangle) \xrightarrow[t]{\tau^*} (\underline{\text{if}}\ s(\text{Bool(GC)})\ \underline{\text{then}}([\![s(\text{GC})]\!]\ \underline{\text{before}}\ [\![\Omega]\!])\ \underline{\text{else}}\ [\![\underline{\text{skip}}]\!], s)$$

$$\xrightarrow[t]{\tau'} m(\underline{\text{done}}, s)$$

The last form is just tr(s).

Case 7. Ω is if GC fi.

The proof is similar to case 6.

Case 8. Ω is $C_1;C_2$.

According to the composition rule in section 2.3 there are three subcases to study:

a. $\langle C_1, s\rangle \xrightarrow[p]{\lambda} \langle C_1', s'\rangle$. Then by the induction hypothesis we have

$tr(\langle C_1, s\rangle) \xrightarrow[t]{u} m([\![C_1']\!], s')$ and $rt(u)=\lambda$. Therefore applying lemma 6.5

$$tr(\langle \Omega, s\rangle) = m([\![C_1]\!]\ \underline{\text{before}}\ [\![C_2]\!], s)$$

$$\xrightarrow[t]{u} m([\![C_1']\!]\ \underline{\text{before}}\ [\![C_2]\!], s')$$

The last is just the form which we expect.

b. $\langle C_1, s\rangle \xrightarrow[p]{\varepsilon} s'$. The proof is similar to subcase (a).

c. $\langle C_1, s\rangle \xrightarrow[p]{\varepsilon} \underline{\text{abortion}}$. Then $\langle C_1;C_2, s\rangle \xrightarrow[p]{\varepsilon} \underline{\text{abortion}}$. By lemma 6.12 we have:

$$tr(\langle C_1, s\rangle) \xrightarrow[t]{u} m(\underline{\text{fail}}, s) \xrightarrow{\tau^+} tr(\underline{\text{abortion}})$$

and $rt(u)=\varepsilon$, therefore by lemma 6.5

$$tr(\langle \Omega, s\rangle) \xrightarrow[t]{u} m(\underline{\text{fail}}\ \underline{\text{before}}\ [\![C_2]\!]), s)$$

$$\xrightarrow[t]{\tau} m(\underline{fail}, s) \xrightarrow{\tau^+} tr(\underline{abortion}).$$

Case 9. Ω is $C_1 \| C_2$.

According to the parallel rule in section 2.3 we need to study the following subcases:

a. $\langle C_1, s\rangle \xrightarrow[p]{\lambda} \langle C_1', s'\rangle$ and $\langle \Omega, s\rangle \xrightarrow[p]{\lambda} \langle C_1' \| C_2, s'\rangle$. By the induction hypothesis we have

$$tr(\langle C_1, s\rangle) \xrightarrow[t]{u} m(\llbracket C_1' \rrbracket, s') \ , \ rt(u)=\lambda$$

By lemma 6.4 (a) we have u' such that $\bar{u}'=u$ and $s'=su'$. Noticing that $Merge(u', \emptyset, u')$ and $s'=su' \oplus s\emptyset=su'$ applying lemma 6.6 we have:

$$tr(\langle \Omega, s\rangle) = m(\llbracket C_1 \rrbracket \underline{par} \llbracket C_2 \rrbracket, s) \xrightarrow[t]{u} m(\llbracket C_1' \rrbracket \underline{par} \llbracket C_2 \rrbracket, s')$$

b. $\langle C_1, s\rangle \xrightarrow[p]{\lambda} \langle C_1', s'\rangle$, $\langle C_2, s\rangle \xrightarrow[p]{\bar{\lambda}} \langle C_2', s\rangle$ and then $\langle \Omega, s\rangle \xrightarrow[p]{\varepsilon} \langle C_1' \| C_2', s'\rangle$. By the induction hypothesis we have:

$$tr(\langle C_1, s\rangle) \xrightarrow[t]{u11.tr(\lambda).u12} m(\llbracket C_1' \rrbracket, s')$$

$$tr(\langle C_2, s\rangle) \xrightarrow[t]{u21.tr(\bar{\lambda}).u22} m(\llbracket C_2' \rrbracket, s)$$

where u11, u12, u21, u22 are sequences which contain τ only. By lemma 6.4 (a) we have v11, v12, v21 and v22 such that $\bar{v}11=u11$, $\bar{v}12=u12$, $\bar{v}21=u21$ and $\bar{v}22=u22$. Let

$$v1=v11.tr(\lambda).v12 \quad \text{and} \quad v2=v21.tr(\bar{\lambda}).v22.$$

we have $s'=s(v1)$ and $s=s(v2)$. Take $v3=v11.v12.\tau'.v12.v22$ then $Merge(v1, v2, v3)$ and $s(v3)=s(v1) \oplus s(v2)=s'$ (Noticing $FL(\llbracket C_1 \rrbracket) \cap FL(\llbracket C_2 \rrbracket) \cap M_a=\emptyset$). So applying lemma 6.6 we have:

$$tr(\langle \Omega, s\rangle) = m(\llbracket C_1 \rrbracket \underline{par} \llbracket C_2 \rrbracket, s) \xrightarrow[t]{\bar{v}3} m(\llbracket C_1' \rrbracket \underline{par} \llbracket C_2' \rrbracket, s')$$

c. $\langle C_1, s\rangle \xrightarrow[p]{\varepsilon} s$ then $\langle \Omega, s\rangle \xrightarrow[p]{\varepsilon} \langle C_2; \underline{skip}, s\rangle$. By the induction

hypothesis we have

$$tr(\langle C_1,s\rangle) \xrightarrow{u}_t m(\underline{done},s) \quad \text{and} \quad rt(u)=\varepsilon.$$

By lemma 6.4 (a) we have u' such that $\bar{u}'=u$ and $s=su'$. Noticing that Merge(u',$\emptyset$,u') and $s=su'\Theta s\emptyset=su'$, applying lemma 6.6 we have

$$tr(\langle \Omega,s\rangle) \xrightarrow{u}_t m(\underline{done}\ \underline{par}[\![C_2]\!],s)$$
$$\xrightarrow{\tau}_t m([\![C_2]\!]\underline{before}[\![\underline{skip}]\!],s)$$

d. $\langle C_1,s\rangle \xrightarrow{\varepsilon}_p \underline{abortion}$ then $\langle \Omega,s\rangle \xrightarrow{\varepsilon}_p \langle C_2;\underline{abort},s\rangle$. By the lemma 6.12 we have:

$$m([\![C_1]\!],s) \xrightarrow{u'} m(\underline{fail},s) \xrightarrow{u''} tr(\underline{abortion})$$

and rt(u')=ε. So applying lemma 6.6 we have:

$$tr(\langle \Omega,s\rangle)=([\![C_1]\!]\underline{par}[\![C_2]\!],s)$$
$$\xrightarrow{u}_t(\underline{fail}\ \underline{par}[\![C_2]\!],s)$$
$$\xrightarrow{\tau}_t([\![C_2]\!]\underline{before}\ [\![\underline{abort}]\!],s)$$

e. By exchanging the positions of C_1 and C_2 in subcases (a), (b), (c) and (d). the proofs of these cases are similar to the proofs in (a), (b), (c) and (d).

Case 10. Ω is R::C.

Then if $\langle R::C,s\rangle \xrightarrow{\lambda} r$ there are three subcases where for some λ' with $\lambda=\phi_R(\lambda')$ we have:

$$\langle C,s\rangle \xrightarrow{\lambda'} \langle C',s'\rangle | s' | \underline{abortion}$$

and r is $\langle R::C',s'\rangle$ or s' or abortion. We just consider the first one. By the induction hypothesis we have

$$tr(\langle C,s\rangle) \xrightarrow{u} tr(\langle C',s'\rangle) \text{ and}$$

rt(u)=λ'. So u=ultr(λ)u2 where u1 and u2 are composed of τ's only. Therefore as $\phi_R(\lambda')$ is defined so is $\phi_R(u)$, by lemma 6.9 we have

$$tr(\langle R::C,s\rangle) \xrightarrow{\phi_R(u)} tr(\langle R::C',s'\rangle)$$

thus we calculate that

$$\begin{aligned} rt(\phi_R(u)) &= rt(u1\phi_R(tr(\lambda'))u2) \\ &= rt(\phi_R(tr(\lambda'))) \\ &= \phi_R(rt(tr(\lambda'))) \quad \text{(by lemma 6.10)} \\ &= \phi_R(\lambda') \\ &= \lambda \end{aligned}$$

Case 10. Ω is process R;C.

Then if $\langle$process R;C,s$\rangle \xrightarrow{\lambda} r$ there are three subcases where for some λ' with $\lambda=\eta_{R,FPL(C)}(\lambda')$ we have $\langle C,s\rangle \xrightarrow{\lambda'} \langle C',s'\rangle | s' |$ abortion and r is $\langle$process R;C',s'$\rangle$ or s' or abortion. Let us just consider the first subcase.

Now by the induction hypothesis we have

$$tr(\langle C,s\rangle) \xrightarrow{u} tr(\langle C',s'\rangle) \quad \text{where } u=u1tr(\lambda)u2$$

and $u1, u2 \in \{\tau\}^*$. As $rt(tr(\lambda'))=\lambda'$ and $\eta_{R,L}(\lambda')$ is defined $\eta_{R,L}(rt(tr(\lambda')))$ is defined; so we can apply lemma 6.11 to see that $\eta_{R,L}(tr(\lambda'))$ is defined and also

$$rt(\eta_{R,L}(tr(\lambda')))=\eta_{R,L}(\lambda')$$

so by lemma 6.9 we have

$$tr(\langle \text{process } R;C,s\rangle) \xrightarrow{u1\eta_{R,L}(tr(\lambda'))u2} tr(\langle \text{process } R;C',s'\rangle)$$

and $rt(u1\eta_{R,L}(\lambda'))u2) = \eta_{R,L}(\lambda')$ as required.

Thus the lemma has been proved. □

Lemma 6.13

If $tr(\langle GC,s\rangle) \xrightarrow{u} x$ then $hd(u) \in tr(\Lambda_p)$.

Proof. The proof is by structural induction on GC.

case 1. GC is b⇒BC;C. According to the definition of the

translation the first transition action of tr(⟨GC,s⟩) must be the same as the first transition action of tr(⟨BC,s⟩). Since BC can only be skip, P!W(e) or Q?W(x) the result is obvious.

case 2. GC is GC1▯GC2. Since computation from tr(⟨GC,s⟩) is the same as tr(⟨GCi,s⟩) for i=1 or 2, by induction on GCi the result is proved. □

We now choose M=$\{\tau\}^*$ and prove the translation satisfies P2.

Theorem 6.4

The translation tr satisfies P2. That is, if $tr(\langle\Omega,s\rangle) \xrightarrow[t]{u} x$, $rt(u)=\lambda$ then there exist $r \in \Gamma_p$, $x' \in \Gamma_t$, and u_1, $u_2 \in \{\tau\}^*$ such that $\langle\Omega,s\rangle \xrightarrow{\lambda} r$ and $tr(r) \xrightarrow[t]{u_1} x'$ and $x \xrightarrow[t]{u_2} x'$ (see figure 5.8).

Proof The proof is by structural induction. Consider the following cases:

Case 1. Ω is one of skip, abort, x:=e, P?W(x) and P!W(e).

Since the computation from tr(⟨Ω,s⟩) is deterministic the result is obvious.

Case 2. Ω is b⇒BC;C.

Then $[\![b]\!]_s$=tt and the computation from tr(⟨Ω,s⟩) is the same as the computation from tr(⟨BC;C,s⟩). By the induction hypothesis the computation from tr(⟨BC;C,s⟩) satisfies the lemma, thus so does the computation from tr(⟨Ω,s⟩).

Case 3. Ω is $GC_1 + GC_2$.

Here we need only notice that the computation from tr(⟨Ω,s⟩) is the same as one of the $tr(\langle GC_i,s\rangle)$.

Case 4. Ω is $C_1;C_2$.

According to lemma 6.5 three subcases can arise:

case 4.1 For some t_1, s' we have:

$x=m(t_1 \underline{\text{before}} [\![C_2]\!],s')$ and $tr(\langle C_1,s\rangle) \xrightarrow{u} m(t_1,s')$

Now by the induction hypothesis we can find r_1, x_1' and $u_1,u_2 \in \{\tau\}^*$

such that

$$\langle C_1, s\rangle \xrightarrow{\lambda} r_1 \text{ and } tr(r_1) \xrightarrow{u_1} x_1' \text{ and } m(t_1, s') \xrightarrow{u_2} x_1'$$

Now there are still three cases according to the form of r_1:

case 4.1.1 $r_1=\langle C_1', s''\rangle$. Then we take $r=\langle C_1'; C_2, s''\rangle$ and certainly $\langle C_1; C_2, s\rangle \xrightarrow{\lambda} r$. Next since $tr(\langle C_1', s''\rangle) \xrightarrow{u_1} x_1'$ and $rt(u_1)=\emptyset$, by theorem 6.2 we have $x_1'=m(t_1', s_1'')$ for some t_1' and s_1''. So we take $x'=m(t_1' \underline{\text{before}} [\![C_2]\!], s_1'')$ and by lemma 6.5 have:

$$tr(r)=tr(\langle C_1'; C_2, s''\rangle) \xrightarrow{u_1} m(t_1' \underline{\text{before}} [\![C_2]\!], s_1'')$$

and since $m(t_1, s') \xrightarrow{u_2} x_1'=m(t_1', s_1'')$ by lemma 6.5 again

$$m(t_1 \underline{\text{before}} [\![C_2]\!], s') \xrightarrow{u_2} m(t_1' \underline{\text{before}} [\![C_2]\!], s'')$$

and so the case is proved.

case 4.1.2 $r_1=s''$. Then $tr(r_1)=tr(s'')$ and so $u_1=\emptyset$ and $x_1'=tr(s'')$. Take $r'=\langle C_2, s''\rangle$ and $x'=tr(\langle C_2, s''\rangle)$. Now by the composition rule we have

$$\langle C_1; C_2, s\rangle \xrightarrow{\lambda} \langle C_2, s'\rangle$$

and by lemma 6.5 we have

$$m(t_1 \underline{\text{before}} [\![C_2]\!], s') \xrightarrow{u_2} m(\underline{\text{done}} \underline{\text{before}} [\![C_2]\!], s'') \xrightarrow{\tau} m(\langle [\![C_2]\!], s''\rangle)$$

and the last is just x' so the case is proved.

case 4.1.3 $r_1=\underline{\text{abortion}}$. Then $tr(r_1)=tr(\underline{\text{abortion}})$ so $u_1=\emptyset$ and $x_1'=tr(\underline{\text{abortion}})$ and $u_2 \varepsilon \{\tau\}^*$ and $m(t_1, s') \xrightarrow{u_2} tr(\underline{\text{abortion}})$. So by lemma 6:12 we have $s'=s$ and for some $u21 \varepsilon \{\tau\}^*$ and $u_2=u21.\tau^+$ and

$$m(t_1, s) \xrightarrow{u21} m(\underline{\text{fail}}, s) \xrightarrow{\tau^+} tr(\underline{\text{abortion}})$$

Thus take $r=\underline{\text{abortion}}$, $x'=tr(\underline{\text{abortion}})$; by lemma 6.5 (c) we have:

$$m(t_1 \underline{\text{before}} [\![C_2]\!], s) \xrightarrow{u_2'} tr(\underline{\text{abortion}})$$

where $u_2'=u_{21}.\tau.\tau^+$ and so the case is proved.

case 4.2 For some w_1, w_2 and s' we have $u=w_1 \tau w_2$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{w_1} m(\underline{done}, s') \text{ and } tr(\langle C_2, s'\rangle) \xrightarrow{w_2} x$$

Since $rt(u)=\lambda$ we see that $rt(w_1)=\emptyset$ or $rt(w_2)=\emptyset$. By theorem 6.2 we cannot have $rt(w_1)=\emptyset$ so $rt(w_2)=\emptyset$.

By the induction hypothesis to C_1 we have r_1, u_1, u_2 and x_1' such that

$$\langle C_1, s\rangle \xrightarrow{\lambda} r_1 \quad \text{and} \quad tr(r_1) \xrightarrow{u_1} x_1' \quad \text{and} \quad m(\underline{done}, s') \xrightarrow{u_2} x_1'$$

Here $\lambda=rt(w_1)$, $u_2=\emptyset$ and $x_1'=m(\underline{done}, s')$. Since $rt(u_1)=\emptyset$ by theorem 6.2 we have $r_1=s'$ and $u_1=\emptyset$.

So we take $r=\langle C_2, s'\rangle$, $x'=x$, $u_2=\emptyset$ and $u_1=w_2$ and have

$$\langle C_1; C_2, s\rangle \xrightarrow{\lambda} \langle C_2, s'\rangle \quad \text{and} \quad tr(\langle C_2, s\rangle) \xrightarrow{w_2} x.$$

case 4.3 For some w_1, w_2 and s' we have $u=w_1 w_2$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{w_1} m(\underline{fail}, s') \xrightarrow{w_2} x$$

Here $rt(w_1)=rt(u)=\lambda$. By the induction hypothesis we have r_1, u_1, u_2 and x_1' such that

$$\langle C_1, s\rangle \xrightarrow{\lambda} r_1 \quad \text{and} \quad tr(r_1) \xrightarrow{u_1} x_1' \quad \text{and} \quad x \xrightarrow{u_2} x_1'.$$

Noticing that the computation from $m(\underline{fail}, s')$ is *deterministic* so we choose $x_1'=tr(\underline{abortion})$ and then by theorem 6.2 we see that $u_1=\emptyset$ and so $r_1=\underline{abortion}$. So take $r=\underline{abortion}$ and $x'=x_1'$; by the composition rule we have $\langle C_1; C_2, s\rangle \xrightarrow{\lambda} \underline{abortion}$ and we already have $x \xrightarrow{u_2} x_1'=x'$ and the case is proved.

Case 5. Ω is *do* GC *od*.

There are two subcases to analyse:

a. $[\![Bool(GC)]\!]_s=ff$. Then:

$$tr(\langle \Omega, s\rangle) \xrightarrow[t]{\tau^*} m(\underline{if}\ s(Bool(GC))\ \underline{then}([\![s(GC)]\!]\underline{before}[\![\Omega]\!])\underline{else}[\![\underline{skip}]\!], s)$$

$\xrightarrow[t]{\tau'}m(\underline{done},s)$

choose $r=s$, $x'=m(\underline{done},s)$ and $u_1=u_2=\emptyset$.

b. $[\![Bool(GC)]\!]_s=tt$. Notice that the initial transition sequence of the computation from $tr(\langle\Omega,s\rangle)$ is a deterministic τ sequence followed by a transition sequence of $m([\![s(GC)]\!]\underline{before}[\![\Omega]\!],s)$ Then the proof can be done by an analysis like that of case 4.

Case 6. Ω is $\underline{if}$ GC $\underline{fi}$.

The proof is similar to case 5.

Case 7. Ω is $C_1 \| C_2$.

This is the most important and complicated case. Suppose

$$tr(\langle C_1 \| C_2, s\rangle)=m([\![C_1]\!]\underline{par}[\![C_2]\!],s) \xrightarrow{u} x \quad \text{and} \quad rt(u)=\lambda.$$

Then according to lemma 6.6 five possible subcases arise:

case 7.1 For some u', v_1, v_2, s' we have $\bar{u}'=u$ and

$Merge(v_1,v_2,u')$ and $x=m(t_1\underline{par}\ t_2,s')$ and

$$m([\![C_1]\!],s) \xrightarrow{\bar{v}_1} m(t_1,s(v_1)) \quad \text{and} \quad m([\![C_2]\!],s) \xrightarrow{\bar{v}_2} m(t_2,s(v_2))$$

where $s'=sv_1\theta sv_2$. Noticing that $rt(u)=\lambda$ and so $rt(u')=\lambda$. According to the definition of merge, there are two subcases to study:

case 7.1.1 $rt(v_i)=\emptyset$ for some i (i=1 or 2). Suppose i=2, thus $rt(v_1)=\lambda$. By the induction hypothesis there exist r_1, and u_{11}, $u_{12}\varepsilon\{\tau\}^*$ such that:

$$\langle C_1,s\rangle \xrightarrow{\lambda} r_1 \quad \text{and} \quad tr(r_1) \xrightarrow{u_{11}} x_1' \quad \text{and} \quad m(t_1,s(v_1)) \xrightarrow{u_{12}} x_1'$$

Now there are still three subcases depending on the form of r_1:

a1. If $r_1=\langle C_1', s_1\rangle$ then by theorem 6.2 $x_1'=m(t_1', s_1')$ for some t_1' and s_1' so

$$\begin{array}{ccc} tr(\langle C_1, s\rangle) & \xrightarrow{\bar{v}_1} & m(t_1, sv_1) \\ & & \downarrow u_{12} \\ tr(\langle C_1', s_1\rangle) & \xrightarrow{u_{11}} & m(t_1', s_1') \end{array} \qquad \begin{array}{c} \langle C_1, s\rangle \\ \downarrow \lambda \\ \langle C_1', s_1\rangle \end{array}$$

where $s_1'=s_1u_{11}'=sv_1u_{12}'$ and $u_{11}=\bar{u}_{11}'$, $u_{12}=\bar{u}_{12}'$ (by lemma 6.4). So take $r'=\langle C_1' \| C_2, s_1\rangle$ and $x'=m(t_1' \underline{par}\ t_2, s3)$, where $s3=s_1'\Theta sv_2$ and $u_1=u_{11}\bar{v}_2$ and $u_2=u_{12}$. Notice $Merge(u_{11}', v_2, u_{11}'v_2)$ and $Merge(u_{12}', \emptyset, u_{12}')$ and $s3=s_1(u_{11}')\Theta s(v_2)=s_1(u_{11}'v_2)$ and $s3=(sv_1)u_{12}'\Theta sv_2=(sv_1v_2)u_{12}'=s'u_{12}'$ $(FL([\![C_1]\!]\cap[\![C_2]\!]\cap M_a=\emptyset)$ so by lemma 6.6 we have

$$\begin{array}{ccc} m([\![C_1]\!] \underline{par} [\![C_2]\!], s) & \xrightarrow{u} & m(t_1 \ \underline{par}\ t_2, s') \\ & & \downarrow u_{12} \\ m([\![C_1']\!] \underline{par} [\![C_2]\!], s_1) & \xrightarrow{u_{11}\bar{v}_2} & m(t_1' \ \underline{par}\ t_2, s3) \end{array}$$

and by the parallel rule we have $\langle C_1 \| C_2, s\rangle \xrightarrow[p]{\lambda} \langle C_1' \| C_2, s_1\rangle$.

a2. If $r_1=s$ then choose $r'=\langle C_2; \underline{skip}, s\rangle$, $x'=m(t_2 \underline{before} [\![\underline{skip}]\!], s)$ and $u_1=\bar{v}_2$ and $u_2=u_{12}\tau$.

a3. If $r_1=\underline{abortion}$ then choose $r'=\langle C_2; \underline{abort}, s\rangle$, $x'=m(t_2 \underline{before} [\![\underline{abort}]\!], s)$, $u_1=\bar{v}_2$ and $u_2=u_{12}\tau$.

case 7.1.2 $rt(u)=\varepsilon$ and $rt(v_1)=\rho$, $rt(v_2)=\bar{\rho}$ (Here $\bar{\rho}$ is the complement of ρ). By the induction hypothesis there exist $\langle C_1', s_1\rangle$, t_1', and u_{11}, $u_{21}\varepsilon\{\tau\}^*$ such that

$$\begin{array}{ccc} m([\![C_1]\!], s) & \xrightarrow{\bar{v}_1} & m(t_1, sv_1) \\ & & \downarrow u_{21} \\ m([\![C_1']\!], s_1) & \xrightarrow{u_{11}} & m(t_1', s_1') \end{array} \qquad \begin{array}{c} \langle C_1, s\rangle \\ \downarrow \lambda \\ \langle C_1', s_1\rangle \end{array}$$

where $s_1' = s_1 u_{11}' = s(v_1 u_{21}')$ and $\bar{u}_{11}' = u_{11}$ and $\bar{u}_{21}' = u_{21}$ as usual.

and $\langle C_2', s\rangle$, t_2', u_{12}, $u_{22} \in \{\tau\}^*$ such that

$$\begin{array}{ccc} m(\llbracket C_2 \rrbracket, s) & \xrightarrow{\bar{v}_2} & m(t_2, sv_2) \\ & & \downarrow u_{22} \\ m(\llbracket C_2' \rrbracket, s) & \xrightarrow{u_{12}} & m(t_2', s_2') \end{array} \qquad \begin{array}{c} \langle C_2, s\rangle \\ \downarrow \lambda \\ \langle C_2', s\rangle \end{array}$$

where $s_2' = su_{12}' = s(v_2 u_{22}')$ and $\bar{u}_{12}' = u_{12}$ and $\bar{u}_{22}' = u_{22}$.

Let $s3 = s_1' \oplus s_2'$. Take $r = \langle C_1' \| C_2', s_1\rangle$, $x' = m(t_1' \underline{\text{par}} t_2', s3)$, $u_1 = u_{11} \cdot u_{12}$ and $u_2 = u_{21} \cdot u_{22}$. Applying lemma 6.6 (a) we have

$$\begin{array}{ccc} m(\llbracket C_1 \rrbracket \underline{\text{par}} \llbracket C_2 \rrbracket, s) & \xrightarrow{u} & m(t_1 \underline{\text{par}} t_2, s') \\ & & \downarrow u_2 \\ m(\llbracket C_1' \rrbracket \underline{\text{par}} \llbracket C_2' \rrbracket, s_1) & \xrightarrow{u_1} & m(t_1' \underline{\text{par}} t_2', s3). \end{array}$$

where the horizontal transition is because

$\text{Merge}(u_{11}', u_{12}', u_{11}'u_{12}')$ and $s3 = s_1' \oplus s_2' = s_1 u_{11}' \oplus s u_{12}' = s_1(u_{11}'u_{12}')$

the last equality is by $FL(\llbracket C_1 \rrbracket) \cap FL(\llbracket C_2 \rrbracket) \cap M_\alpha = \emptyset$. The vertical transition is because

$\text{Merge}(u_{21}', u_{22}', u_{21}'u_{22}')$ and
$s3 = s_1' \oplus s_2' = s(v_1 u_{21}') \oplus s(v_2 u_{22}') = (sv_1v_2)(u_{21}'u_{22}') = s'(u_{21}'u_{22}')$.

By the parallel rule we also have $\langle C_1 \| C_2, s\rangle \xrightarrow[p]{\varepsilon} \langle C_1' \| C_2', s'\rangle$ and so the case is proved.

case 7.2 For some w_1, w_2, s' and t_2 such that $u = \bar{w}_1 \tau w_2$

and $tr(\langle C_1 \| C_2, s\rangle) \xrightarrow{\bar{w}_1} m(\underline{done}\ \underline{par}\ t_2, s')$

and $m(t_2\ \underline{before}\tau\underline{done}, s') \xrightarrow{w_2} x$

and for some v_1, v_2, we have $Merge(v_1, v_2, w_1)$

and $tr(\langle C_1, s\rangle) \xrightarrow{\bar{v}_1} m(\underline{done}, sv_1)$ and $tr(\langle C_2, s\rangle) \xrightarrow{\bar{v}_2} m(t_2, sv_2)$

where $s' = sv_1 \theta sv_2$ as usual. Since $rt(u) = \lambda$ we have $rt(w_1) = \emptyset$ or $rt(w_2) = \emptyset$. By theorem 6.2 we cannot have $rt(w_1) = \emptyset$ so $rt(w_2) = \emptyset$. Similarly $rt(v_1) = \lambda \neq \emptyset$

By the induction hypothesis on C_1 we have r_1 and u_{11}, u_{21} in $\{\tau\}^*$ and x_1' such that

$$\langle C_1, s\rangle \xrightarrow{\lambda} r_1 \quad \text{and} \quad tr(r_1) \xrightarrow{u_{11}} x_1' \quad \text{and} \quad m(\underline{done}, s') \xrightarrow{u_{21}} x_1'$$

Here $u_{21} = \emptyset$ and $x_1' = m(\underline{done}, s')$. Hence as $u_{11} \in \{\tau\}^*$ by theorem 6.2 again we have $r_1 = s'$ and so by lemma 2.1 $\lambda = \varepsilon$ and $u_{11} = \emptyset$.

So take $r = \langle C_2; \underline{skip}, s'\rangle$, $x' = x$, $u_1 = w_2$ and $u_2 = \emptyset$ and we have:

$$tr(r) = tr(\langle C_2; \underline{skip}, s'\rangle) \xrightarrow{w_2} x \quad \text{and} \quad \langle C_1 \| C_2, s\rangle \xrightarrow{s} \langle C_2; \underline{skip}, s\rangle.$$

case 7.3 For some w_1, w_2, t_2 and s' we have $u = \bar{w}_1 \tau w_2$

and $tr(\langle C_1 \| C_2, s\rangle) \xrightarrow{\bar{w}_1} m(\underline{fail}\ \underline{par}\ t_2, s')$

and $m(t_2 \underline{before} [\![\underline{abort}]\!], s') \xrightarrow{w_2} x$

and for some v_1, v_2 we have $Merge(v_1, v_2, w_1)$ and $s' = sv_1 \theta sv_2$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{\bar{v}_1} m(\underline{fail}, sv_1) \quad \text{and} \quad tr(\langle C_2, s\rangle) \xrightarrow{\bar{v}_2} m(t_2, sv_2)$$

Notice the computation from $m(\underline{fail}, s')$ is deterministic and its transition sequence contains τ's only. By theorem 6.2 $rt(v_1) = \emptyset$ is impossible, therefore $rt(w_1) \neq \emptyset$ and $rt(w_2) = \emptyset$. By the induction hypothesis on C_1 we have r_1, x_1', u_{11} and u_{21}. Again the computation

from m(fail,s) is deterministic so r_1=abortion and x_1'=tr(abortion). Then by lemma 2.1 $\lambda=\varepsilon$. We take $r=\langle c_2;\underline{abort},s\rangle$, $x'=x$, $u_1=w_2$, $u_2=\emptyset$ and have:

$$tr(\langle C_2;\underline{abort},s'\rangle)\xrightarrow{w_2}x \quad \text{and} \quad \langle C_1 \| C_2,s\rangle \xrightarrow{\varepsilon} \langle C_2;\underline{abort},s\rangle$$

case 7.4 Exchange the positions of C_1 and C_2 in case 7.2. The proof is then similar to that of case 7.2.

case 7.5 Exchange the positions of C_1 and C_2 in case 7.3. The proof is then similar to that of case 7.3.

Thus the case 7 is proved.

Case 8. Ω is R::C.

According to lemma 6.9 there are two subcases:

case 8.1 $x=m(t[\phi_R],s')$ and $m([\![C]\!],s)\xrightarrow{u'}m(t,s)$ and $\phi_R(u')=u$. By the induction hypothesis there exist r',t' and u_1', $u_2'\varepsilon\{\tau\}^*$ such that

$$\begin{array}{ccc} m([\![C]\!],s) \xrightarrow{u'} m(t,s') & & \langle C,s\rangle \\ \qquad\qquad\quad \downarrow u_2' & & \quad \downarrow rt(u') \\ tr(r') \xrightarrow{u_1'} m(t',s_1'') & & r' \end{array}$$

If $r'=\langle C',s''\rangle$ then choose $r=\langle R::C',s''\rangle$, $x'=m(t'[\phi_R],s_1'')$ and $u_1=u_1'$, and $u_2=u_2'$. According to the definition of ϕ_R rt(u') can only be one of ε, $(*,P)?W(v)$ and $(*,P)!W(v)$ $(P\neq R)$, so $\phi_R(rt(u'))$ is defined and then by lemma 6.10 we have

$\phi_R(rt(u'))=rt(\phi_R(u'))=rt(u)$ and so $\langle R::C,s\rangle \xrightarrow{rt(u)} \langle R::C',s'\rangle$.
By lemma 6.9 we have $m(t[\phi_R],s')\xrightarrow{u_1}x'$ and $x\xrightarrow{u_2}x'$.

If $r'=s''$ then choose $r=s''$, $x'=m(\underline{done},s'')$ and $u_1=\emptyset$, $u_2=u_2'$. (Noticing $u_1'=\emptyset$).

If $r'=\underline{abortion}$ then choose $r=\underline{abortion}$, $x'=tr(\underline{abortion})$ and $u_1=\emptyset$ and $u_2=u_2'$.

case 8.2 For some w_1, w_2 $tr(\langle C,s\rangle) \xrightarrow{\bar{w}_1} m(\underline{fail},s') \xrightarrow{w_2} x$ where $u=\phi_R(\bar{w}_1)w_2$. So by the induction hypothesis we have r_1 and u_1', u_2' in $\{\tau\}^*$ and x_1' such that

$$\langle C,s\rangle \xrightarrow{rt(u')} r_1 \quad \text{and} \quad tr(r_1) \xrightarrow{u_1'} x_1' \text{ and } x \xrightarrow{u_2'} x_1'$$

As usual we can take $x_1'=tr(\underline{abortion})$ then by theorem 6.2 $u_1'=\emptyset$ and so $r_1=\underline{abortion}$. Now we calculate that

$$\phi_R(rt(w_1w_2))=rt(\phi_R(w_1w_2))=rt(\phi_R(w_1)w_2)=rt(u).$$

Therefore we take $r=\underline{abortion}$, $u_1=\emptyset$ and $u_2=w_2$ and $x=x'$ and the result holds.

Case 9. Ω is process R;C.

The proof is similar to case 8.

Thus the theorem has been proved. □

Theorem 6.5

If $tr(\langle\Omega,s\rangle)\uparrow$ is an infinite computation then it must contain at least one translated label.

Proof The proof is by structural induction on Ω.

By looking at their complete computations it is easy to see that Ω cannot be skip, abort, x:=e, P?W(x) and P!W(e). Thus we only need to examine the following cases.

Case 1. Ω is b⇒BC;C.

Then $[\![b]\!]_s=tt$ and the transition sequence $tr(\langle\Omega,s\rangle)\uparrow$ must be the same as $tr(\langle BC;C,s\rangle)$. By the induction hypothesis we know that the transition sequence $tr(\langle BC;C,s\rangle)\uparrow$ contains at least one translated label.

Case 2. Ω is $GC_1 \square GC_2$.

The case is true since the computation from $tr(\langle\Omega,s\rangle)$ must be a

computation of one of $tr(\langle GC_i, s\rangle)$ i=1,2.

Case 3. Ω is if GC fi.

Then $[\![Bool(GC)]\!]_s$=tt and $tr(\langle \Omega, s\rangle)\uparrow$ must have the same computation as $tr(\langle GC, s\rangle)$. By the induction hypothesis the case is done.

Case 4. Ω is do GC od.

Then $[\![Bool(GC)]\!]_s$=tt and the initial transition sequence is

$$tr(\langle \Omega, s\rangle) \xrightarrow{\tau^*} \underline{if}\ s(Bool(GS))\ \underline{then}\ [\![s(GC)]\!]\ \underline{before}\ [\![\Omega]\!]\ \underline{else}\ [\![\underline{skip}]\!], s)$$

there are two subcases:

a. $tr(\langle GC, s\rangle)\uparrow$, then by the induction hypothesis it must contain at least one translated label so the case is proved.

b. the computation from $tr(\langle GC, s\rangle)$ is finite. Then it cannot be stuck, therefore applying lemma 6.5 we have:

$$tr(\langle GC, s\rangle) \xrightarrow{u} m(\underline{done}, s').$$

By theorem 6.2 $rt(u)\neq\emptyset$, that is u contains at least one translated label therefore $tr(\langle \Omega, s\rangle)\uparrow$ contains at least one translated label.

Case 5. Ω is $C_1;C_2$.

Let $tr(\langle C_1;C_2, s\rangle) \xrightarrow{\lambda 1} x_1 \xrightarrow{\lambda 2} x_2 \xrightarrow{\lambda 3} \ldots$

be an infinite computation and $u_n=\lambda 1\lambda 2\ldots\lambda n$. Applying lemma 6.5 we see three possibilities and consider the latter two first:

case 5.1 For some w_n, v_n and s_n we have $u_n=w_n\tau v_n$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{w_n} m(\underline{done}, s_n) \quad \text{and} \quad tr(\langle C_2, s_n\rangle) \xrightarrow{v_n} x_n$$

In this case, by theorem 6.2, w_n must contain at least one translated label and so does u_n.

case 5.2 For some w_n, v_n and s_n we have $u_n=w_nv_n$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{w_n} m(\underline{fail}, s_n) \xrightarrow{v_n} x_n$$

But this is not the case since the above computation is finite.

case 5.3 For every n we have for some t_n and s_n and
$x_n = m(t_n \underline{\text{before}} [\![C_2]\!], s_n)$ and $tr(\langle C_1, s\rangle) \xrightarrow{u_n} m(t_n, s_n)$

We have an infinite computation:

$$tr(\langle C_1, s\rangle) \xrightarrow{\lambda 1} m(t_1, s_1) \xrightarrow{\lambda 2} m(t_2, s_2) \xrightarrow{\lambda 3} \ldots$$

applying the induction hypothesis to C_1 we see that this computation must contain at least one translated label.

Case 6. Ω is $C_1 \| C_2$.

Let $tr(\langle C_1 \| C_2, s\rangle) \xrightarrow{\lambda 1} x_1 \xrightarrow{\lambda 2} x_2 \xrightarrow{\lambda 3} \ldots$
be an infinite computation and $u_n = \lambda 1 \lambda 2 \ldots \lambda n$. Then applying lemma 6.6 we see five possibilities and consider the latter four first:

case 6.1 For some w_n, v_n, s_n and t_{2n} we have $u_n = \bar{w}_n \tau v_n$ and

$$tr(\langle C_1 \| C_2, s\rangle) \xrightarrow{\bar{w}_n} m(\underline{\text{done}}\ \underline{\text{par}}\ t_{2n}, s_n)$$
$$m(t_{2n} \underline{\text{before}} [\![\underline{\text{skip}}]\!], s) \xrightarrow{v_n} x_n$$

and for some w_n', w_n'' we have $Merge(w_n', w_n'', w_n)$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{\bar{w}_n'} m(\underline{\text{done}}, sw_n')$$
$$tr(\langle C_2, s\rangle) \xrightarrow{\bar{w}_n''} m(t_{2n}, sw_n'')$$

and $s_n = sw_n' \Theta sw_n''$. By theorem 6.2 w_n' must contain at least one translated label, then so does w_n by the definition of merge.

case 6.2 For some w_n, v_n, s_n and t_{2n} we have $u_n = \bar{w}_n \tau v_n$ and

$$tr(C_1 \| C_2, s\rangle) \xrightarrow{\bar{w}_n} m(\underline{\text{fail}}\ \underline{\text{par}}\ t_{2n}, s_n)$$
$$m(t_{2n} \underline{\text{before}} [\![\underline{\text{abort}}]\!], s_n) \xrightarrow{v_n} x_n$$

and for some w_n', w_n'' we have $Merge(w_n', w_n', w_n)$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{\bar{w}'_n} m(\underline{fail}, sw'_n)$$

$$tr(\langle C_2, s\rangle) \xrightarrow{\bar{w}''_n} m(t_{2n}, sw'_n)$$

and $s_n = sw'_n \theta sw''_n$. As usual since $m(\underline{fail}, sw'_n) \xrightarrow{\tau+} tr(\underline{abortion})$ so by theorem 6.2 we see that w'_n must contain at least one translated label and so w_n contains at least one translated label.

case 6.3 Exchanging the positions of C_1 and C_2 in case 6.1, the proof is similar to case 6.1.

case 6.4 Exchanging the positions of C_1 and C_2 in case 6.2, the proof is similar to case 6.2.

case 6.5 For every n we have for that some t_{1n}, t_{2n} and s_n $x_n = m(t_{1n}\ \underline{par}\ t_{2n}, s_n)$ and for some u'_n, u''_n we have $Merge(u'_n, u''_n, u_n)$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{\bar{u}'_n} m(t_{1n}, su'_n)$$

$$tr(\langle C_2, s\rangle) \xrightarrow{\bar{u}''_n} m(t_{2n}, su''_n)$$

where $s_n = su'_n \theta su''_n$. We see that for some n at least one of u'_n or u''_n contains a translated label, otherwise since u_n is arbitrarily long so, by the definition of merge, at least one of u'_n or u''_n is arbitrarily long. Therefore one of the computations from $tr(\langle C_1, s\rangle)$ or $tr(\langle C_2, s\rangle)$ is infinite and by the induction hypothesis it must contain at least one translated label; and this is a contradiction.

Case 7. Ω is R::C.

Since $tr(\langle \Omega, s\rangle) = m([\![C]\!][\phi_R], s)$ by lemma 6.9 and 6.10 as usual we see that $tr(\langle C, s\rangle)\uparrow$. Then, by the induction hypothesis, the latter must contain at least one translated label and therefore the result is true.

Case 8. Ω is process R;C.

The proof is similar to case 7

Thus the theorem is proved. □

Theorem 6.6

The translation tr satisfies P4. That is $\xrightarrow{}_t$ in Ex-tr(Γ_p) is $\{\tau\}$ commutative.

Proof Let $\Omega \in Syn$ and $tr(\langle\Omega,s\rangle) \xrightarrow{u}_t x$. We are going to prove any computation at x is $\{\tau\}$ commutative. The proof is still by structural induction. Consider the following cases:

Case 1. Ω is one of the forms skip, abort, x:=e, P?W(x) and P!W(e). Then the computation from $tr(\langle\Omega,s\rangle)$ is deterministic and the result is immediate.

Case 2. Ω is $b \Rightarrow BC;C$.

Then $[\![b]\!]_s = tt$ and $tr(\langle b \Rightarrow BC;C,s\rangle) \xrightarrow{u}_t x$ is the same as $tr(\langle s(BC);C,s\rangle) \xrightarrow{u}_t x$. By the induction hypothesis computation at t is $\{\tau\}$ commutative.

Case 3. Ω is $GC_1 [] GC_2$.

According to lemma 6.13 any computation from $tr(\langle\Omega,s\rangle)$ must be:

$$tr(\langle\Omega,s\rangle) = m([\![GC_1]\!] + [\![GC_2]\!], s) \xrightarrow{\rho}_t x' \xrightarrow{u'}_t x$$

where $\rho \in tr(\Lambda_p)$. Thus if any computation from t contains a transition label τ then length(u)>1, and therefore the computation from $tr(\langle\Omega,s\rangle)$ is the same as a computation from $tr(\langle GC_i,s\rangle)$ i=1 or 2. By the induction hypothesis any computation at x is $\{\tau\}$ commutative.

Case 4. Ω is $C_1;C_2$.

Let $tr(\langle C_1;C_2,s\rangle) \xrightarrow{u} x$. Applying lemma 6.5 we see three possibilities of the form of x.

case 4.1 $x = m(t_1 \underline{\text{before}} [\![C_2]\!], s_1)$ and $tr(\langle C_1,s\rangle) \xrightarrow{u} m(t_1,s_1)$. Now we

still have two subcases:

a. t_1 is done or fail. Then computation at x is deterministic, so is commutative.

b. t_1 is neither done nor fail. Then suppose

$$x \xrightarrow{\lambda} x_1 \text{ and } x \xrightarrow{\tau} x_2$$

Notice here we only deal with one transition step, applying lemma 6.5 and theorem 6.2, so for some t_{11}, t_{12}, s_{11} and s_{12} we have:

$$x_1 = m(t_{11} \underline{\text{before}} [\![C_2]\!], s_{11}) \quad \text{and} \quad m(t_1, s_1) \xrightarrow{\lambda} m(t_{11}, s_{11}).$$

$$x_2 = m(t_{12} \underline{\text{before}} [\![C_2]\!], s_{12}) \quad \text{and} \quad m(t_1, s_1) \xrightarrow{\tau} m(t_{12}, s_{12}).$$

By the induction hypothesis on $m(t_1, s_1)$ (since $tr(\langle C_1, s\rangle) \xrightarrow{u} m(t_1, s_1)$) we have: either $\lambda = \tau$, $m(t_{11}, s_{11}) = m(t_{12}, s_{12})$ or for some t_{13}, s_{13} the following diagram commutes:

$$\begin{array}{ccc} m(t_1, s_1) & \xrightarrow{\lambda} & m(t_{11}, s_{11}) \\ \tau \downarrow & & \downarrow \tau \\ m(t_{12}, s_{12}) & \xrightarrow{\lambda} & m(t_{13}, s_{13}) \end{array}$$

Therefore by lemma 6.5 we see that either $\tau = \lambda$ and $x_1 = x_2$ or take $x' = m(t_{13} \underline{\text{before}} [\![C_2]\!], s_{13})$ and have

$$x_1 = m(t_{11} \underline{\text{before}} [\![C_2]\!], s_{11}) \xrightarrow{\tau} m(t_{13} \underline{\text{before}} [\![C_2]\!], s_{13})$$
$$x_2 = m(t_{12} \underline{\text{before}} [\![C_2]\!], s_{12}) \xrightarrow{\lambda} m(t_{13} \underline{\text{before}} [\![C_2]\!], s_{13})$$

So in this case the result is true.

case 4.2 For some u_1, u_2 and s_1 we have $u = u_1 \tau u_2$ and

$tr(\langle C_1, s\rangle) \xrightarrow{u_1} m(\underline{done}, s_1)$ and
$tr(\langle C_2, s_1\rangle) \xrightarrow{u_2} x$.

By the induction hypothesis on C_2 we see that computation at x is commutative.

case 4.3 For some u_1, u_2 and s_1 we have:

$$tr(\langle C_1, s\rangle) \xrightarrow{u_1} m(\underline{fail}, s_1) \xrightarrow{u_2} x$$

Then we see that computation at x is deterministic so is commutative.

Case 5. Ω is do GC od.

If $[\![Bool(GC)]\!]_s = ff$ then the result is immediate, since the computation from $tr(\langle \Omega, s\rangle)$ is deterministic. Otherwise there are two possibilities:

a. $tr(\langle \Omega, s\rangle) \xrightarrow{\tau^*} x$

$\xrightarrow{\tau^*} m(\underline{if}\ s(Bool(GC))\ \underline{then}\ ([\![s(GC)]\!]\ \underline{before}\ [\![\Omega]\!])\ \underline{else}\ [\![\underline{skip}]\!], s)$

Since the computation at x is deterministic so the result holds.

b. $m([\![s(GC)]\!]\ \underline{before}\ [\![\Omega]\!], s) \xrightarrow{u} x$

Then to prove that any computation at x is commutative we need examine three subcases which are similar to case 4. Let us consider the second subcase: for a given u we find some u_1, u_2 and s' and have $u = u_1 \tau u_2$

$$tr(\langle s(GC), s\rangle) \xrightarrow{u_1} m(\underline{done}, s') \quad \text{and} \quad tr(\langle \Omega, s'\rangle) \xrightarrow{u_2} x.$$

Here $length(u_2) < length(u)$. Since u is given so by repeating this procedure on the last transition finite times we will have for some s" either the above case (a) holds then the result is immediate or

$$tr(\langle s''(GC), s''\rangle) \xrightarrow{u''} x$$

By the induction hypothesis on GC the result is true.

Case 6. Ω is *if* GC *fi*.

The proof is similar to case 5.

Case 7. Ω is $C_1 \| C_2$.

Let $tr(\langle C_1 \| C_2, s\rangle) \xrightarrow{u} x$. Then applying lemma 6.6 we have the following five possibilities:

case 7.1 For some u', u_1 and u_2

$\bar{u}' = u$ and $Merge(u_1, u_2, u')$ and $x = m(t_1 \underline{par}\ t_2, s_1)$
$tr(\langle C_1, s\rangle) \xrightarrow{\bar{u}_1} m(t_1, su_1)$ and
$tr(\langle C_2, s\rangle) \xrightarrow{\bar{u}_2} m(t_2, su_2)$

There are still two cases:

case 7.1.1 Neither t_1 nor t_2 is *done* or *fail*. Let

$$x \xrightarrow{\lambda} x_1 \quad \text{and} \quad x \xrightarrow{\tau} x_2$$

For the first transition notice that we only consider one transition step at x, by lemmas 6.6 and theorem 6.2 we see that: for some ρ_1 and ρ_2 and $x_1 = m(t_{11} \underline{par}\ t_{21}, s_{11})$

$$m(t_1, s_1) \xrightarrow{\bar{\rho}_1} m(t_{11}, s_1\rho_1) \quad \text{and} \quad m(t_2, s_1) \xrightarrow{\bar{\rho}_2} m(t_{21}, s_1\rho_2)$$

and $Merge(\rho_1, \rho_2, \lambda')$ and $\bar{\lambda}' = \lambda$ and $s_{11} = s_1\rho_1 \oplus s_1\rho_2$.

For the second transition we have for some i (i=1,2), say i=1, $x_2 = m(t_{12} \underline{par}\ t_2, s_{12})$

and $m(t_1, s_1) \xrightarrow{\tau} m(t_{12}, s_{12})$

Now there are various possibilities:

a. $\rho_2 \neq \emptyset$. Then $\lambda' = \rho_1$ and $t_{21} = t_2$ applying the induction hypothesis to $m(t_1, s_1)$ we see two subcases:

a.1. $\tau = \lambda$ and $m(t_{11}, s_1\rho_1) = m(t_{12}, s_{12})$ then $x_1 = x_2$ and the result holds.

a.2 For some t_{13}, s_{13} the following diagram commutes:

$$\begin{array}{ccc} m(t_1, s_1) & \xrightarrow{\lambda} & m(t_{11}, s_1\rho_1) \\ \downarrow \tau & & \tau \downarrow \\ m(t_{12}, s_{12}) & \xrightarrow{\lambda} & m(t_{13}, s_{13}) \end{array}$$

So we take $x' = m(t_{13} \underline{\text{par}}\ t_2, s_{13})$ and apply lemma 6.6 have

$$\begin{array}{ccc} m(t_1 \underline{\text{par}}\ t_2, s_1) & \xrightarrow{\lambda} & m(t_{11} \underline{\text{par}}\ t_2, s_1\rho_1) \\ \downarrow \tau & & \tau \downarrow \\ m(t_{12} \underline{\text{par}}\ t_2, s_{12}) & \xrightarrow{\lambda} & m(t_{13} \underline{\text{par}}\ t_2, s_{13}) \end{array}$$

b. $\rho_1 \neq \emptyset$ and $\rho_2 \neq \emptyset$. Then $\lambda = \tau'$ and ρ_1, $\rho_2 \varepsilon tr(\Gamma_p)$ and $\rho_1 = \bar{\rho}_2$. In this case $\tau \neq \rho_1$ and similar to case (a.2) we take $x' = m(t_{13} \underline{\text{par}}\ t_{21}, s_{12})$ and apply lemma 6.6 and have:

$$\begin{array}{ccc} m(t_1 \underline{\text{par}}\ t_2, s_1) & \xrightarrow{\tau'} & m(t_{11} \underline{\text{par}}\ t_{21}, s_1) \\ \downarrow \tau & & \tau \downarrow \\ m(t_{12} \underline{\text{par}}\ t_2, s_{12}) & \xrightarrow{\tau'} & m(t_{13} \underline{\text{par}}\ t_{21}, s_{12}) \end{array}$$

c. $\rho_2 \neq \emptyset$ and $\rho_1 = \emptyset$. Then $\lambda' = \rho_2$. In this case the computation from x is interleaving its constituents, applying lemma 6.6 the result is certainly true.

case 7.1.2 One of t_1 or t_2 is $\underline{\text{done}}$ or $\underline{\text{fail}}$. For example, let $t_1 = \underline{\text{done}}$ then $x = m(\underline{\text{done}}\ \underline{\text{par}}\ t_2, s_1)$ and the proof is similar to case 7.1.1 (a) and (c).

case 7.2 For some u_1, u_2, t_2 and s' we have $u=\bar{u}_1 \tau u_2$ and

$$tr(\langle C_1 \| C_2, s\rangle) \xrightarrow{\bar{u}_1} m(\underline{done}\ \underline{par}\ t_2, s') \text{ and}$$
$$m(t_2 \underline{before} [\![\underline{skip}]\!], s') \xrightarrow{u_2} x$$

and for some u_1', u_1'' we have $Merge(u_1', u_1'', u_1)$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{\bar{u}_1'} m(\underline{done}, su_1') \quad \text{and} \quad tr(\langle C_2, s\rangle) \xrightarrow{\bar{u}_1''} m(t_2, su_1'')$$

By lemma 6.5 we have

$$tr(\langle C_2; \underline{skip}, s\rangle) \xrightarrow{\bar{u}_1'' u_2} x$$

So similar to the proof of case 4 we see that computation at x is commutative.

case 7.3 For some u_1, u_2, t_2 and s' we have $u=\bar{u}_1 \tau u_2$ and

$$tr(\langle C_1 \| C_2, s\rangle) \xrightarrow{\bar{u}_1} m(\underline{fail}\ \underline{par}\ t_2, s') \text{ and}$$
$$m(t_2 \underline{before} [\![\underline{abort}]\!], s') \xrightarrow{u_2} x$$

and for some u_1', u_1'' we have $Merge(u_1', u_1'', u_1)$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{\bar{u}_1'} m(\underline{fail}, su_1') \quad \text{and} \quad tr(\langle C_2, s\rangle) \xrightarrow{\bar{u}_1''} m(t_2, su_1'')$$

By lemma 6.5 we have

$$tr(\langle C_2; \underline{abort}, s\rangle) \xrightarrow{\bar{u}_1'' u_2} x$$

So similar to case 4 the result is true.

case 7.4 Exchanging the positions of C_1 and C_2 in case 7.2. The proof is similar to case 7.2.

case 7.5 Exchanging the positions of C_1 and C_2 in case 7.3. The

proof is similar to case 7.3.

Case 8. Ω is R::C.

Then applying lemma 6.9 we see two possibilities:

a. x has the form $m(t[\phi_R], s_1)$ and $tr(\langle C,s\rangle) \xrightarrow{u'}_{t} m(t,s_1)$ where $\phi_R(u')=u$. Applying lemma 6.9 we have

$$m(t[\phi_R],s_1) \xrightarrow{\rho} m(t'[\phi_R],s_2) \text{ and } m(t[\phi_R],s_1) \cdot \rightarrow m(t''[\phi],s_3)$$
$$\text{iff } m(t,s_1) \xrightarrow{\rho'} m(t',s_2) \text{ and } m(t,s_1) \xrightarrow{\tau} m(t'',s_3)$$

where $\phi_R(\rho')=\rho$. So by the induction hypothesis any computation at $m(t,s_1)$ is $\{\tau\}$ commutative, so therefore computation at $m(t[\phi_R],s_1)$ is commutative.

b. $tr(\langle R::C,s\rangle) \xrightarrow{u_1} m(\underline{fail},s') \xrightarrow{\tau^+} x$. Then computation at x is deterministic so is commutative.

Case 9. Ω is process R;C.

The proof is similar to case 8. Thus the lemma has been proved. □

Theorem 6.7

The translation tr satisfies P5. That is if $\langle\Omega,s\rangle$ is active and $tr(\langle\Omega,s\rangle) \xrightarrow{u} x$ and $rt(u)=\emptyset$ then there exist r', x', u_1 and u_2 such that

$$r \xrightarrow{\lambda} r' \text{ and } \begin{array}{ccc} tr(r) & \xrightarrow{u} & x \\ & & \downarrow u_2 \\ tr(r') & \xrightarrow{u_1} & x' \end{array} \qquad rt(u_2)=\lambda \text{ and } rt(u_1)=\emptyset$$

Proof. The proof is still by structural induction on Ω.

case 1. Ω is one of skip, abort, x:=e, P?W(x) and P!W(e). Since computation from $tr(\langle\Omega,s\rangle)$ is deterministic the result is obvious.

case 2. Ω is GC. By lemma 6.13 the first transition label from $tr(\langle GC,s\rangle)$ must be a translated label, so the result is obvious.

case 3. Ω is C1;C2.

Let $tr(\langle C_1;C_2,s\rangle) \xrightarrow{u} x$. Applying lemma 6.5 we see three possibilities:

case 3.1 $x=m(t_1 \underline{before} [\![C_2]\!], s_1)$ and $tr(\langle C_1,s\rangle) \xrightarrow{u} m(t_1,s_1)$. Since $C_1;C_2$ is active according to the composition rule (see chapter 2) C_1 must be active. So applying the induction hypothesis we can find r_1, x_1, u_1' and u_2' such that

$\langle C_1,s\rangle \xrightarrow{\lambda} r_1$ and

$$\begin{array}{ccc} tr(\langle C_1,s\rangle) & \xrightarrow{u} & m(t_1,s_1) \\ & & \downarrow u_2' \\ tr(r_1) & \xrightarrow{u_1'} & x_1 \end{array}$$

and $rt(u_2')=\lambda$ and $rt(u_1')=\emptyset$

Now there are three cases depending on the form of r_1:

a. $r_1=s'$. Then $tr(r_1)=m(\underline{done},s')$ and so $u_1'=\emptyset$ and $x_1=m(\underline{done},s')$. Take $r'=\langle C_2,s'\rangle$, $x'=m([\![C_2]\!],s')$, $u_1=\emptyset$ and $u_2=u_2'\tau$ then $rt(u_2)=\lambda$.

b. $r_1=\underline{abortion}$. Then by theorem 6.2 $x_1=tr(\underline{abortion})$ and so $u_1'=\emptyset$. Take $r'=\underline{abortion}$, $x'=tr(\underline{abortion})$, $u_1=\emptyset$ and $u_2=u_2'\tau$and $rt(u_2)=\lambda$.

c. $r_1=\langle C_1',s'\rangle$. Then $x_1=m(t_1',s')$ for some t_1'. So take $r'=\langle C_1';C_2,s'\rangle$ and $x'=m(t_1'\underline{before}[\![C_2]\!],s')$ and $u_1=u_1'$ and $u_2=u_2'$ and $rt(u_2)=\lambda$.

Applying lemma 6.5 we see that for these three cases the result is true.

case 3.2 For some w, v, t_2 and s' we have u=wτv and

$$tr(\langle C_1,s\rangle) \xrightarrow{w} m(\underline{done},s') \text{ and } tr(C_2,s'\rangle) \xrightarrow{v} x$$

According to theorem 6.2 $rt(w)\neq\emptyset$ so this is not the case.

case 3.3 For some w, v, t_2 and s' we have u=wv and

$$tr(\langle C_1,s\rangle) \xrightarrow{w} m(\underline{fail},s') \xrightarrow{v} x$$

According to theorem 6.2 $rt(u)\neq\emptyset$ so this is still not the case.

case 4. Ω is do GC od.

Then we have:

a. $tr(\langle \Omega,s\rangle) \xrightarrow{r^*}$

$$m(\underline{if}\ s(Bool(GC))\ \underline{then}\ [\![s(GC)]\!]\underline{before}[\![\Omega]\!]\underline{else}[\![\underline{skip}]\!]),s)$$

Note that the initial transition sequence is deterministic. Thus

If $[\![Bool(GC)]\!]_s=ff$ then the result is obvious, since the computation is deterministic.

If $[\![Bool(GC)]\!]_s=tt$ then according to the alternative rule GC must be active. Since computation from $tr(\langle\Omega,s\rangle)$ is the same as computation from $tr(\langle s(GC);\Omega,s\rangle)$, we can prove this case using a similar analysis to the proof of case 3.

case 5. Ω is if GC fi. The proof is obtained by induction on GC.

case 6. Ω is C1‖C2.

Now let $tr(\langle C_1 \| C_2,s\rangle) \xrightarrow{\bar{u}} x$ and $rt(u)=\emptyset$. Applying lemma 6.6 we have five possibilities:

case 6.1 For some u', u", t_1, t_2 and s_1 we have:

$x=m(t_1 \underline{par}\ t_2,s')$ and $merge(u',u'',u)$ and
$tr(\langle C_1,s\rangle) \xrightarrow{\bar{u}'} m(t_1,s_1)$ and $tr(C_2,s\rangle) \xrightarrow{\bar{u}''} m(t_2,s_2)$

where $s_1=su'$ and $s_2=su''$ and $s'=s_1 \theta s_2$.

According to the parallel rule we have the following cases:

case 6.1.1 Only one of C_1 and C_2 is active.

Let C_1 be active. Applying the induction hypothesis to C_1 for some

r_1, x_1, u_1' and u_2' we have:

$$\langle C_1, s\rangle \xrightarrow{\lambda} r_1 \quad \text{and} \quad \begin{array}{ccc} tr(\langle C_1, s\rangle) & \xrightarrow{\bar{u}} & m(t_1, s_1) \\ & & \downarrow \bar{u}_2' \\ tr(r_1) & \xrightarrow{\bar{u}_1'} & x_1 \end{array} \quad \text{and} \quad \begin{array}{l} rt(\bar{u}_2') = \lambda \\ \\ rt(\bar{u}_1') = \emptyset \end{array}$$

Here are still three subcases depending on the form of r_1.

a. $r_1 = \langle C_1', s''\rangle$. Then by theorem 6.2 $x_1 = m(t_1', s_1')$, where $s_1' = s''u_1' = s_1 u_2'$. So take $r' = \langle C_1' \| C_2, s''\rangle$, $x' = m(t_1' \underline{\text{par}}\ t_2, s3)$ and $s3 = s_1' \Theta s_2$ and $u_1 = u_1' u''$ and $u_2 = u_2'$ applying lemma 6.6 we have:

$$\begin{array}{ccc} tr(\langle C_1 \| C_2, s\rangle) & \xrightarrow{\bar{u}} & m(t_1 \underline{\text{par}}\ t_2, s') \\ & & \downarrow \bar{u}_2 \\ tr(\langle C_1' \| C_2, s''\rangle & \xrightarrow{\bar{u}_1} & m(t_1' \underline{\text{par}}\ t_2, s3) \end{array}$$

The horizontal transition is from $Merge(u_1', u'', u_1'u'')$ and $s3 = s_1' \Theta s_2 = s''u_1' \Theta su'' = s''(u_1'u'') = s''u_1$. $(FL(\llbracket C_1 \rrbracket) \cap FL(\llbracket C_2 \rrbracket) \cap M_a = \emptyset)$

the vertical transition is from $Merge(u_2', \emptyset, u_2')$ and $s3 = (s_1)u_2' = s'u_2'$.

b. $r_1 = s''$. Then $x_1 = m(\underline{\text{done}}, s'')$ and $u_1' = \emptyset$. So take $r' = \langle C_2; \underline{\text{skip}}, s''\rangle$, $x' = m(t_2 \underline{\text{before}} \llbracket \underline{\text{skip}} \rrbracket, s'')$, and $u_1 = u''$ and $u_2 = u_2'\tau$.

c. $r_1 = \underline{\text{abortion}}$. Then $x_1 = tr(\underline{\text{abortion}})$ and $u_1' = \emptyset$. By lemma 6.12 we have for some u_{21}'

$$tr(\langle C_1, s\rangle) \xrightarrow{\bar{u}'} m(t_1, s) \xrightarrow{\bar{u}_{21}'} m(\underline{\text{fail}}, s) \xrightarrow{\tau^+} tr(\underline{\text{abortion}})$$

So take $r' = \langle C_2; \underline{\text{abort}}, s\rangle$ and $x' = m(t_2 \underline{\text{before}} \llbracket \underline{\text{abort}} \rrbracket, s)$ note $Merge(u_{21}', \emptyset, u_{21}')$ and by lemma 6.6 we have

$x \xrightarrow{\bar{u}'_{21}} m(\underline{\text{fail}}\ \underline{\text{par}}\ t_2, s) \xrightarrow{\tau} m(t_2\ \underline{\text{before}}[\![\underline{\text{abort}}]\!], s)$

and

$tr(\langle C_2; \underline{\text{abort}}, s\rangle) \xrightarrow{\bar{u}''} m(t_2 \underline{\text{before}}[\![\underline{\text{abort}}]\!], s\ )$

and the case is proved.

case 6.1.2 Both C_1 and C_2 are active. Then applying the induction hypothesis to C_1 and C_2 we have

$\langle C_1, s\rangle \xrightarrow{\lambda_1} r_1$ and

$$\begin{array}{lll} tr(\langle C_1, s\rangle) \xrightarrow{\bar{u}'} m(t_1, s_1) & & rt(\bar{u}'_2)=\lambda_1 \\ \Big\downarrow \bar{u}'_2 & \text{and} & \\ tr(r_1) \xrightarrow{\bar{u}'_1} x_1 & & rt(\bar{u}'_1)=\emptyset \end{array}$$

and

$\langle C_2, s\rangle \xrightarrow{\lambda_2} r_2$ and

$$\begin{array}{lll} tr(\langle C_2, s\rangle) \xrightarrow{\bar{u}''} m(t_2, s_2) & & rt(\bar{u}''_2)=\lambda_2 \\ \Big\downarrow \bar{u}''_2 & \text{and} & \\ tr(r_2) \xrightarrow{\bar{u}''_1} x_2 & & rt(\bar{u}''_1)=\emptyset \end{array}$$

Now there are two subcases depending on the forms of λ_1 and λ_2.

a. $\lambda_1=\bar{\lambda}_2$. Then $\lambda_i \neq \varepsilon$ $i=1, 2$. By lemma 2.1, say, we have

$r_1=\langle C'_1, s\rangle$ and $r_2=\langle C'_2, s''\rangle$ and by Theorem 6.2

$x_1=m(t'_1, s_{12})$ and $x_2=m(t'_2, s_{22})$.

$s_{12}=su'_1=s_2u'_2$ and $s_{22}=s''u''_1=s_2u''_2$ and

$u'_2=u'_{21}tr(\rho_1)u'_{22}$ and $u''_2=u''_{21}tr(\rho_2)u''_{22}$

So take $\lambda=\varepsilon$ and $r'=\langle C'_1 \| C'_2, s''\rangle$ and $x'=m(t'_1 \underline{\text{par}}\ t'_2, s_{12}\theta s_{22})$.

Let $u_1=u'_1u''_1$ and $u_2=u'_{21}u''_{21}\tau' u'_{22}u''_{22}$. Then applying lemma 6.6 we have:

$$\begin{array}{lll} tr(\langle C_1 \| C_2, s\rangle) \xrightarrow{\bar{u}} m(t_1 \underline{\text{par}}\ t_2, s') \\ \Big\downarrow \bar{u}_2 \\ tr(\langle C'_1 \| C'_2, s''\rangle) \xrightarrow{\bar{u}_1} m(t'_1 \underline{\text{par}}\ t'_2, s_{12}\theta s_{22}) \end{array}$$

and the case is proved.

b. $\lambda_1 \neq \bar{\lambda}_2$. Then there is no interaction between C_1 and C_2. So in this case the proof is similar to case 6.1.1.

case 6.2 For some w_1, w_2 and s' and t_2 we have $u=\bar{w}_1 \tau w_2$ and

$$tr(\langle C_1 \| C_2, s\rangle) \xrightarrow{\bar{w}_1} m(\underline{done}\ \underline{par}\ t_2, s') \text{ and}$$
$$m(t_2 \underline{before} [\![\underline{skip}]\!], s') \xrightarrow{w_2} x$$

and for some w_1', w_1'' we have $Merge(w_1', w_1'', w_1)$ and

$$tr(\langle C_1, s\rangle) \xrightarrow{\bar{w}_1'} m(\underline{done}, sw_1') \text{ and}$$
$$tr(\langle C_2, s\rangle) \xrightarrow{\bar{w}_1''} m(t_2, sw_1'')$$

where $s'=sw_1' \theta sw_1''$. By theorem 6.2 $rt(w_1') \neq \emptyset$ so $rt(w_1) \neq \emptyset$ and this is not the case.

case 6.3 For some w_1, w_2, s' and t_2 we have $u=w_1 \tau w_2$ and

$$tr(\langle C_1 \| C_2, s\rangle \xrightarrow{\bar{w}_1} m(\underline{fail}\ \underline{par}\ t_2, s') \text{ and}$$
$$m(t_2 \underline{before} [\![\underline{abort}]\!], s') \xrightarrow{w_2} x$$

Similar to case 6.2 we see $rt(w_1) \neq \emptyset$ so this is still not the case.

case 6.4 Exchange the positions of C_1 and C_2 in case 6.2. Then the proof is similar to case 6.2.

case 6.5 Exchange the positions of C_1 and C_2 in case 6.3. Then the proof is similar to case 6.3.

case 7. Ω is R::C.

Then C must be active. Since $rt(u)=\emptyset$ by lemma 6.9 we have $tr(\langle C, s\rangle) \xrightarrow{u} x_1$, and by theorem 6.2 we know $x=m(t[\phi_R], s_1)$ and $x_1=m(t, s_1)$. So by the induction on C we have:

$$tr(\langle C,s\rangle) \xrightarrow{u} x_1$$

$\langle C,s\rangle \xrightarrow{\lambda} r_1$ and u_{12} $rt(u_{12})=\lambda$ and $rt(u_{11})=\emptyset$

$$tr(r') \xrightarrow{u_{11}} x_2$$

Then by lemma 2.1 r_1 can only be $\langle C',s'\rangle$ or s' or abortion. Let us consider the first case. Since $rt(u_{11})=\emptyset$ by theorem 6.2 we have $x_2=m(t',s'')$. So take

$r'=\langle R::C',s'\rangle$, $x'=m(t'[\phi_R],s'')$ and $u_1=u_{11}$

Noticing $rt(u_{11})=\emptyset$ so $\phi_R(u_1)=u_1=u_{11}$. and by lemma 6.9 $tr(r') \xrightarrow{u_1} m(t'[\phi_R],s'')=x'$.

Since R::C is suntactically correct and by lemma 2.3 λ can only be ε or $(*,P)!W(v)$ or $(*,P)?W(v)$. Therefore by L-process rule in section 2.3 we have

$$\langle R::C,s\rangle \xrightarrow{\phi_R(\lambda)} \langle R::C',s'\rangle$$

and $\phi_R(rt(u_{12})$ is defined. So take $u_2=\phi_R(u_{12})$ and by lemma 6.9 we have

$m(t[\phi_R],s) \xrightarrow{\phi_R(u_{12})} m(t'[\phi_R],s')$ that is $x \xrightarrow{u_2} x'$.

and the case is proved.

case 8. Ω is process R;C.

Then C must be active and the proof is similar to case (7).

The theorem is proved. □

Theorem 6.8

The translation $tr:\mathbb{T}_p \rightarrow \mathbb{T}_t'$ is an adequate translation.

Proof The proof is directly from corollary 1, theorems 3 to 7 and lemma 5.10. □

Corollary 6.2

The translation $tr' \circ tr:\mathbb{T}_p \rightarrow \mathbb{T}_c$ from CSP to CCS is correct.

7. A proposal for implementing multitasking in Ada

A proposal for implementing multitasking in Ada is given in this chapter. The approach is that the implementation is composed of two syntax-directed translations. We first the translate multitasking constructs of Ada into an Edison-like language (Edison.1). More precisely, in the first translation the entries of a task are translated to give a module which contains message buffers, and the communication statements of Ada are translated into calls to the corresponding procedures of this module. The second stage of the translation is to implement the when statement using a language we call Edison.0 consisting of sequential Ada plus some primitive constructs for scheduling.

In section 7.1 a translation algorithm from Ada.1 into Edison.1 is given. The language Edison.0 is introduced in section 7.2.1 and a translation from Edison.1 into Edison.0 is given in section 7.2.3.

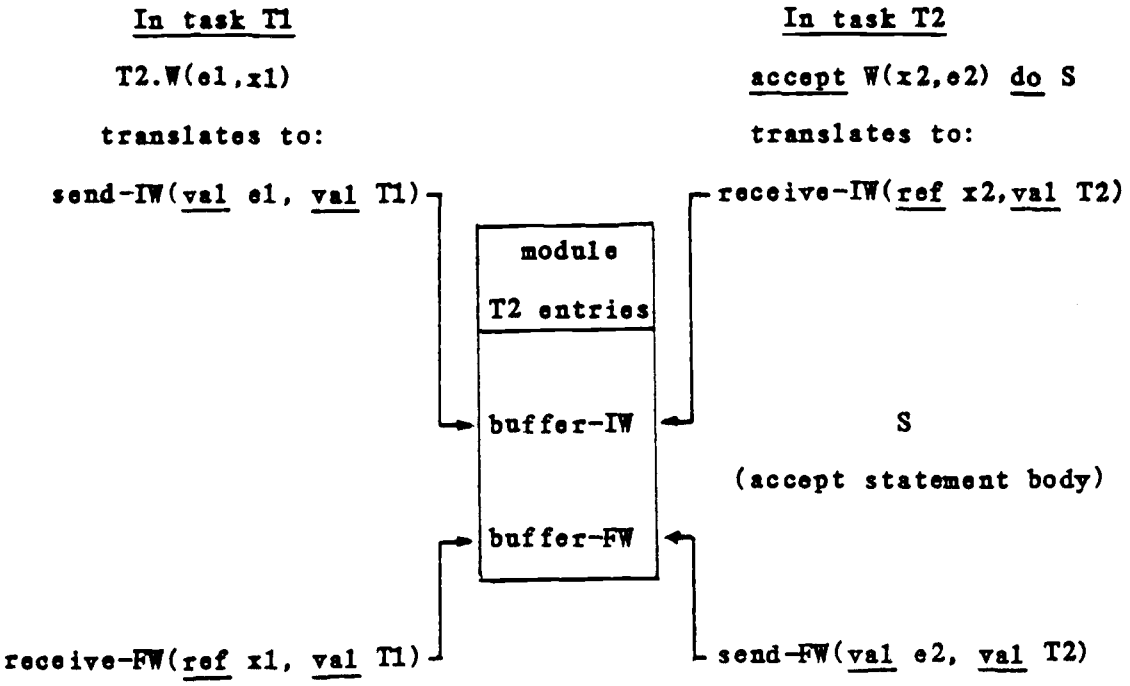
7.1 A translation from Ada.1 into Edison.1

In this section we give a translation algorithm from Ada.1 into Edison.1. At first glance the communication mechanisms in Ada.1 and in Edison seem completely different since communication between Ada tasks is achieved by rendezvous and communication in Edison is based on the idea of exclusive access to shared variables. But when we study their working processes thoroughly we will discover the common ground between these two mechanisms, and this will lead us to a simple translation algorithm which gives efficient implementation of communication.

As we discovered in section 3.1 a rendezvous in Ada consists of three phases: the initialization phase, the phase of executing the accept statement body and the termination phase. In the

initialization phase synchronization between the calling task and the called task begins and the calling task sends a value (maybe only a signal) to the called task which receives that value. During the second phase the called task executes the accept statement body and the calling task waits for an acknowledgement. In the termination phase the called task sends an acknowledgement accompanied by a value to the calling task, and the calling task receives this acknowledgement and then the synchronization (and of course the rendezvous) is finished.

It is easy to see that during a rendezvous communication must occur in the first and the third phases. These communications can be implemented in Edison.1 using a simple communication mechanism --- a message buffer. We will need two buffers for each entry W, one for the initialization phase (called buffer-IW --- I is for initialization, W is the entry name), one for the termination phase (called buffer-FW --- F is for termination (finale) and W is the entry name). A rendezvous is implemented by sending and receiving messages through these buffers as follows:



Each send and receive call must include the name of the calling task as a parameter; this is necessary to keep track of multiple rendezvous with the same entry W.

Suppose that, as in the diagram above, task T2 owns entry W, and suppose that x2 and e2 are of type integer. In fact Edison.1 as described in chapter 4 does not include types at all, but for expository reasons we will pretend it has an Ada-like type scheme for the moment. Suppose that taskname is a predefined type and stack is defined to be a stack of tasknames. Then buffer-IW can be implemented as follows:

```
module T2-entries(..., send-IW, receive-IW,...)
        ...
        var full-IW: bool;
        var buffer-IW: integer
        var last-IW-sender: taskname;
        var callstack-W: stack;
        ...
        proc send-IW(val i: integer, val sender: taskname)
             when not full-IW ⇒(buffer-IW:=i;
                                last-IW-sender:=sender;
                                full-IW:=true) end;
        proc receive-IW(ref i: integer, val receiver: taskname)
             when full-IW and receiver=T2 ⇒
                               (i:=buffer-IW;
                                push(last-IW-sender, callstack-W);
                                full-IW:=false) end;

   full-IW:=false; callstack:=empty;
```

If we omit the variables, parameters and statements dealing with task names then this module is a typical message buffer as studied in chapter 4. Since an accept statement for an entry may contain other accept statements for the same entry (or for other entries) the variable last-IW-sender and the stack callstack-W are introduced to deal with nested accept statements. The variable last-IW-sender contains the name of the most recent calling task for entry W (i.e the last caller of send-IW). When this call is received (i.e. when task T2 calls receive-IW) this name is pushed onto callstack-W. Thus the name of the calling task which most recently finished the initialization phase (and has not yet entered the termination phase) is always at the top of the stack. Later we will see that the stack is used by the termination phase (procedures send-FW and receive-FW) to keep track of the order of acknowledgements.

Note that T2 is the only task which should call receive-IW and so the guard includes a test for this. Although this condition will always be satisfied in a program which is the result of translating from Ada.1, it makes the proof of adequacy tidier.

Similarly, buffer-FW is:

```
module T2-entries(..., send-FW, receive-FW,...)

        var full-FW: bool;
        var buffer-FW: integer;
        var caller-W: taskname;
        ...
        proc send-FW(val i: integer, val sender: taskname)
             when not full-FW and sender=T2 ⇒
                       (buffer-FW:=i;
                        pop(caller-W, callstack-W);
                        full-FW:=true) end;
        proc receive-FW(ref i: integer, val receiver: taskname)
             when full-FW and receiver=caller-W ⇒
                       (i:=buffer-FW;
                        full-FW:=false) end;

   full-FW:=false;
```

Acknowledgements must be sent in the reverse order to that in which calls were received. The most recent caller for entry W is at the top of callstack-W, so when task T2 calls send-FW to send an acknowledgement callstack-W is popped and the top value stored in the variable caller-W. All of the (calling) tasks which have completed the initialization phase have called receive-FW and are waiting for buffer-FW to become full, and the value of caller-W is used to pick out the proper one to acknowledge (see the guard in receive-FW).

The module T2-entries will contain not only procedures send-IW, receive-IW, send-FW and receive-FW, but also the corresponding

procedures for all other entries owned by task T2.

Having provided a rendezvous mechanism we can now study the translation algorithm for the statements. Noticing that skip, abort, assignment, select and loopselect are same in Ada.1 and in Edison.1 (but select and loopselect are called if and do respectively), we focus our attention on translating the call, accept and multitask statements. We assume that a block statement of Edison is extended to be a statement.

Since the name of the task containing an entrycall or accept statement is needed as a parameter for the calls of send-IW, receive-IW, send-FW and receive-FW these generate, the translation algorithm must be parameterised by the name of the task (say L). As suggested above the translation of the entrycall, accept and multitask statements are:

$[\![$T.W(e,x)$]\!]_L$= T-entries.send-IW(val e, val L);
T-entries.receive-FW(ref x, val L)

$[\![$accept W(x,e) do S$]\!]_L$= L-entries.receive-IW(ref x, val L);
$[\![$S$]\!]_L$;
L-entries.send-FW(val e, val L)

$[\![$wait(T.W,x)$]\!]_L$= T-entries.receive-FW(ref: x, val: L)

$[\![$ack(T,W,e)$]\!]_L$= L-entries.send-FW(val: e, val: L)

$[\![$T::E,S$]\!]_L$= $[\![$S$]\!]_T$

$[\![MS_1 \| MS_2]\!]_L$= $[\![MS]\!]_L \| [\![MS]\!]_L$

```
[[task T::E, MS]]_L = module T-entries:(exported-proc-names(E))
                        (var mutex-T;
                         local-vars(E);
                         exported-procs(E)_T;
                         mutex-T:=false; init(E));
                      [[MS]]_T
```

where exported-proc-names(E) defines exported procedure names as follows

```
exported-proc-names(E1;E2) = exported-proc-names(E1);
                             exported-proc-names(E2)

exported-proc-names(W) = send-IW(val i, val sender);
                         receive-IW(ref i, val receiver);
                         send-FW(val i, val sender);
                         receive-FW(ref i, val receiver)
```

local-vars(E) defines local variables

```
local-vars(E1;E2)= local-vars(E1); local-vars(E2)

trv(W)= var full-IW; var full-FW;
        var buffer-IW; var buffer-FW;
        var last-IW-sender; var caller-W;
        var callstack-W;
```

exported-procs(E) defines exported procedures

exported-procs(E_1;E_2)$_T$= exported-procs(E_1)$_T$;
exported-procs(E_2)$_T$

exported-procs(W)$_T$=

```
proc send-IW(val i, val sender)
     with mutex-T when not full-IW ⇒
                (buffer-IW:=i;
                 last-IW-sender:=sender;
                 full-IW:=true) end;

proc receive-IW(ref i, val receiver)
     with mutex-T when full-IW and receiver=T ⇒
                (i:=buffer-IW;
                 push(last-IW-sender, callstack-W);
                 full-IW:=false) end;

proc send-FW(val i, val sender)
     with mutex-T when not full-FW and sender=T⇒
                (buffer-FW:=i;
                 pop(caller-W, callstack-W);
                 full-FW:=true) end;

proc receive-FW(ref i, val receiver)
     with mutex-T when full-FW and receiver=caller-W ⇒
                (i:=buffer-FW;
                 full-FW:=false) end;
```

init(E_1;E_2)= init(E_1); init(E_2)

init(W)= full-IW:=false;
full-FW:=false;
callstack:=empty

The translation differs slightly from the above explanation in that we replace the simple form of the when statement by the general form:

with mutex-T when GS end

where mutex-T is a local boolean variable for each task T (see section 4.3). Thus our use of this form means that mutual exclusion on access to message buffers will be task-wide rather than program-wide.

It should be mentioned that Luckham and his colleagues proposed a method for identifying techniques for efficient implementation in a similar way by translating Ada constructs concerned with multitasking into an intermediate language, Adam (see [Luckham et al 81] and [Stevenson 80]). The translation algorithm given here is simpler and more efficient than those given in [Stevenson 80] since the key point of our translation algorithm is translating entries into message buffers. In [Stevenson 80] the basic idea is that each entry of a task is translated into a procedure containing the bodies of all the accept statements for that entry, and the rest of the body of the task is itself translated into a separate procedure, with calls to a scheduler replacing the accept statements. The shortcomings of this method are:

1. It destroys the integrity of the text of programs since accept

statements become calls, with the translation of the accept statement bodies positioned elsewhere.

2. In Ada several different accept statements associated with the same entry may occur in a task, and an accept statement may contain several different accept statements for the same entry in its body. In order to make the method work in these cases, they introduce extra notions such as "synchronization levels" which further complicate the algorithm and make it more difficult to understand, to formalise and to prove correct.

7.2 Translating Ada.1 into Edison.0

In this section we investigate how to translate the when statement. We first introduce a small language called Edison.0 which consists of sequential Edison plus some primitive constructs for scheduling, and then we study the translation algorithm.

7.2.1 An introduction to Edison.0

The language Edison.0 is a lower-level concurrent programming language. It contains all the language entities in Edison.1 except the when statement.

Edison.0 also contains the following predefined types:

Semaphore

Variables of type boolean are used to implement primitive mutual exclusion. When a variable is used for this purpose we call it a semaphore. The following two indivisible procedures may be used to implement mutual exclusion:

set(var sem:bool): busy waits until sem is ff, then gains exclusive access by assigning tt to sem.

reset(var sem: bool): gives up exclusive access by changing sem from tt to ff.

These are just like the well-known P and V operations.

Queues

Values of type queue are FIFO (first in first out) queues of task names. For simplicity, we assume that queues cannot be full. The following three procedures are operations on queues and all these operations are indivisible:

insert(var T: taskname, var Q: queue): inserts the task name T in the queue Q.

remove(ref T: taskname, var Q: queue): removes the first task name from the queue Q and returns it as the value of T.

empty(Q: queue) returns tt if the queue Q is empty, otherwise ff.

Q-matrix

Values of type q-matrix are two dimensional arrays defined by:

```
type mode is (receive-I, send-I, receive-F, send-F)
type q-matrix is array(mode, entryname) of queue
```

Let us consider a portion of a typical q-matrix qm:

send-I	qm(send-I,W)
receive-I	qm(receive-I,W)
send-F	qm(send-F,W)
receive-F	qm(receive-F,W)

This shows that for every entry named W there are four queues of tasknames. All tasks in the queue qm(send-I,W) are tasks calling that entry in the initialization phase, in order of arrival. The

queue qm(receive-I,W) contains at most one task name,the called task name (the owner of W). If this queue is nonempty then the called task has reached an accept statement with entry W.

The type mode has an operation cmpl defined by:

cmpl(send-I)=receive-I and cmpl(receive-I)=send-I
cmpl(send-F)=receive-F and cmpl(receive-F)=send-F

The operation cmpl says that the complement of a receiving action is a sending action and vice versa.

Finally, we assume that the implementation of Edison.0 will provide a minimal kernel for control of task execution. Interfacing to this kernel is accomplished in the language by assuming a predefined scheduling module SUPERVISOR with the following visible procedures:

suspend: results in the suspension of the calling task.

activate(var T:taskname): reactivates the task T after suspension.

All these predefined procedures are assumed to be implemented using standard methods, for example those in [Brinch-Hansen 77].

7.2.2 Translating Edison.1 into Edison.0

In this subsection we investigate the problem of translating Edison.1 to Edison.0. We should have already realised that:

Firstly, the only difference between Edison.1 and Edison.0 is that in place of the when statement in Edison.1, Edison.0 uses the lower-level parallel facilities introduced in the previous subsection.

Secondly, in general a when statement can occur anywhere in an Edison program, but in a program which results from translation of an Ada.1 program a when statement can only appear in the procedures send-IW, receive-IW, send-FW and receive-FW.

Therefore to obtain a translation algorithm from Edison.1 to Edison.0 we only need a translation algorithm for the when statement in these special contexts. The idea is to introduce a local module named SCHD into every module generated by the translation algorithm described in section 7.1. The module SCHD has two visible procedures ENTER and EXIT for scheduling. We translate the when statement in these special contexts as follows:

$[\![$when GS end$]\!]_{(T,W,m)}$

=SCHD.ENTER(Bool(GS),T,W,m); if GS fi; SCHD.EXIT(T,W,m)

where (T,W,m) denotes the context: T is the calling task name, W is the entry involved and the mode m means that

m	meaning
send-I	Ω is in a send-IW procedure.
receive-I	Ω is in a receive-IW procedure.
send-F	Ω is in a send-FW procedure.
receive-F	Ω is in a receive-FW procedure.

where Ω denotes the when statement. Bool(GS) is the disjunction of the guards in GS (see section 4.2).

The local module SCHD is defined below:

```
module SCHD(ENTER,EXIT)
   var tm: taskname;
   var qm: q-matrix;

   proc ENTER(val is-open: bool, t: taskname, w: entryname, m: mode)
        set(mutex-T);
        do not is-open → insert(t,qm(m,w));
                         reset(mutex-T);
                         suspend;
                         set(mutex-T) od
   end;

   proc EXIT(t: taskname, w: entryname, m: mode)
        if empty(qm(cmpl(m),w))      →reset(mutex-T)
          □
          not empty(qm(cmpl(m),w)) → remove(tm,qm(cmpl(m),w));
                                     reset(mutex-T);
                                     activate(tm)
        fi;
   end;
begin mutex-T:=false; qm:=empty end
```

Let us explain the meaning of module SCHD informally. Consider the case m=receive-I, i.e. the when statement occurs in a receive-IW procedure. The procedure ENTER first gains exclusive access. It then says that if none of the guards of the when statement are open then insert the task name T into the queue qm(receive-I,W), give up exclusive access and suspend the calling task; otherwise exit from the procedure ENTER but retain exclusive access (so control will execute the when statement body). The procedure EXIT says that if

the queue qm(send-I,W) is empty then give up exclusive access; otherwise remove the first task waiting in this queue, give up exclusive access and activate that task.

Putting the translation of the when statement and the module SCHD together, a when statement is implemented as follows:

1. Call exported procedure ENTER of the module SCHD.

1.a. Gain exclusive access; i.e. at any time only one of the when statements occurring in a module T-entries as given in section 7.1 can be in its critical phase.

1.b. If Bool(GS)=ff (that is, all guards of GS are closed) then the when statement cannot be executed. Therefore insert the name of this task in the corresponding send or receive waiting queue of the entry W, give up exclusive access and suspend the this task. By the wait queue corresponding to entry W we mean that if the when statement is in a procedure receive-IW or receive-FW then put the name of task which calls this procedure into the queue qm(receive-I,W) or qm(receive-F,W) respectively; if the when statement is in a procedure send-IW or send-FW then put the name of this task into the queue qm(send-I,W) or qm(send-F,W) respectively.

If Bool(GS)=tt (i.e. at least one of the alternatives is open) then exit from procedure ENTRY, but retain exclusive access.

2. Execute the if statement, i.e, one of the open alternative may be chosen and executed.

3. The completion of the if statement means that a send (or receive) action has succeeded, and its complementary action (if any) waiting in the corresponding queue is available for execution. Activating the waiting task is the job of the procedure EXIT which says that if there is no complementary action waiting in the queue then just give up exclusive access; otherwise remove the first one from the

complementary queue, give up exclusive access and activate that task.

Having explained the idea of the translation we now give the formal translation algorithm:

```
[[with mutex-T when GS end]](T,W,m) =
    SCHD.ENTER(val Bool(GS), val T, val W, val m);
    if GS fi;
    SCHD.EXIT(val T, val W, val m)

[[proc send-IW(val i, val sender) BS ]](T,W,m) =
    proc send-IW(val i, val sender)
    [[BS]](sender,W,send-I)

[[proc receive-IW(ref i, val receiver) BS]](T,W,m) =
    proc receive-IW(ref c: integer, val receiver: taskname)
    [[BS]](receiver,W,receive-I)

[[proc send-FW(val i, val sender) BS ]](T,W,m) =
    proc send-FW(val i, val sender,)
    [[BS]](sender,W,send-F)

[[proc receive-FW(ref i, val receiver) BS]](T,W,m) =
    proc receive-FW(ref i, val receiver)
    [[BS]](receiver,W,receive-F)

[[module T-entries(EP) BS]](T,W,m) = module T-entries(EP)
                                     SCHD(T-entries);
                                     [[BS]](T,W,m);
```

```
where  SCHD(T-entries) =

module SCHD(ENTER,EXIT)
        var: tm;
        var: qm;

        proc ENTER(val is-open, val t, val w, val m)
             set(mutex-T);
             do not is-open ⇒  (insert(t,qm(w,m));
                               reset(mutex-T);
                               suspend;
                               set(mutex-T))
             od;

        proc EXIT(val t, val w, val m)
          if
             empty(qm(cmpl(m),w)) ⇒  reset(mutex-T)
             []
             not empty(qm(cmpl(m),w)) ⇒ remove(tw,qm(cmpl(m),w));
                                        reset(mutex-T);
                                        activate(tm)
          fi;
begin  qm:=empty end
```

Thus we have completed a translation algorithm from Edison.1 to Edison.0.

It should be mentioned that we can obtain a direct translation algorithm from Ada.1 to Edison.0 by composing this translation algorithm and the translation algorithm given in the previous section. In fact, this can be done by introducing module SCHD into

every module T-entries given in section 7.1 and replacing when statements occurring in procedures send-IW, receive-IW, send-FW, and receive-FW by their translations as given above. Therefore to implement multitasking in Ada we only need to implement Edison.0. The language Edison.0 does not exist in reality; it is only a tool which we use to explain our translation algorithm. What is important is that the target language must be composed of all the sequential constructs of Ada and constructs for multitasking without communication and some primitive structures for scheduling as given in section 7.2.1. In fact the when statement is just a device for explaining the translation algorithm and helping to prove its correctness.

Finally, strictly speaking, in this chapter we have not described how to implement multitasking but only how to implement communication between cooperating processes. The problem of implementing multiple parallel processes is not touched upon. This is an operating system problem with an almost standard solution.

8. Conclusion

In conclusion we summarise the achievements and the inadequacies of the work presented here and discuss areas for future research.

Generally speaking our work is in two areas: investigating the semantics for concurrent programming languages and studying the translation problem between these languages.

In the semantics area we use the structural operational approach to give the semantics of four representative languages for concurrent programming. They are CCS, CSP, Ada and Edison. The work includes:

a. Formalising the static semantics for all syntactic entities.

b. Giving semantics for the standard sequential constructs of these languages.

c. Describing the parallel structure explicitly and defining its semantics by the interleaving of its constituent actions.

d. Investigating the semantics of handshaking communication mechanisms, such as in CCS, CSP and Ada.

e. Investigating the semantics of communication mechanisms, such as those in Edison, which use mutually exclusive access to shared variables.

f. Studying the semantics of exception handling and the interaction between exception and communication.

g. Studying the semantics of declaration and the scope of variables, procedures and modules in Edison, and of processes and tasks in CSP and Ada.

h. Proving the properties of these semantics and the interactions between static semantics and dynamic semantics.

As we have already seen, the structural operational approach is indeed a powerful tool to define the semantics for concurrent programming languages. Its flexible expressive power makes possible the description of many kinds of language structure involving parallelism and communication. Using this approach the semantics for languages can be constructed without employing heavy mathematical machinery while still providing a solid base for proving the semantic properties of programs. Its operational "nature" is close to the programming experience of programmers and thus may be more acceptable to them.

Our work in this area is far from complete and further research is needed to overcome the following inadequacies:

We defined the semantics for a parallel structure by interleaving its constituents and thus the concurrency happens in a rather "macroscopic" sense, i.e. only asynchronous interaction is considered. We did not investigate the semantics for the real time constructs which are an important part of some concurrent programming languages. Perhaps ideas like those of Milner [Milner 82] might guide and motivate such a study.

To focus on parallelism and communication we omitted and did not concern ourselves with the abstract data types which play an important role in the modern concurrent programming languages. Here a lot of further research is needed.

In the translation area, we introduced the concept of correctness of a translation between two concurrent programming languages and proved the composition theorem. The adequacy problem was also

studied and a set of sufficient conditions was found. As applications we gave a syntax-directed translation from CSP to CCS, and proved its correctness, and declared a proposal for implementing multitasking in Ada.

There are several counts on which our work is certainly inadequate. We have not examined the composition problem for adequate translations. Basically, our definition of correctness does not mention the behaviour but only the results of the computations of the program and its translation. The definition of adequacy deals with the behaviour of the program and its translation, but the sufficient conditions are not as succinct as could be hoped for and are rather complicated. As a result of this, the proof of the adequacy for a translation is too long and complicated. Recently, more powerful mathematical tools for transition relations have appeared , such as the bisimulations given by Milner [Milner 82] and some equivalence relations given by de Nicola and Hennessy [de Nicola and Hennessy 82]. Introducing these concepts may simplify and strengthen the conditions. For example, those dealing with commutativity (P2) and activity (P5) may be omitted.

Finally, in a semantics for a nontrivial language it is not difficult to define and understand individual rules, but it is difficult to manage and understand all the interactions between rules. The proofs involving semantics are basically by structural induction and are not difficult to construct but rather detailed and boring. Therefore automatic aids are a possibility for checking details of this type of proofs. For example, interactive theorem proving systems like LCF [Gordon, Milner and Wardsworth 79] could be used to prove the semantic properties and the conditions of adequacy.

9. References

LNCS n denotes Lecture Notes in Computer Science Volume n, Springer-Verlag.

[Apt 81] Apt, K.R, Recursive assertions and parallel programs, Acta Informatica, Vlo.15, 1981.

[Brinch-Hansen 77] Brinch Hansen, P., The architecture of concurrent programs, Prentice-Hall, 1977.

[Brinch-Hansen 81a] Brinch Hansen, P., Edison -— a multiprocessor language, Software-Practice and Experience, Vol.11, 1981.

[Brinch-Hansen 81] Brinch Hansen, P., The design of Edison, Software-Practice and Experience, Vol.11, 1981.

[Bjórner and Oest 80] Bjorner, D., Oest, O.N., Towards a formal description of Ada. LNCS 98, 1980.

[Burstall and Goguen 81] Burstall, R.M., Goguen, J.A., Algebras, theories and frees: an introduction for computer science. The University of Edinburgh, 1981.

[Curry, Feys, Craig 68] Curry, H.B., Feys, R., Craig, W., Combinatory logic. Vol.1, Amsterdam, North Holland, 1968.

[Burstall and Landin 69] Burstall, R.M., Landin, P.J., Programs and their proofs: an algebraic approach, Machine Intelligence 4, The University of Edinburgh, 1969.

[de Nicola and Hennessy 82] de Nicola, R., Hennessy, M.C.B, Testing Equivalences for processes, CSR-123-82, The University of Edinburgh, 1982.

[Dijkstra 76] Dijkstra, E., A Discipline of programming, Prentice

Hall, 1976.

[Gordon 79] Gordon, M.,The Denotational description of programming languages, Springer-Verlag, 1979.

[Gordon, Milner and Wadsworth] Gordon, M., Milner, R. and Wadsworth, C.P, Edinburgh LCF, LNCS 78, 1979.

[Hennessy and Plotkin 80] Hennessy, M., Plotkin, G., A term model for CCS, LNCS 88, 1980.

[Hennessy, Li and Plotkin 81] Hennessy, M., Li, W., Plotkin, G., A first attempt at translating CSP into CCS. Proceedings of the Second International Conference on Distributed Systems, Paris, 1981.

[Hennessy and Li 82] Hennessy, M., Li, W., Translating Ada tasking into CCS, Proceedings of IFIP working conference, 1982.

[Hindley, Lercher, Seldin 72] Hindley, J.R., Lercher, B., Seldin, J.P., Introduction to combinatory logic. Cambridge University Press, Cambridge, 1972.

[Hoare 74] Hoare, C.A.R, Monitors: an operating system structuring concept, CACM 18, 10, 1974.

[Hoare 78] Hoare, C.A.R., Communicating sequential processes, CACM 21, 8, 1978.

[Hoare 81] Hoare, C.A.R., The Emperor's old clothes, CACM 24, 2, 1981.

[Huet 80] Huet, G., Confluent reduction: abstract properties and applicationsto term rewriting system. JACM, Oct. 1980, Vol 27, No 4.

[Jensen 79] Jensen, K., A method to compare the description power of different Types of Petri nets, LNCS 88, 14(b), 1979.

[Ichbiah et al 79] Ichbiah. J., et al Rational for the design of the Ada programming language, SIGPLAN Notices 14(b), 1979.

[Keller 75] Keller, R., A fundamental theorem of asynchronous parallel computation, LNCS 25, 1975.

[Landin 63] Landin, P.J The mathematical evaluation of expressions Comp.J. 6(4), 1963.

[Li 82] Li, W., An operational semantics for Ada multitasking and exception handling, Proceedings of AdaTEC conference, Washington, 1982.

[Lovengreen and Bjórner 80] Lovengreen, H.H., Bjorner, D., On a formal model of the tasking concept in Ada, SIGPLAN Notices, Vol 15, 11, Nov, 1980.

[Luckham and Polak 80] Luckham, D., Polak, W., Ada exception handling: an axiomatic approach, ACM TOPLAS Vol. 2, No. 2, April, 1980.

[Luckham et al 81] Luckham, D., et al, ADAM - An Ada based language for multi-processing, Stanford University Report.No.STAN-CS-81-867, 1981.

[McCarthy 63] McCarthy, J. Towards a mathematical science of computation, Information Processing 1962, Proc.IFIP Congress, 1962, North-Holland, Amsterdam, 1963.

[McCarthy and Painter 67] McCarthy, J. and Painter, J., Correctness of a compiler for arithmetic expressions, Mathematical Aspects of Computing Science, Proceedings of Symposia in Applied Mathematics, Vol.19, 1967.

[Milner 80] Milner, R., A calculus of communicating system, LNCS 92, 1980.

[Milner 82] Milner, R., Calculi for synchrony and asynchrony, CSR-104-82, The University of Edinburgh, 1982.

[Morris 73] Morris, F.L., Advice on structuring compilers and prove them correct Proceedings ACM Symposium on principles of Programming Languages, Boston, 1973.

[Plotkin 80] Plotkin, G., Dijkstra's predicate transformer and Smyth's powerdomain, LNCS 86, 1980.

[Plotkin 81] Plotkin, G., A structural approach to operational semantics, LectureNotes, Aarhus University, 1981.

[Plotkin 82] Plotkin, G., An operational semantics for CSP, Proceedings of IFIP working conference, 1982.

[Plotkin 82] Plotkin, G., Powerdomain for countable non-determinism Internal report, The University of Edinburgh, 1982.

[Shoenfield 67] Shoenfield, J.R., Mathematical logic, Addison-Wesley 1967.

[Stevenson 80] Stevenson, D.R., Algorithms for translating Ada multitasking, SIGPLAN Notices, Vol. 15, Number 11, Nov, 1980.

[Stoy 77] Stoy. J., Denotational semantics: the Scott-Strachey approach to programming language theory, MIT Press, 1977.

[Tennent 81] Tennent, R.D., Principles of programming languages, Prentice-Hall, 1981.

[DoD 80] United States Department of Defence, Reference manual for

the AdaProgramming Language, July 1980.

[Wegner 72] Wegner, P., The Vienna definition language, Computing Surveys Vol. 4 1972.

[Welsh and Lister 81] Welsh, J., Lister, A., A comparative study of task Communication in Ada, Software-Practice and Experience, Vol. 11, 1981.

[Zhou and Hoare 81] Zhou, C.C., Hoare, C.A.R., Partial correctness of communication protocols, Technical Monograph PRG-20, Computing Laboratory, Oxford University. 1981.

Appendix 1

In this appendix we prove lemmas 6.4, 6.5 and 6.6. First of all we need the following lemmas. In the following lemmas and proofs we use t to range over terms, λ and ρ to range over transition actions and u, v and w to range over the sequences of transition actions in CCS. We will follow the notational remarks given in section 5.2 concerning the congruence relation ~ on the transition systems.

Lemma 1.1

a. If $t_1 \xrightarrow{\lambda} t_1'$ and $\delta, \sigma \neq \lambda$ then t_1 before $t_2 \xrightarrow{\lambda} t_1'$ before t_2.

b. done before $t_2 \xrightarrow{\tau} t_2$.

c. fail before $t_2 \xrightarrow{\tau}$ fail.

Proof. For clause (a) since $\delta, \sigma \neq \lambda$ according to the definition we have

$t_1[\pi] \xrightarrow{\lambda} t_1'[\pi]$ and $\gamma_1, \gamma_2 \neq \lambda$, so

$t_1[\pi] \mid (\gamma_1 x.t_2 + \gamma_2 y.\underline{fail}) \xrightarrow{\lambda} t_1'[\pi] \mid (\gamma_1 x.t_2 + \gamma_2 y.\underline{fail})$

and so

t_1 before $t_2 \xrightarrow{\lambda} t_1'$ before t_2.

clauses (b) and (c) are directly from the definition. □

Lemma 1.2

Let t_1 and t_2 be well-formed terms. If t_1 before $t_2 \xrightarrow{\lambda} t$ then one of the following must hold.

a. $t = t_1'$ before t_2 and $t_1 \xrightarrow{\lambda} t_1'$ for some t_1'.

b. $t = t_2$ and $t_1 \sim$ done and $\lambda = \tau$.

c. $t = \underline{fail}$ and $t_1 \sim \underline{fail}$ and $\lambda = \tau$.

Proof. We have

$$t_1 \ \underline{before}\ t_2 = (t_1[\pi] \mid (\gamma_1 x.t_2 + \gamma_2 y.\underline{fail})) \setminus \{\gamma_1, \gamma_2\}$$

where x is not in t_2 and $\pi = \{\gamma_1/\sigma, \gamma_2/\delta\}$

Since $t_1 \ \underline{before}\ t_2 \xrightarrow{\lambda} t$ we have that

$$(t_1[\pi] \mid \gamma_1 x.t_2 + \gamma_2 y.\underline{fail}) \xrightarrow{\lambda} t'$$

and $\gamma_1, \gamma_2 \notin \lambda$ and $t = t' \setminus \{\gamma_1, \gamma_2\}$. Now there are three cases

case 1 $t_1[\pi] \xrightarrow{\lambda} t''$ and $\gamma_1, \gamma_1 \notin \lambda$ and $t' = (t'' \mid (\gamma_1 x.t_2 + \gamma_2 y.\underline{fail}))$. Here we have $t_1 \xrightarrow{\lambda} t_1'$ and $\gamma_1, \gamma_2, \sigma, \delta \notin \lambda$ and $t'' = t_1'[\pi]$. So concluding, we have $t_1 \xrightarrow{\lambda} t_1'$ and

$$\begin{aligned} t &= t' \setminus \{\gamma_1, \gamma_2\} \\ &= (t'' \mid (\gamma_1 x.t_2 + \gamma_2 y.\underline{fail}) \setminus \{\gamma_1, \gamma_2\} \\ &= (t_1'[\pi] \mid (\gamma_1 x.t_2 + \gamma_2 y.\underline{fail}) \setminus \{\gamma_1, \gamma_2\} \\ &= t_1' \ \underline{before}\ t_2 \end{aligned}$$

and case (a) holds.

case 2 $t_1[\pi] \xrightarrow{\gamma_1 ! v} t_1'$ and $t' = (t_1' \mid t_2) \setminus \{\gamma_1, \gamma_2\}$ and $\lambda = \tau$.

Here we have $t_1 \xrightarrow{\sigma ! v} t_1'$ and $t_1' = t_1''[\pi]$. But now, as t_1 is well-formed we have $v = 0$, $t_1'' \sim Nil$ and $t_1 \sim \underline{done}$ and so in conclusion $\lambda = \tau$ and

$$t = t' \setminus \{\gamma_1, \gamma_2\} = (t_1''[\pi] \mid t_2) \setminus \{\gamma_1, \gamma_2\} \sim (Nil[\pi] \mid t_2) \setminus \{\gamma_1, \gamma_2\} \sim t_2$$

and $t_1 \sim \underline{done}$ and so case (b) holds.

case 3 $t_1[\pi] \xrightarrow{\gamma_2 ! v} t_1'$ and $t' = (t_1' \mid \underline{fail}) \setminus \{\gamma_1, \gamma_2\}$ and $\lambda = \tau$. As in case 2 we see that since t_1 is well-formed $t_1 \sim \underline{fail}$ and $t_1' \sim Nil$ so $t = (t_1' \mid \underline{fail}) \setminus \{\gamma_1, \gamma_2\} \sim \underline{fail}$ and so case (c) holds. □

Lemma 1.3

a. If $t_1 \xrightarrow{\lambda} t_1'$ and $\delta, \sigma \notin \lambda$ then $t_1 \underline{\text{par}}\ t_2 \xrightarrow{\lambda} t_1' \underline{\text{par}}\ t_2$.

b. If $t_2 \xrightarrow{\lambda} t_2'$ and $\delta, \sigma \notin \lambda$ then $t_1 \underline{\text{par}}\ t_2 \xrightarrow{\lambda} t_1 \underline{\text{par}}\ t_2'$.

c. If $t_1 \xrightarrow{\lambda} t_1'$ and $t_2 \xrightarrow{\bar{\lambda}} t_2'$ and $\delta, \sigma, \tau \notin \lambda$ Then
$t_1 \underline{\text{par}}\ t_2 \xrightarrow{\tau'} t_1' \underline{\text{par}}\ t_2'$

d. $(\underline{\text{done}}\ \underline{\text{par}}\ t_2) \xrightarrow{\tau} (t_2\ \underline{\text{before}}\tau'.\underline{\text{done}})$

e. $(t_1\ \underline{\text{par}}\ \underline{\text{done}}) \xrightarrow{\tau} (t_1\ \underline{\text{before}}\tau'.\underline{\text{done}})$

f. $(\underline{\text{fail}}\ \underline{\text{par}}\ t_2) \xrightarrow{\tau} (t_2\ \underline{\text{before}}\tau'.\underline{\text{fail}})$

g. $(t_1\ \underline{\text{par}}\ \underline{\text{fail}}) \xrightarrow{\tau} (t_1\ \underline{\text{before}}\tau'.\underline{\text{fail}})$.

Proof. Let us examine cases (c), (d) and (f).

case c. Since $\delta, \sigma \notin \lambda$ we have

$$t_1[\pi] \xrightarrow{\lambda} t_1'[\pi] \quad \text{and} \quad t_2[\pi] \xrightarrow{\bar{\lambda}} t_2'[\pi]$$

By the composition rule we have

$$(t_1[\pi] \mid t_2[\pi] \mid \text{parx}) \xrightarrow{\tau} (t_1' \mid t_2' \mid \text{parx})$$

and so

$$t_1\ \underline{\text{par}}\ t_2 \xrightarrow{\tau} t_1'\ \underline{\text{par}}\ t_2'$$

and case (c) holds.

case d. Since $\underline{\text{done}}[\pi] \xrightarrow{\gamma_1!0} \text{Nil}$ by the composition rule we have

$$(\underline{\text{done}}[\pi] \mid t_2[\pi] \mid \text{parx}) \xrightarrow{\tau} (\text{Nil} \mid t_2[\pi] \mid (\gamma_1 x.\tau'.\underline{\text{done}} + \gamma_2 y.\underline{\text{fail}}))$$

and so

$$\begin{aligned}(\underline{\text{done}}\ \underline{\text{par}}\ t_2) &= (\underline{\text{done}}[\pi] \mid t_2[\pi] \mid \text{parx}) \setminus \{\gamma_1, \gamma_2\} \\ &\xrightarrow{\tau} (t_2[\pi] \mid (\gamma_1 x.\tau'.\underline{\text{done}} + \gamma_2 y.\underline{\text{fail}})) \setminus \{\gamma_1, \gamma_2\} \\ &= t_2\ \underline{\text{before}}\ \tau'.\underline{\text{done}}\end{aligned}$$

and case (d) holds.

case f. Since $\underline{fail}[\pi] \xrightarrow{\gamma_2!0} \text{Nil}$ by the composition rule we have

$$(\underline{fail}[\pi] | t_2[\pi] | parx) \xrightarrow{\tau} (\text{Nil} | t_2[\pi] | (\gamma_1 x.\tau'.\underline{fail} + \gamma_2 y.\underline{fail}))$$

and so

$$\begin{aligned}(\underline{fail}\ \underline{par}\ t_2) &= (\underline{fail}[\pi] | t_2[\pi] | parx) \setminus \{\gamma_1, \gamma_2\} \\ &\xrightarrow{\tau} (t_2[\pi] | (\gamma_1 x.\tau'.\underline{fail} + \gamma_2 y.\underline{fail})) \setminus \{\gamma_1, \gamma_2\} \\ &= t_2 \underline{before}\ \tau'.\underline{fail}\end{aligned}$$

and so case (f) holds. □

Lemma 1.4

Suppose t_1 and t_2 are well-formed. If $t_1\ \underline{par}\ t_2 \xrightarrow{\lambda} t$ then one of the following holds:

a. For some t_1' $t=(t_1'\ \underline{par}\ t_2)$ and $t_1 \xrightarrow{\lambda} t_1'$ and δ, $\sigma \notin \lambda$.

b. For some t_2' $t=(t_1\ \underline{par}\ t_2')$ and $t_2 \xrightarrow{\lambda} t_2'$ and δ, $\sigma \notin \lambda$.

c. For some t_1' and t_2' and $\rho \neq \tau$. $t=(t_1'\ \underline{par}\ t_2')$ and $t_1 \xrightarrow{\rho} t_1'$ and $t_2 \xrightarrow{\bar{\rho}} t_1'$ and $\lambda=\tau$.

d. $t=(t_2\ \underline{before}\ \tau'.\underline{done})$ and $t_1=\underline{done}$ and $\lambda=\tau$.

e. $t=(t_1\ \underline{before}\ \tau'.\underline{done})$ and $t_2=\underline{done}$ and $\lambda=\tau$.

f. $t=(t_2\ \underline{before}\ \tau'.\underline{fail})$ and $t_1=\underline{fail}$ and $\lambda=\tau$.

g. $t=(t_1\ \underline{before}\ \tau'.\underline{fail})$ and $t_2=\underline{fail}$ and $\lambda=\tau$.

Proof. We must have for some t' that $t=t' \setminus \{\gamma_1, \gamma_2\}$ and

$$(t_1[\pi] | t_2[\pi] | parx) \xrightarrow{\lambda} t' \text{ and } \gamma_1,\ \gamma_2 \notin \lambda$$

and now according to the composition rule together with the fact that only γ_1 and γ_2 moves are available to parx we have the following cases:

case 1 Moves of t_1 alone. Here we have $t_1[\pi] \xrightarrow{\lambda} t_1''$ and $t'=t_1''|t_2[\pi]|parx$. Then as $\gamma_1, \gamma_2 \notin \lambda$ we have $t_1 \xrightarrow{\lambda} t_1'$ for some t_1' and $\delta, \sigma \notin \lambda$ and $t_1''=t_1'[\pi]$ so

$$\begin{aligned} t=t'\backslash\{\gamma_1,\gamma_2\} &= (t_1''|t_2[\pi]|parx)\backslash\{\gamma_1,\gamma_2\} \\ &= (t_1'[\pi]|t_2[\pi]|parx)\backslash\{\gamma_1,\gamma_2\} \\ &= t_1' \underline{par}\ t_2 \end{aligned}$$

and case (a) holds.

case 2 Moves of t_2 alone. The proof is similar to case 1.

case 3 Interaction between t_1 and t_2.
Here $\lambda=\tau$ and for some $\rho \neq \tau$ we have

$$t_1[\pi] \xrightarrow{\rho'} t_1'' \quad \text{and} \quad t_2 \xrightarrow{\bar{\rho}'} t_2'' \quad \text{and} \quad t'=(t_1''|t_2''|parx)$$

Then for some ρ_1 we have

$$t_1 \xrightarrow{\rho_1} t_1' \quad \text{and} \quad \rho'=\rho_1[\pi] \quad \text{and} \quad t_1''=t_1'[\pi],$$

and for some ρ_2 we have

$$t_2 \xrightarrow{\rho_2} t_2' \quad \text{and} \quad \bar{\rho}'=\rho_2[\pi] \quad \text{and} \quad t_2''=t_2'[\pi].$$

We cannot have δ or $\sigma \varepsilon \rho_1$. For otherwise $\gamma_1, \gamma_2 \varepsilon \rho'$ and so as $\bar{\rho}'=\rho_2[\pi]$ we have δ or $\sigma \varepsilon \rho_2$. So σ or $\delta \varepsilon \rho_1$ and σ or $\delta \varepsilon \rho_2$ and so clearly either both $\sigma \varepsilon \rho_1$ and ρ_2 or δ in ρ_1 and ρ_2. But then as both t_1 and t_2 are well-formed either $\rho_1=\rho_2=\sigma!0$ or $\rho_1=\rho_2=\delta!0$ and in both cases we have a contradiction; for example in the first case we have $\rho'=\rho_1[\pi]=\gamma_1!0$ and $\bar{\rho}'=\rho_2[\pi]=\gamma_2!0$ which contradicts the definition of complementary actions. So $\delta, \sigma \notin \rho_1$ and $\delta, \sigma \notin \rho_2$. Therefore $\rho'=\rho_1[\pi]=\rho_1$ and $\bar{\rho}'=\rho_2[\pi]=\rho_2$ and we see that $\lambda=\tau$, $t_1 \xrightarrow{\rho'} t_1'$ and $t_2 \xrightarrow{\bar{\rho}'} t_2'$ and

$$\begin{aligned} t=t'\backslash\{\gamma_1,\gamma_2\} &= (t_1''|t_2''|parx)\backslash\{\gamma_1,\gamma_2\} \\ &= t_1'[\pi]|t_2'[\pi]|parx)\backslash\{\gamma_1,\gamma_2\} \\ &= t_1' \underline{par}\ t_2' \end{aligned}$$

and case (c) holds.

case 4 Interaction between $t_1[\pi]$ and parx.

Here $\lambda=\tau$ and we have

$$t_1[\pi] \xrightarrow{\rho} t_1'' \quad \text{and} \quad \text{parx} \xrightarrow{\bar{\rho}} u$$

for some $\rho \neq \tau$ and t_1'' and u and $t'=(t_1''|t_2|u)$

Looking at parx we see that either

$\bar{\rho}=\gamma_1?v$ (for some v) and $u=\gamma_1 x.\tau'.\underline{done}+\gamma_2 y.\underline{fail}$

or

$\bar{\rho}=\gamma_2?v$ (for some v) and $u=\gamma_1 x.\tau'.\underline{fail}+\gamma_2 y.\underline{fail}$

Thus we have two subcases:

case 4.1 $\bar{\rho}=\gamma_1?v$. Since γ_1 does not occur in t_1 we must have

$$t_1 \xrightarrow{\sigma!0} t_1' \text{ and } t_1''=t_1'[\pi] \text{ for some } t_1'.$$

But as t_1 is well-formed we have $v=0$ and $t_1' \sim Nil$ and $t_1=\underline{done}$. So $t_1=\underline{done}$ and $\lambda=\tau$ and

$$\begin{aligned} t=t'\backslash\{\gamma_1,\gamma_2\} &= (t_1''|t_2[\pi]|u)\backslash\{\gamma_1,\gamma_2\} \\ &= (Nil[\pi]|t_2[\pi]|(\gamma_1 x.\tau'.\underline{done}+\gamma_2 y.\underline{fail}))\backslash\{\gamma_1,\gamma_2\} \\ &\sim t_2 \ \underline{\text{before}} \ \tau'.\underline{done}. \end{aligned}$$

case 4.2 $\bar{\rho}=\gamma_2?v$. The proof is similar to case 4.1.

case 5 Interaction between $t_2[\pi]$ and parx. The proof is similar to case 4. □

We now study "many step" generation of the lemmas 1.1 and 1.2.

Lemma 1.5

Suppose t_1 and t_2 are well-formed. Then $t_1 \ \underline{\text{before}} \ t_2 \xrightarrow{u} t$ with

δ, $\sigma \not\downarrow u$ iff one of the following holds:

a. $t = t_1'$ before t_2 and $t_1 \xrightarrow{u} t_1'$.

b. $t = t_2'$ and $t_1 \xrightarrow{u_1}$ done and $t_2 \xrightarrow{u_2} t_2'$ and $u = u_1 \tau u_2$.

c. $t =$ fail and $t_1 \xrightarrow{u_1}$ fail and $u = u_1 \tau$.

Proof. The "only if part" is proved by induction on $|u|$ the length of u.

case 1 $|u| = 0$. Here $u = \emptyset$ and (a) holds with $t_1' = t_1$.

case 2 $u = \lambda u'$. Here t_1 before $t_2 \xrightarrow{\lambda} t' \xrightarrow{u'} t$. There are the following subcases according to lemma 1.2.

case 2a. For some t_1'', $t' = t_1''$ before t_2 and $t_1 \xrightarrow{\lambda} t_1''$. Now apply the induction hypothesis to t_1'' before $t_2 \xrightarrow{u'} t$.

a. $t = t_1'$ before t_2 and $t_1'' \xrightarrow{u'} t_1'$. Here $t_1 \xrightarrow{\lambda} t_1'' \xrightarrow{u'} t_1'$ and case (a) holds.

b. $t = t_2'$ and $t_1'' \xrightarrow{u_1}$ done and $t_2 \xrightarrow{u_2} t_2'$ and $u' = u_1 \tau u_2$. Here $t_1 \xrightarrow{\lambda} t_1'' \xrightarrow{u_1}$ done and $t = t_2'$ and $t_2 \xrightarrow{u_2} t_2'$ and $u = (\lambda u_1) \tau u_2$ and case (b) holds.

c. $t =$ fail and $t_1'' \xrightarrow{u'}$ fail. Here $t =$ fail and $t \xrightarrow{\lambda} t_1'' \xrightarrow{u'}$ fail and so as $u = \lambda u'$ we have $t_1 \xrightarrow{u}$ fail and case (c) holds.

case 2b Here $\lambda = \tau$ and $t' = t_2$ and $t_1 \sim$ done. Now we see that $t_1 \xrightarrow{\emptyset}$ done and $t_2 \xrightarrow{u'} t$ and $u = \tau u' = \emptyset \tau u'$ and case (b) holds.

case 2c Here $\lambda = \tau$ and $t' =$ fail and $t_1 =$ fail. So fail $\xrightarrow{u'} t$. But $\delta, \sigma \not\downarrow u$ and so they are not in u' and so $u' = \emptyset$. Now we see that $t =$ fail and also $t_1 \xrightarrow{\emptyset}$ fail and $u = \emptyset\tau$ and so case (c) holds.

We now consider the "if" part.

a. If clause (a) holds then by lemma 1.1 the result is true.

b. If clause (b) holds then we have

$$t_1 \ \underline{\text{before}} \ t_2 \xrightarrow{u_1} \underline{\text{done}} \ \underline{\text{before}} \ t_2 \quad \text{(by (a))}$$
$$\xrightarrow{\tau} t_2$$
$$\xrightarrow{u_2} t_2' = t$$

c. If clause (c) holds then we have

$$t_1 \ \underline{\text{before}} \ t_2 \xrightarrow{u_1} \underline{\text{fail}} \ \underline{\text{before}} \ t_2 \quad \text{(by (a))}$$
$$\xrightarrow{\tau} \underline{\text{fail}} = t. \quad \square$$

Similarly, we can prove the following "many step" generation of lemmas 1.3 and 1.4.

Lemma 1.6

Suppose t_1 and t_2 are well-formed. Then $t_1 \ \underline{\text{par}} \ t_2 \xrightarrow{w} t$ with $\delta, \sigma \not\downarrow u$ iff one of the following holds:

a. $t = t_1' \ \underline{\text{par}} \ t_2'$ and for some u and v
$t_1 \xrightarrow{u} t_1'$ and $t_2 \xrightarrow{v} t_2'$ and Merge(u,v,w)

b. $t = t_2'$ and for some w_1, w_2 and t_2'' we have $w = w_1 \tau w_2$ and

$t_1 \ \underline{\text{par}} \ t_2 \xrightarrow{w_1} \underline{\text{done}} \ \underline{\text{par}} \ t_2''$ and
$t_2'' \ \underline{\text{before}} \ \tau'.\underline{\text{done}} \xrightarrow{w_2} t_2'$
and for some u and v Merge(u,v,w_1) and
$t_1 \xrightarrow{u} \underline{\text{done}}$ and $t_2 \xrightarrow{v} t_2''$.

c. $t = t_1'$ and for some w_1, w_2 and t_1'' we have $w = w_1 \tau w_2$ and

t_1 par $t_2 \xrightarrow{w_1} t_1''$ par done and

t_1'' before τ'.done $\xrightarrow{w_2} t_1'$

and for some u and v Merge(u,v,w_1) and

$t_1 \xrightarrow{u} t_1''$ and $t_2 \xrightarrow{v}$ done.

d. $t=t_2'$ and for some w_1, w_2 and t_2'' we have $w=w_1\tau w_2$ and

t_1 par $t_2 \xrightarrow{w_1}$ fail par t_2'' and

t_2'' before τ'.fail $\xrightarrow{w_2} t_2'$

and for some u and v Merge(u,v,w_1) and

$t_1 \xrightarrow{u}$ fail and $t_2 \xrightarrow{v} t_2'$.

e. $t=t_1'$ and for some w_1, w_2 and t_1'' we have $w=w_1\tau w_2$ and

t_1 par $t_2 \xrightarrow{w_1} t_1''$ par fail and

t_1'' before τ'.fail $\xrightarrow{w_2} t_1'$

and for some u and v Merge(u,v,w_1) and

$t_1 \xrightarrow{u} t_1''$ and $t_2 \xrightarrow{v}$ fail. □

We now prove lemma 6.4.

Lemma 6.4

Let t be well-formed and $w \varepsilon \Lambda_t^*$. Then $m(t,s) \xrightarrow{w} x$ iff one of the following holds:

a. For some t', u and s' we have $t \xrightarrow{u} t'$ and $s'=su$ and $w=\bar{u}$ and $x=m(t',s')$ and neither $\sigma \varepsilon u$ or $\delta \varepsilon u$.

b. For some u, u'and s' and we have σ, $\delta \not\varepsilon u$ and

$t \xrightarrow{u}$ fail and $w=\bar{u}u'$ and $s'=su$ and

$m(t,s) \xrightarrow{\bar{u}} m(\text{fail},s') \xrightarrow{u'} x$.

(and of course $u' \varepsilon \{\tau\}^*$).

Proof. The proof is by induction on the length of w.

If $|w|=0$ then $w=\emptyset$ so take $u=\emptyset$, $t'=t$, $s'=s$ and $x=m(t,s)$ and case (a) holds.

Let $w=\lambda w'$. Noticing that M_s and R cannot communicate with each other, so there are three cases need to analyse:

case 1 There is no communication between t and M_s or t and R. Let $t \xrightarrow{\lambda} t''$ and $m(t'',s) \xrightarrow{w'} x$.

By the induction hypothesis on w' we have further two subcases:

case 1.1 For some t', u' and s' we have $t'' \xrightarrow{u'} t'$ and $s'=su'$ and $\bar{u}'=w'$ and $x=m(t',s')$ and $\delta, \sigma \notin u'$. So take $u=\lambda u'$ we have $t \xrightarrow{\lambda} t'' \xrightarrow{u'} t'$ and $\bar{u}=\overline{\lambda u}'=\lambda\bar{u}'=\lambda w'=w$ and and $su=s\lambda u'=su'$ $(\alpha_i, \beta_i \notin \lambda)$ and case (a) holds.

case 1.2 For some u_1, u_2 and s' we have $\delta, \sigma \notin u_1$ and $t'' \xrightarrow{u_1} \underline{fail}$ and $m(\underline{fail},s) \xrightarrow{u_2} x$ and $w'=\bar{u}_1 u_2$.
So take $u=\lambda u_1$ and $u'=u_2$ we have $t \xrightarrow{\lambda} t'' \xrightarrow{u_1} \underline{fail}$ and $w'=\bar{u}_1 u_2$ and $m(t,s) \xrightarrow{\lambda} m(t'',s) \xrightarrow{\bar{u}_1} m(\underline{fail},s') \xrightarrow{u'} x$. and $\bar{u}.u'=\overline{\lambda u}_1 u_2=\lambda\bar{u}_1 u_2=\lambda w'=w$ and case (b) holds.

case 2 There is a communication between t and M_s.

Then $\lambda=\tau$ and there are two subcases:

case 2.1 $t \xrightarrow{\alpha_i!v} t''$ and $M_s \xrightarrow{\alpha_i?v} M_{s[v/x_i]}$ and $m(t'',s[v/x_i]) \xrightarrow{w'} x$.

Applying the induction hypothesis we have further two subcases to consider.

case 2.1.1 For some u_1 and s' we have $t'' \xrightarrow{u_1} t'$ and $\bar{u}_1=w'$ and $s'=s[v/x_i]u_1$ and $x=m(t',s')$. So take $u=\alpha_i!v.u_1$ then we have

$t \xrightarrow{\alpha_i!v} t'' \xrightarrow{u_1} t'$ and

$\bar{u}=\overline{\alpha_i!v}.u_1=\tau\bar{u}_1=\tau w'=w$ and

$su=s(\alpha_i!v.u_1)=s[v/x_i]u_1=s'$ and case (a) holds.

case 2.1.2 For some u_1, u_2 and s' we have

$t'' \xrightarrow{u_1} \underline{fail}$ and $m(\underline{fail}, s[v/x_i]) \xrightarrow{u_2} x$

and $w'=\bar{u}_1 u_2$ and $s'=s[v/x_i]u_1$ and $u_2 \varepsilon \{\tau\}^+$

Take $u=\alpha_i!v.u_1$ and $u'=u_2$ then we have

$t \xrightarrow{\alpha_i!v} t'' \xrightarrow{u'} \underline{fail}$ and

$\bar{u}u'=\overline{\alpha_i!v.u_1}u_2=\tau\bar{u}_1u_2=\tau w'=w$

$su=s(\alpha_i!v.u_1)=s[v/x_i]u_1=s'$

and case (b) holds.

case 2.2 $t \xrightarrow{\beta_i?v} t''$ and $M_s \xrightarrow{\beta_i!v} M_s$ and $m(t'',s) \xrightarrow{w'} x$. The proof is similar to case 2.1.

case 3 There is a communication between t and R.
Since t is well-formed $t=\underline{fail}$ and then the computation from $m(\underline{fail},s)$ is deterministic and the case (b) holds obviously. □

Lemma 6.5

Let t_1 and t_2 be well-formed with no free occurrences of γ_i. Then $m(t_1 \underline{\text{ before }} t_2, s) \xrightarrow{w} x$ iff one of the following holds:

a. For some t_1' and s' we have:
$m(t_1,s) \xrightarrow{w} m(t_1',s')$ and $x=m(t_1' \underline{\text{ before }} t_2, s')$

b. For some t', s' and u, u' we have:
$m(t_1,s) \xrightarrow{u} m(\underline{done},s')$ and $m(t_2,s') \xrightarrow{u'} x$ and $w=u\tau u'$.

c. For some t', s' and u, u' we have:

$m(t_1,s) \xrightarrow{u} m(\underline{fail},s') \xrightarrow{u'} x$ and $w=uu'$.

Proof. Consider the "if" part. There are three cases.

a. Clause (a) holds. Noticing $m(t_1,s) \xrightarrow{w} m(t_1',s')$ Then by lemma 6.4 for some t_1'', u and s" we have $t_1 \xrightarrow{u} t_1''$ and $\bar{u}=w$ and $su=s''$ and δ, $\sigma \not\# u$ and $m(t_1'',s'')=m(t_1',s')$. so $t_1'=t_1''$ and $s'=s''$.

So $t_1 \xrightarrow{u} t_1'$ and $su=s'$ with δ, $\sigma \not\# u$. Now applying lemma 1.5 we see that

$$t_1 \ \underline{before}\ t_2 \xrightarrow{u} t_1' \ \underline{before}\ t_2$$

and so by lemma 6.4 since $\bar{u}=w$ and $su=s'$

$$m(t_1 \ \underline{before}\ t_2,s) \xrightarrow{w} m(t_1' \ \underline{before}\ t_2,s').$$

b. Clause (b) holds. We have

$$\begin{aligned} m(t_1 \ \underline{before}\ t_2,s) &\xrightarrow{u} m(\underline{done}\ \underline{before}\ t_2,s') \quad \text{(by (a))} \\ &\xrightarrow{\tau} (t_2,s') \\ &\xrightarrow{u'} x. \end{aligned}$$

c. Clause (c) holds. We have

$$\begin{aligned} m(t_1 \ \underline{before}\ t_2,s) &\xrightarrow{u} m(\underline{fail}\ \underline{before}\ t_2,s') \quad \text{(by (a))} \\ &\xrightarrow{\tau} m(\underline{fail},s') \\ &\xrightarrow{u'} x. \end{aligned}$$

We now consider the "only if" part. Suppose $m(t_1 \ \underline{before}\ t_2,s) \xrightarrow{w} x$. We must examine two cases:

case 1 Normal termination. Here we find u, s', t' with δ, $\sigma \not\# u$, $su=s'$ and $\bar{u}=w$ such that

$(t_1 \underline{\text{before}}\ t_2) \xrightarrow{u} t'$ and $x=m(t',s')$

now according to lemma 1.5 we have a further three cases to examine:

case 1.1 There is a t_1' with $t_1 \xrightarrow{u} t_1'$ and $t'=t_1' \underline{\text{before}}\ t_2$. But now we have

$$m(t_1,s) \xrightarrow{w} m(t_1',s') \quad \text{and} \quad x=m(t',s')=m(t_1' \underline{\text{before}}\ t_2,s')$$

and we see that clause (a) holds.

case 1.2 t_1 terminates normally. Here there are u_1 and u_2 such that

$$u=u_1\tau u_2 \quad \text{and} \quad t_1 \xrightarrow{u_1} \underline{\text{done}} \quad \text{and} \quad t_2 \xrightarrow{u_2} t'$$

Now take $s''=su_1$ and $s'=s''u_2$, then by lemma 6.4 we have

$$m(t_1,s) \xrightarrow{\bar{u}_1} m(\underline{\text{done}},s'') \quad \text{and}$$
$$m(t_2,s'') \xrightarrow{\bar{u}_2} m(t',s')=x$$

Now note that $\bar{u}_1\tau\bar{u}_2=w$ and we have the case (b) holds.

case 1.3 t_1 aborts. Here there is u_1 such that $u=u_1\tau$, $t'=\underline{\text{fail}}$ and $t_1 \xrightarrow{u_1} \underline{\text{fail}}$. Note $s'=su_1$. Then we see that $m(t_1,s) \xrightarrow{\bar{u}_1} m(\underline{\text{fail}},s')$. So putting $u'=\emptyset$ we are in case (c) since $x=m(\underline{\text{fail}},s')$ and $w=\bar{u}_1\tau$.

case 2 $t_1 \underline{\text{before}}\ t_2$ aborts. Here for some u, u' and s' we have

$$(t_1 \underline{\text{before}}\ t_2) \xrightarrow{u_1} \underline{\text{fail}} \quad \text{and} \quad (M_s,|\bar{R})\backslash M \xrightarrow{u'} x$$

where $\bar{R}=\alpha_1(0.\alpha_2(0,\ldots\alpha_n(0,\underline{\text{fail}})\ldots)$ and $w=\bar{u}_1\tau u'$ and $s'=su_1$ and δ, $\sigma \not\leq u$. By lemma 1.5 There are two cases to consider:

case 2.1 t_1 terminates normally and t_2 aborts. Here $t_1 \xrightarrow{v_1} \underline{done}$ and $t_2 \xrightarrow{v_2} \underline{fail}$ and $u_1 = v_1 \tau v_2$. Set $s'' = sv_1$ (so $s' = s''v_2$) so

$$m(t_1, s) \xrightarrow{\bar{v}_1} m(\underline{done}, s'') \quad \text{and}$$
$$m(t_2, s'') \xrightarrow{\bar{v}_2} m(fail, s') \xrightarrow{\tau} (M_s, |R^-) \backslash M \xrightarrow{u'} x$$

But now note that $w = \bar{u}_1 \tau u' = \bar{v}_1 \tau \bar{v}_2 \tau u' = \bar{v}_1 \tau (\bar{v}_2 \tau u')$ and we are in case (b).

case 2.2 t_1 aborts. Here $t_1 \xrightarrow{v_1} \underline{fail}$ and $s' = sv_1$ and

$$m(t_1, s) \xrightarrow{\bar{v}_1} m(\underline{fail}, s') \xrightarrow{\tau} (M_s, |R^-) \backslash M \xrightarrow{u'} x$$

and we have $w = \bar{v}_1 \tau \tau u' = \bar{v}_1 \tau (\tau u')$ and case (c) holds. □

Similarly, we can prove lemma 6.6.

Lemma 6.6

Let t_1 and t_2 be well-formed with no free occurrences of γ_i. and $FL(t_1) \cap FL(t_2) \cap M_\alpha = \emptyset$. Then $m(t_1 \ \underline{par} \ t_2, s) \xrightarrow{w} x$ iff one of the following holds:

a. For some u, v, and w' we have: $\bar{w}' = w$ and

$x = m(t_1 \ \underline{par} \ t_2, sw')$ and $Merge(u, v, w')$ and $sw' = su \Theta sv$ and
$m(t_1, s) \xrightarrow{\bar{u}} m(t_1', su)$ and $m(t_2, s) \xrightarrow{\bar{v}} m(t_2', sv)$.

b. For some w', w'' and t_2' we have $w = \bar{w}' \tau w''$ and:

$m(t_1 \ \underline{par} \ t_2, s) \xrightarrow{\bar{w}'} m(\underline{done} \ \underline{par} \ t_2', sw')$ and
$m(t_2' \ \underline{before} \ \underline{done}, sw') \xrightarrow{w''} x$

and for some u, v we have Merge(u,v,w') and sw'=suθsv and

$m(t_1,s) \xrightarrow{\bar{u}} m(\underline{done},su)$ and $m(t_2,s) \xrightarrow{\bar{v}} m(t_2',sv)$

c. For some w', w" and t_1', s' we have $w=\bar{w}'\tau w''$ and

$m(t_1 \ \underline{par}\ t_2,s) \xrightarrow{\bar{w}'} m(t_1' \ \underline{par}\ \underline{done},sw')$ and

$m(t_1' \ \underline{before}\ \underline{done},sw') \xrightarrow{w''} x$

and for some u, v we have Merge(u,v,w') and sw'=suθsv and

$m(t_1,s) \xrightarrow{\bar{u}} m(t_1',su)$ and $m(t_2,s) \xrightarrow{\bar{v}} m(\underline{done},sv)$

d. For some w', w" and t_2' we have $w=\bar{w}'\tau w''$ and

$m(t_1 \ \underline{par}\ t_2,s) \xrightarrow{\bar{w}'} m(\underline{fail} \ \underline{par}\ t_2',sw')$ and

$m(t_2' \ \underline{before}[\![\underline{abort}]\!],sw') \xrightarrow{w''} x$

and for some u, v we have Merge(u,v,w') and sw'=suθsv

and $m(t_1,s) \xrightarrow{\bar{u}} m(\underline{fail},su)$ and $m(t_2,s) \xrightarrow{\bar{v}} m(t_2',sv)$

e. For some w', w" and t_1' we have $w=\bar{w}'\tau w''$ and

$m(t_1 \ \underline{par}\ t_2,s) \xrightarrow{\bar{w}'} m(t_1' \ \underline{par}\ \underline{fail},sw')$ and

$m(t_1' \ \underline{before}[\![\underline{abort}]\!],sw') \xrightarrow{w''} x$

and for some u, v we have Merge(u,v,w') and sw'=suθsv

and $m(t_1,s) \xrightarrow{\bar{u}} m(t_1',su)$ and $m(t_2,s) \xrightarrow{\bar{v}} m(\underline{fail},sv)$