The Fork Calculus

Towards a Logic for Concurrent ML

Klaus Havelund

Ph.D. Thesis

DIKU – Datalogisk Institut
København's Universitet
Danmark

January 1994

Acknowledgements

I would like to thank Kim Guldstrand Larsen for having provided me with invaluable inspiration and for having offered his collaboration on certain parts of this work. I also thank Klaus Grue for his never failing relaxed confidence, and Neil Jones for initially having supported the idea of writing a thesis. I should also mention Don Sannella with which I have had electronic communication via email. He has faithfully answered my letters. Finally I thank Ecole Normale Supérieure in Paris, France, where I have carried out most of the work from September 1991 to January 1994. In particular Patrick and Radhia Cousot have been so kind to involve me in their groups, where I have had many discussions with Eric Goubault and Regis Cridlig in particular.

Contents

Part I

Introduction and Preliminaries

Chapter 1

Introduction

1.1 The Problem

The study of combinations of functional and parallel programming is getting an increasing attention these years. Many of the languages that are designed can be seen as extensions of the lambda calculus with concurrency primitives of various kinds. Typically the concurrency primitives are inspired by CCS [Mil89] and CSP [Hoa85a]. Amongst these are CML [Rep91a, Rep92], Facile [GMP89, PGM90, Pra91], Poly/ML [Mat91], LCS [Ber93], and the two calculi [Nie89] and [NT92b]. However, other approaches are also taken, for example [AMST92].

One of these new languages is CML (Concurrent ML), which is an extension of the functional programming language ML [MTH90] with CCS/CSP-like concurrency primitives. Already the fact that CML is based on ML makes it interesting since ML itself has had quite a success, at least within the academic world. One reason for this success is of course the functional programming paradigm, which seems to ease programming in many situations. To this comes that ML with its module system (and even an assignment statement) appears to be applicable to real sized problems. Finally, ML is a formally well defined language [MTH90]. The extension with concurrency primitives additionally seems to be relatively simple with a formal definition [Rep92, BMT92], and it appears to be quite powerful.

Another branch of research is that of program specification, where the goal is to provide theories for the formal refinement of specifications into programs via sequences of verified-correct development steps. Examples are RAISE [Hav92, BHH92, GHNW89] (a refinement of VDM [BJ82, Jon80]), LOTOS [BB89], and Z [Spi89]. Few of these theories, however, formally relate the specification language to a particular programming language. One theory that does so is EML (Extended ML) [San89, ST90, KST93], which is an extension of ML with specification constructs, basically the predicate calculus. EML allows formal development of modular ML programs which are correct with respect to a specification of their required behaviour.

The idea behind the work in this thesis came from observing the triple: ML, EML and CML. There was one missing: ECML (Extended CML) being an extension of CML with specification constructs. The idea was to combine concepts from predicate logic and modal logic. Our initial attempt to define such a logic quickly lead to problems caused by the concurrency primitives found in CML. They are in fact not directly borrowed from CCS, and the modal logics for CCS could therefore not be directly applied. Let us give a short presentation of CML to illustrate the problems involved.

The model behind CML is that of concurrently executing expressions that communicate by handshake message passing on typed channels. The concurrent aspects of CML, however, differ from CCS in a essential way: in CCS there is a binary parallel operator $|$, and two processes p and q are put in parallel by $p \mid q$. In CML the binary parallel operator has been replaced by a unary `fork` operator, and a process `p` is activated to "run in parallel with the rest of the program" by `fork(p)`.

To be more precise, the concurrency primitives of CML are provided as a set of "functions", from which the following may be derived[1]:

```
val channel  : unit -> '1a chan
val transmit : 'a chan * 'a -> unit
val accept   : 'a chan -> 'a
val fork     : (unit -> 'a) -> unit
```

The function `channel` yields a fresh channel each time it is applied. The function `transmit` sends a value to a channel. The function `accept` reads a value from a channel. Finally, the function `fork` starts a separate evaluation of its argument function; that is: `fork(f)` starts the evaluation of `f()`.

As an example, consider an implementation in CML of a function `calc:real->real` having the following specification:

```
calc(x)=cos(x)+sin(x)
```

The implementation can be given as follows:

[1]There are minor differences in the choice of CML primitives, their types and names, in [Rep92] and [BMT92]. In fact, [BMT92] does not call the defined language CML; but it is stated that the semantics applies to CML. In terms of the primitive features `sync`, `send` and `receive` of [BMT92], which we have chosen to follow, `transmit` and `accept` may be defined as: `transmit(x) = (sync(send(x));())` and `accept(x) = sync(receive(x))`.

```
fun calc(x) =
      let val r1 = channel()
          val r2 = channel()
      in
        fork(fn () => transmit(r1,cos(x)));
        fork(fn () => transmit(r2,sin(x)));
        (accept(r1) + accept(r2))
      end;
```

The function evaluates the expressions `cos(x)` and `sin(x)` in parallel. First, two (local) channels `r1` and `r2` are allocated. Then, the two evaluations are forked; The two forked evaluations send their respective result back on the channels `r1` and `r2`.

Our ultimate goal is to define an appropriate equivalence between CML expressions. Obviously, we have a number of expectations to properties enjoyed by such an equivalence. As an example, consider the following alternative implementation of the `calc` function, we name it `calc'`:

```
fun calc'(x) =
      let val r1 = channel()
          val r2 = channel()
      in
        fork(fn () => transmit(r2,sin(x)));
        fork(fn () => transmit(r1,cos(x)));
        (accept(r1) + accept(r2))
      end;
```

The only difference from `calc` is that we have changed the order of the two `fork` expressions. A natural requirement is that `calc = calc'`: it seems reasonable to require the two sequences of `fork` expressions to be equal since the order of forking should not matter. The result in both cases is two parallel evaluations of the expressions `cos(x)` and `sin(x)` and we really do not care about how this result is obtained. In general, we will not want to observe the particular forking strategy — only the collected behaviour of the result. As an even more radical example, the two implementations given here should both satisfy the original specification `calc(x)=cos(x)+sin(x)`. That is, the equation must hold when substituting the right hand sides of the definitions for `calc(x)`.

Another natural and important requirement to our equivalence is that it is a *congruence* with respect to all CML constructs as this will allow compositional verification. In particular, equivalences between composite expressions should be inferable from equivalences of sub–expressions. As an example, we want the equivalence `calc = calc'` to be inferred from an equivalence between the two `fork`–sequences:

```
fork(fn () => transmit(r1,cos(x)));
fork(fn () => transmit(r2,sin(x)))
```

and

```
fork(fn () => transmit(r2,sin(x)));
fork(fn () => transmit(r1,cos(x)))
```

1.2 The Approach

The issues of equivalence and compositionality on the one hand, and logics on the other, in the presense of forking, can best be studied in isolation from the other issues in CML, so we design a calculus, FC (Fork Calculus), that is close to CCS, but which provides a one-argument fork operator instead of the two-argument parallel-operator found in CCS. Also, the unary prefix–operator of CCS is replaced by a dyadic operator for sequential composition.

FC is thus a projection of CML which focuses on the concurrency aspects. Other existing calculi may also be regarded as projections of CML, although this was not an objective for their design. CCS [Mil89] is another projection focusing on the concurrency aspects. CHOCS [Tho89] is a projection focusing on the higher orderness of CML in the sense that processes can be communicated over channels as "values". The π-calculus [MPW89a, MPW89b] is a projection that focuses on channels, their scope and communication. Finally, the λ-calculus is a projection focusing on the functional subset of CML.

We give a CCS–like, structured operational semantics of FC, in which communication and the forking of processes yield internal transitions. Based on this operational semantics we carry out our search for appropriate equivalences. However, it turns out that the congruence requirement is substantially more difficult to achieve compared with the situation in CCS. As an indication of this, consider the following suggested equivalence:

$$\boldsymbol{fork}(a); b \equiv a; b + b; a \tag{1.1}$$

Given that $\boldsymbol{fork}(p); q$ is the FC–calculus version of the CCS parallel composition of p and q, (1.1) seems to be a direct consequence of the expansion law of CCS. However, placing the two expressions $\boldsymbol{fork}(a); b$ and $a; b + b; a$ in the context $\_; c$, we obtain two clearly inequivalent expressions:

$$(\boldsymbol{fork}(a); b); c \not\equiv (a; b + b; a); c \tag{1.2}$$

As ; is associative, a will be in parallel with $b; c$ on the left–hand side of (1.2) and may thus occur *after* c. Clearly, this is not possible on the right–hand side, and therefore we must reject the equivalence between the two expressions of (1.2). However, requiring $\equiv$ to be a congruence, this means that we must also reject the equivalence in (1.1). The difference between the two terms of (1.1) can be loosely characterized as a difference in the ability of a to be in parallel with a potential future computation. Attempting to remove this difference we may suggest the following equivalence instead of (1.1):

$$\boldsymbol{fork}(a); b \equiv a; b + b; \boldsymbol{fork}(a) \tag{1.3}$$

It is not only the fork operator, that causes a challenge when defining a logic for CML. Also the fact that CML expressions may allocate new channels and communicate these as values is a challenge. We consequently investigate two extensions of the Fork Calculus, one where channel allocation is possible, but where channels cannot be communicated, and one where they can additionally be communicated. Hence, we investigate three calculi, and a logic for each of them. The results obtained from investigating these calculi are finally applied in order to obtain a logic for CML.

Our work is close to work on Facile, which in many ways is comparable to CML — there is for example a fork operator. In [PGM90], Facile is given an operational semantics and a notion of equivalence is developed. The Facile equivalence is, however, not shown to be a congruence in the general case. This is caused by the fact that processes are values. We have concluded, that the issues of equivalence, compositionality and logic, in the presence of forking, can best be studied in isolation via the Fork Calculus. Although some of our techniques are related to those of [PGM90], this simplification gives us a congruence, and in addition we obtain a complete axiomatisation.

1.3 Contents of the Thesis

The contents of this thesis are divided into five parts. Part I includes this introduction, which is chapter 1, and a further chapter 2 on preliminary theory of processes, which is used in the remainder. Part V contains the conclusion as chapter 10, and is followed by four appendices containing various proofs of properties stated throughout the thesis. The parts II, III and IV constitute the main substance. The contents of these three parts are illustrated in figure 1.1, which we shall comment below. Each box in the figure illustrates a calculus or a logic. White boxes illustrate our contribution, while dashed boxes illustrate already existing theories.

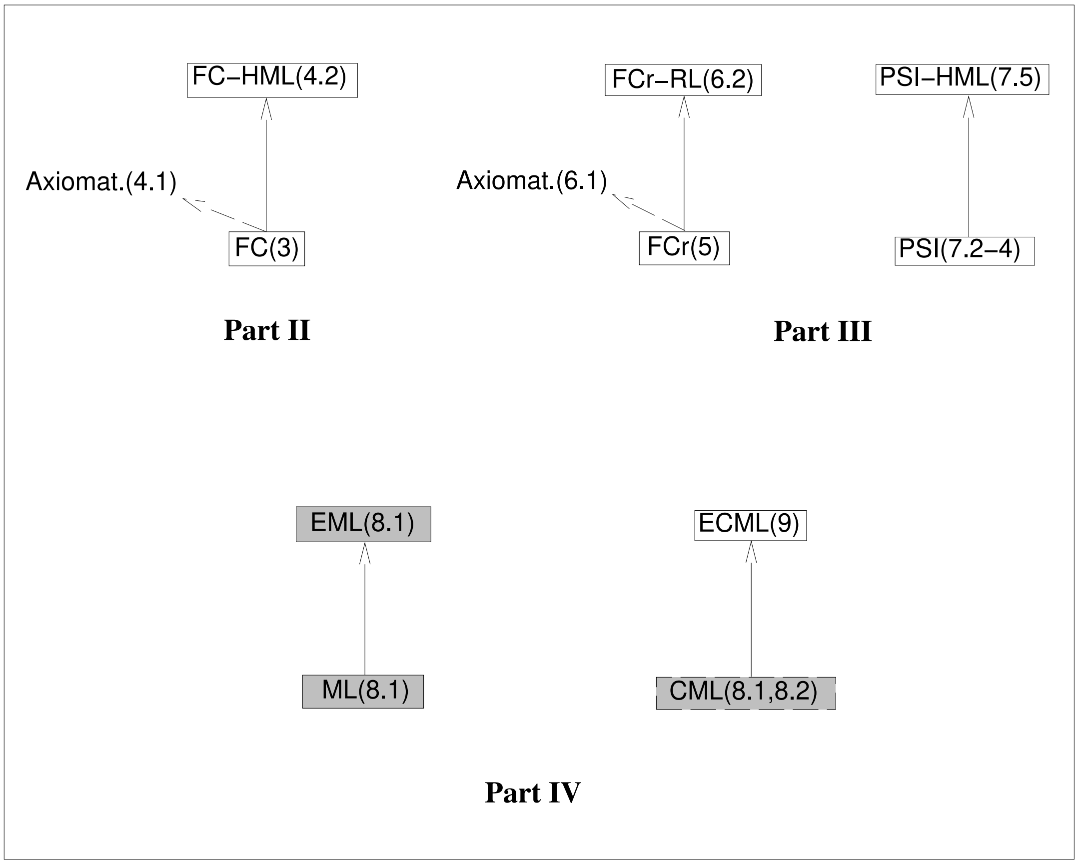

Figure 1.1: Five Language Towers

Part II: A Basic Process Calculus

This part defines the Fork Calculus, FC.

Chapter 3 defines the syntax and operational semantics of the calculus, together with various notions of process equivalence. The main problem is to obtain an equivalence relation that is also a congruence with respect to the operators of the calculus. First, we present the obvious (strong bisimulation) equivalence between processes based on the operational semantics. As indicated in the discussion above, it turns out that this equivalence is *not* a congruence with respect to the operators of FC, a rather remarkable fact as strong bisimulation equivalence is preserved by almost all constructs introduced so far in process algebra (see e.g. [GV89]). Then we provide an explicit characterisation of the congruence induced by the strong bisimulation equivalence. The congruence is essentially a refinement of the strong bisimulation equivalence taking into account forked processes' ability to compute in parallel with potential future computations.

Chapter 4 presents two ways of reasoning about processes.

One way of reasoning about processes is to prove equivalences between them, using an axiomatisation of the calculus. Section 4.1 presents such an axiomatisation, which is shown sound and complete with respect to the strong congruence defined in chapter 3. The axiomatisation consists of a number of basic equational axioms and two expansion laws allowing as much as possible the ***fork***–operator to be replaced by non–deterministic choice. The completeness proof is based on a notion of normal–form. Note, that in contrast to the parallel composition operator of CCS, the fork operator can generally not be expanded totally away.

Another way of reasoning about processes is to formulate a property in some modal logic, and then to show that the process satisfies the property. Section 4.2 defines such a logic for FC, and it is shown to be adequate with respect to the equivalence axiomatised in section 4.1. That is, two processes are equivalent if and only if they satisfy the same properties. The logic is quite close to Hennessy-Milner logic (HML) [HM85], but differs slightly in order to deal with the properties of the fork operator. A logic for FC is an essential step towards a logic for CML.

Part III: Extensions of the Fork Calculus

This part presents two extensions of FC towards CML. The Fork Calculus defined in the previous part provides no mechanism for internalising channels. That is, all channels are globally visible. The two extensions both provide mechanisms for dealing with the internalisation of channels.

Chapter 5 defines a calculus, FC_r, which has a channel restriction operator, quite similar to the one in CCS. As such, only after this extension we arrive at a calculus that is really comparable to CCS. Chapter 6 presents the two ways of reasoning about processes that we saw for the original Fork Calculus: an axiomatisation in section 6.1 and a logic in section 6.2. The logic is, however, quite different from the logic defined in section 4.2. It is a refinement logic (RL), which allows compositional reasoning via the fact that the operators of the Fork Calculus are operators in the logic in addition to the modal logic operators. This means for example, that programs are special cases of formulae. Such a logic allows stepwise refinement of specifications into programs.

Chapter 7 defines a further extension of the Fork Calculus called the ψ-calculus, where channels can be communicated as values between processes in the same way as is the case in the π-calculus, which in turn can be seen as an extension of CCS. Also a logic is defined for the ψ-calculus. The logic is inspired by our Hennessy-Milner-like logic for FC in section 4.2, and it is also inspired by the logic for the π-calculus [MPW91]. The latter solves the problem of how to specify scope extrusion: the fact that a process communicates a channel that is local to the process, and which first becomes global after the communication.

Part IV: Towards a Logic for Concurrent ML

The last part approaches the task of defining a logic for CML.

Chapter 8 contains a short tutorial on ML, EML and CML (section 8.1). In addition (section 8.2) we define an operational semantics of CML. That is, we define a *labelled* transition system, where the labels are the actions (communications) that CML expressions perform. The action-labels on transitions are needed in order to define a logic, since the logic "talks about" these actions, just as does traditional Hennessy-Milner logic. The existing semantics of CML are basically defined as *unlabelled* transition systems.

Chapter 9 defines the extension ECML, which is an extension of CML with specification constructs. A complete syntax for a realistic language is defined, rather than for a toy calculus as in the previous chapters. This allows some more interesting examples to be written. On the other hand, we shall be less formal what concerns the semantics. The chapter can be seen a bit as an 'example driven' language design in contrast to the previous chapters where language (calculus) design is 'semantics driven'.

Chapter 2

Preliminary Theory of Processes

This chapter introduces an extract from standard process theory (we have used [Mil89]) of a set of definitions and theorems, all of which are used in the rest of the work. The chapter thus presents part of our theory construction and is not just a survey of "related work". Section 2.1 defines the notion of a labelled transition system, which will be our fundamental semantic tool for defining the semantics of process calculi. Sections 2.2 and 2.3 define what it means for two processes to be equivalent – the basic notion being that of a bisimulation. Section 2.2 deals with strong equivalence, where internal transitions are observable, while section 2.3 deals with weak equivalence where internal transitions are not observable.

2.1 Labelled Transition Systems

The approach to semantics we shall introduce in this chapter is that of 'structured operational semantics'. That is, the approach taken is that of labelled transition systems as introduced by Plotkin [Plo81] and later applied to CCS as described in [Mil89]. The operational semantics approach is also the one applied in the existing two semantics of CML: [Rep91c, BMT92]. We shall introduce the general concepts, which are independent of the particular language given semantics. The fundamental structure in the theory is that of a labelled transition system:

Definition 2.1 (Labelled Transition System) *A labelled transition system is a triple:*

$$(St, Lb, \longrightarrow)$$

where St is a set of states, Lb is a set of transition labels, and $\longrightarrow \subseteq St \times Lb \times St$ is a transition relation.

The elements in St may for example be programs (processes,...). The elements of Lb may be the actions that programs can perform, and the transition relation

$\longrightarrow$ may be thought of as defining the dynamic change of programs as they perform actions. For any p, q in St and l in Lb, we shall write $p \stackrel{l}{\longrightarrow} q$ instead of $(p, l, q) \in \longrightarrow$. We shall interpretate this as *the state p can turn into the state q via an l transition.* If we think of the elements of St as programs and if we think of the elements in Lb as actions that programs can perform, we can alternatively read $p \stackrel{l}{\longrightarrow} q$ as: *the program p can perform the action l and thereby become the program q.*

2.2 Strong Equivalence

Given a labelled transition system $lts = (St, Lb, \longrightarrow)$, one can derive a notion of equivalence between the elements in St. The notion of equivalence that we adopt is based on the concept of bisimulation [Mil89], which again is based on the idea that we only want to distinguish between two programs, if the distinction can be observed by an "observer" examining the two programs in a stepwise manner.

To give an idea of what 'stepwise' means, suppose two programs p and q are examined, and regard the actions they can perform as "buttons" that the observer can press. If a process can perform a certain action, then the "button" can be pressed, otherwise not. Suppose further that the observer can press a button in p and get a changed program p'. Then (for the programs to be equivalent) he should be able to press the same button in q and get also there are changed program q'. Now he can continue with p' and q'. If he is always able to perform this "bisimulation" activity on the (changed) programs (also symmetrically starting with q) then p and q are equivalent. If he, however, reaches a situation where in one program we can press a button, but not in the other, then p and q are not equivalent.

Thus, for the two processes to be equivalent, they must be able to simulate each other, step by step. At any point in the "simulation", the two processes must be able to perform exactly the same set of actions. We can formulate this slightly more formal as follows. We say that q is an l-derivative of p, if $p \stackrel{l}{\longrightarrow} q$. Then what we are looking for is a relation with the following property:

> *p and q are equivalent iff, for every action l,*
> *every l-derivative of p is equivalent to some*
> *l-derivative of q and conversely.*

Based on these informal remarks, we proceed with the formal definition and we first define the notion of a bisimulation.

Definition 2.2 (Bisimulation) *A binary relation $S \subseteq St \times St$ is a bisimulation iff $(p, q) \in S$ implies, for all $l \in Lb$,*

1. *Whenever* $p \xrightarrow{l} p'$ *for some* p' *then* $q \xrightarrow{l} q'$ *for some* q' *and* $(p', q') \in S$

2. *Whenever* $q \xrightarrow{l} q'$ *for some* q' *then* $p \xrightarrow{l} p'$ *for some* p' *and* $(p', q') \in S$

We can now define $\sim$ as the union of all bisimulations:

Definition 2.3 (Equivalence) *The equivalence relation* $\sim$ *is obtained as the union of all bisimulations:*

$$\sim \quad \stackrel{def}{=} \quad \bigcup\{S \mid Bisimulation(S)\}$$

Put otherwise: p is equivalent to q, written $p \sim q$, iff. $(p, q) \in S$ for some bisimulation S. This gives us a proof technique: to prove two programs equivalent, prove that the pairing of the two belongs to some bisimulation.

The relation $\sim$ is itself a bisimulation, in fact:

Lemma 2.4 $\sim$ *is the largest bisimulation.*

Proof

See appendix A lemma A.2.

■

The relation is an equivalence in that it is reflexive, symmetric and transitive:

Theorem 2.5 ($\sim$ is an equivalence) *For any programs* p, q *and* r *it holds that:*

$$\begin{array}{ll} (1) & p \sim p \\ (2) & p \sim q \textit{ implies } q \sim p \\ (3) & p \sim q \textit{ and } q \sim r \textit{ implies } p \sim r \end{array}$$

Proof

See appendix A lemma A.3.

■

The relation $\sim$ satisfies the following important property, which is frequently used in proofs.

Proposition 2.6 (Behaviour proposition) *For all programs* p, q*:* $p \sim q$ *iff. for all* $l \in Lb$*,*

1. *Whenever* $p \xrightarrow{l} p'$ *for some* p' *then* $q \xrightarrow{l} q'$ *for some* q' *and* $p' \sim q'$
2. *Whenever* $q \xrightarrow{l} q'$ *for some* q' *then* $p \xrightarrow{l} p'$ *for some* p' *and* $p' \sim q'$

Proof

See appendix A proposition A.4

■

Note that the 'only if' ($\Rightarrow$) part of this proposition already follows from the fact that $\sim$ is a bisimulation. The interesting part of the proposition is the 'if' ($\Leftarrow$) part, so if for two programs p and q we can show that they can perform the same actions and that the resulting programs are equivalent, then p and q are equivalent.

2.3 Weak Equivalence

In the previous presentation, all actions in Lb were regarded equally observable. In this section we shall follow the idea that certain actions are not observable, we call these actions for silent actions. When we compare two programs, we are not really interested in observing the silent actions. It is standard to assume the existence of a single silent action called τ (tau). The domain of labels Lb can thus be refined into:

$$Lb \stackrel{def}{=} ObsLb \cup \{\tau\}$$

where $ObsLb$ (observable labels) is a set of labels that we want to observe.

In this section we shall develop a notion of equivalence that is based on what we can observe. In order to define such an equivalence on programs, we need to define an extra transition relation that ignores τ actions:

Definition 2.7 (Weak Transition Relation) *Let* $\Longrightarrow$ *be the smallest subset of* $St \times Lb \times St$ *closed under the following rules:*

$$\frac{}{p \stackrel{\tau}{\Longrightarrow} p}$$

$$\frac{p \stackrel{\tau}{\Longrightarrow} \stackrel{l}{\longrightarrow} \stackrel{\tau}{\Longrightarrow} p'}{p \stackrel{l}{\Longrightarrow} p'}$$

Then we can define a weak notion of bisimulation, that basically "ignores" τ actions.

Definition 2.8 (Weak Bisimulation) *A binary relation* $S \subseteq St \times St$ *is a weak bisimulation iff* $(p, q) \in S$ *implies, for all* $l \in Lb$,

1. *Whenever* $p \xrightarrow{l} p'$ *for some* p' *then* $q \stackrel{l}{\Longrightarrow} q'$ *for some* q' *and* $(p', q') \in S$
2. *Whenever* $q \xrightarrow{l} q'$ *for some* q' *then* $p \stackrel{l}{\Longrightarrow} p'$ *for some* p' *and* $(p', q') \in S$

We can now define $\approx$ as the union of all weak bisimulations:

Definition 2.9 (Weak Equivalence) *The equivalence relation* $\approx$ *is obtained as the union of all weak bisimulations:*

$$\approx \quad \stackrel{def}{=} \quad \bigcup\{S \mid Weak\_Bisimulation(S)\}$$

Put otherwise: p is weakly equivalent to q, written $p \approx q$, iff. $(p, q) \in S$ for some weak bisimulation S. So, as for the strong case, to prove two programs equivalent, prove that the pairing of the two belong to some weak bisimulation.

The relation $\approx$ is itself a weak bisimulation, in fact:

Lemma 2.10 $\approx$ *is the largest weak bisimulation.*

Proof

See appendix A lemma A.7.

∎

Theorem 2.11 ($\approx$ is an equivalence) *For any programs* p, q *and* r *it holds that:*

$$\begin{array}{ll} (1) & p \approx p \\ (2) & p \approx q \ \textit{implies}\ q \approx p \\ (3) & p \approx q \ \textit{and}\ q \approx r \ \textit{implies}\ p \approx r \end{array}$$

Proof

See appendix A lemma A.8.

∎

The relation $\approx$ satisfies the following important property, which is frequently used in proofs.

Proposition 2.12 (Weak behaviour proposition) *For all programs p, q: $p \approx q$ iff. for all $l \in Lb$,*

1. *Whenever $p \xrightarrow{l} p'$ for some p' then $q \xRightarrow{l} q'$ for some q' and $p' \approx q'$*
2. *Whenever $q \xrightarrow{l} q'$ for some q' then $p \xRightarrow{l} p'$ for some p' and $p' \approx q'$*

Proof

See appendix A proposition A.9.

$\blacksquare$

Part II

A Basic Process Calculus

Chapter 3

The Fork Calculus

In this chapter, we define the Fork Calculus, FC. It is a calculus similar to CCS[Mil89], but there is a unary fork operator and sequential composition instead of parallel composition and action prefixing as in CCS.

Of related work that deals with a fork operator, we can mention the SMoLCS theory [AR87a], [AR87b], [AR87c], and [AR92]. Also the theory of Facile [PGM90] necessarily deals with the fork operator since Facile has a such. As we shall see, processes that work in parallel are kept in a multiset, and forking a process p adds p to the multiset. Multisets are also used in the Chemical Abstract Machine CHAM [BB90]. CHAM has consequently been used to define one of the semantics of Facile [LT92b]. Also [NT92a] demonstrates the use of CHAM to define the semantics of a language that combines functional programming with concurrency, although in that work there is a parallel composition operator and not a fork operator.

Section 3.1 defines the calculus with its operational semantics. Section 3.2 defines a strong congruence relation between processes. A congruence relation is an equivalence relation, which is preserved by the operators of the calculus. The congruence is strong since it is sensitive to internal transitions. Section 3.3 defines a weak congruence relation which is not sensitive to internal transitions.

3.1 Syntax and Semantics

3.1.1 Syntax and Informal Semantics

The syntax, generating the language $\mathcal{L}$, is as follows.

$$p \quad ::= \quad \boldsymbol{nil} \mid \alpha \mid p_1 + p_2 \mid p_1; p_2 \mid \boldsymbol{fork}(p) \mid N$$

$$\alpha \quad ::= \quad \tau \mid a? \mid a!$$

The language $\mathcal{A}$ of actions is ranged over by α. $\boldsymbol{nil}$ is the terminated process that can perform no action. Amongst the actions $\mathcal{A}$ that a process can perform are τ, the internal action, input actions of the form $a?$, and output actions of the form $a!$, where a is a channel name. Two processes that run in parallel may synchronise on complementary actions, one being an input action and the other being an output action containing the same name. The choice between two processes is written as $p_1 + p_2$.

Two processes can be sequentially composed by $p_1; p_2$. This has the traditional interpretation that p_1 is evaluated first until it terminates (becomes $\boldsymbol{nil}$) whereupon p_2 continues. Note that FC here differs from CCS which instead of sequential composition has action prefixing. The other main difference from CCS is the $\boldsymbol{fork}$ expression: a process is forked with $\boldsymbol{fork}(p)$. It means that a separate evaluation of p is begun such that p is made to run in parallel with the rest of the program. The $\boldsymbol{fork}$ expression itself terminates immediately after starting the separate evaluation of p.

N is the call of a process, that has been named in a definition of the form $N \stackrel{def}{=} p$. In the present theory we disallow recursion in order to obtain a complete axiomatisation, and we only introduce process naming to be able to write some more appealing (nonrecursive) examples. Except for the complete axiomatisation, it is very easy to extend the results of this theory to allow recursion.

So what are the consequences of this seemingly minor change compared to CCS: having sequential composition and forking instead of action prefixing and parallel composition? There are pragmatic as well as semantic consequences. As we shall see, the semantic consequences are quite drastic. Concerning the pragmatic consequences, let us study an example. Consider the following informal requirement specification:

A 'session' at a computer terminal consists of an 'initialisation' followed by a 'run'. The initialisation consists of a 'setup' phase followed by a 'dialog' phase. After the 'setup' phase, a report is sent to a paper printer which will 'print' the report.

This requirement specification can be directly presented in FC as follows (note that for convenience, we shall often leave out the '?' when writing input actions; actions are written with small letters by convention):

$$\begin{aligned} \mathit{Session} &\stackrel{def}{=} \mathit{Initialise}; \mathit{run} \\ \mathit{Initialise} &\stackrel{def}{=} \mathit{setup}; \boldsymbol{fork}(\mathit{print}); \mathit{dialog} \end{aligned}$$

The point to note is that we can locally in *Initialise* express the parallel activation of the printer (*print*) after the *setup* phase. This local activation of a process is not possible in CCS, where we would have to write something like:

$$Session \stackrel{def}{=} setup.(print \mid dialog.run.\mathbf{o})$$

We see that it is not possible to give a name (*Initialise*) to the pair *setup* and *dialog*: one looses abstraction. So characteristic of the ***fork*** operator is that we can start a process locally where the need arises, and this gives a possibility of naming a sequential composition that performs parallel activation as a "side effect".

Of course the above pragmatic difference between FC and CCS is just a matter of taste, and we shall in the following concentrate on the formal implications of having '***fork***' and ';'.

3.1.2 Formal Semantics

First, we define an operational semantics for the language of the calculus. That is, the approach taken is that of labelled transition systems as introduced in chapter 2. This approach is also the one applied in the existing two semantics of CML: [Rep91c, BMT92]. The difference between the semantics here and these two is (except for the different languages) that we let transitions be labelled with actions. Both in [Rep91c] and in [BMT92] transitions are basically not labelled. Labels are important from the point of view of defining an appropriate equivalence relation and a logic.

The semantics of CCS is normally given in terms of a single labelled transition system $(St, Lb, \longrightarrow)$, where St is the set of CCS processes. In contrast to the CCS semantics, the FC semantics is divided into two layers, corresponding to two labelled transition systems. In the first layer we give semantics to processes seen in isolation (St is the set of processes). In the next layer, we give semantics to collections (multisets) of processes running in parallel (St is the set of such process collections). As an example, when "running" the process $\boldsymbol{fork}(p); q$ we start out with a collection consisting just of that process. After the forking, we have a collection containing two processes, p and q, running in parallel. We shall refer to a collection of processes as a program (thus in the second layer St is the set of programs).

Processes

In this section we give semantics to processes seen in isolation. We shall do this by defining a labelled transition system $(\mathcal{L}, Lab, \hookrightarrow)$ where $\mathcal{L}$ is the set of processes introduced in section 3.1.1. Concerning the definition of the labels Lab, assume an infinite set of (channel) names *Chan*. Then *Lab* (the labels on process transitions) is gradually defined as follows:

$$\begin{array}{lcl} Com & = & \{a? \mid a \in Chan\} \cup \{a! \mid a \in Chan\} \\ Act & = & Com \cup \{\tau\} \\ Lab & = & Act \cup \{\phi(p) \mid p \in \mathcal{L}\} \end{array}$$

The set *Com*, ranged over by c, is the set of input-output communications that processes can perform. The set *Act*, ranged over by $\alpha, \beta, \gamma, \ldots$, includes in addition the τ action, and it is the set of actions that we will be able to observe in the end, when executing programs. The set *Lab*, ranged over by l, includes further labels of the form $\phi(p)$ ($p \in \mathcal{L}$) which arise from evaluation of processes of the form $\textbf{\textit{fork}}(p)$. These labels will not be observable at the program layer, since at that level they will be converted into τ actions.

We now define the transition relation $\hookrightarrow \subseteq \mathcal{L} \times Lab \times \mathcal{L}$. Before defining this transition relation we define the predicate $Stop \subseteq \mathcal{L}$. We shall use the notation $Stop(p)$ instead of $p \in Stop$. Now *Stop* is defined as the least subset of $\mathcal{L}$ satisfying the following:

1) $Stop(\textbf{\textit{nil}})$
2) If $Stop(p_1)$ and $Stop(p_2)$ then $Stop(p_1 + p_2)$
3) If $Stop(p_1)$ and $Stop(p_2)$ then $Stop(p_1; p_2)$
4) If $Stop(p)$ and $N \stackrel{def}{=} p$ then $Stop(N)$

Essentially, this defines a grammar of stopped processes. The operational semantics of FC processes is then as follows.

Definition 3.1 (The transition relation $\hookrightarrow$) *Let $\hookrightarrow$ be the smallest subset of $\mathcal{L} \times Lab \times \mathcal{L}$ closed under the following rules:*

$$(\textbf{Action}) \qquad \frac{}{\alpha \stackrel{\alpha}{\hookrightarrow} \textbf{\textit{nil}}}$$

$$(\textbf{Choice}_1) \qquad \frac{p_1 \stackrel{l}{\hookrightarrow} p_1'}{p_1 + p_2 \stackrel{l}{\hookrightarrow} p_1'}$$

$$(\textbf{Choice}_2) \qquad \frac{p_2 \stackrel{l}{\hookrightarrow} p_2'}{p_1 + p_2 \stackrel{l}{\hookrightarrow} p_2'}$$

$$(\textbf{Sequence}_1) \quad \frac{p_1 \stackrel{l}{\hookrightarrow} p_1'}{p_1; p_2 \stackrel{l}{\hookrightarrow} p_1'; p_2}$$

$$(\textbf{Sequence}_2) \quad \frac{p_2 \stackrel{l}{\hookrightarrow} p_2'}{p_1; p_2 \stackrel{l}{\hookrightarrow} p_2'} \, Stop(p_1)$$

$$(\textbf{Fork}) \quad \frac{}{\boldsymbol{fork}(p) \stackrel{\phi(p)}{\hookrightarrow} \boldsymbol{nil}}$$

$$(\textbf{Constant}) \quad \frac{p \stackrel{l}{\hookrightarrow} p'}{A \stackrel{l}{\hookrightarrow} p'} \, A \stackrel{def}{=} p$$

The rules should be fairly simple to read. The **Action**-rule says that a process of the form α can perform the action α and become $\boldsymbol{nil}$. The $\textbf{Choice}_1$-rule says that if p_1 can perform l and become p_1', then the sum $p_1 + p_2$ can also, and symmetrically by the $\textbf{Choice}_2$-rule. The sequencing rules explain how sequencing proceeds with the leftmost process until it has stopped ($\textbf{Sequence}_1$), whereupon the rightmost process continues ($\textbf{Sequence}_2$). The **Fork**-rule shows how the higher order labels $\phi(p)$ are created. When we come to the program semantics their use will be explained. Finally, the **Constant**-rule explains how the name of a process behaves as the defining body. Let us look at an example. The process $(\alpha; \boldsymbol{fork}(\beta); \gamma)$ can evaluate as follows:

$$\alpha; \boldsymbol{fork}(\beta); \gamma \stackrel{\alpha}{\hookrightarrow} \boldsymbol{nil}; \boldsymbol{fork}(\beta); \gamma \stackrel{\phi(\beta)}{\hookrightarrow} \boldsymbol{nil}; \gamma \stackrel{\gamma}{\hookrightarrow} \boldsymbol{nil}$$

Note that the forked process β just becomes part of the label, and that it is not further used. When executing programs, we will make sure that the forked process is put in parallel with the "rest of the program". As another example, consider the process $\boldsymbol{fork}(\alpha); \boldsymbol{fork}(\beta)$. Its evaluation proceeds as follows:

$$\boldsymbol{fork}(\alpha); \boldsymbol{fork}(\beta) \stackrel{\phi(\alpha)}{\hookrightarrow} \boldsymbol{nil}; \boldsymbol{fork}(\beta) \stackrel{\phi(\beta)}{\hookrightarrow} \boldsymbol{nil}$$

Again, at this level, nothing is put in parallel.

Programs

A program is a multiset[1] of processes. We let $\mathcal{M}$ denote the set of programs ($\mathcal{M} = MS(\mathcal{L})$).

[1] A multiset can contain several copies of the same element, (in contrast to normal sets), corresponding to the fact that at a certain moment there may be several processes active with

The semantics of programs is given in terms of the labelled transition system $(\mathcal{M}, Act, \longrightarrow)$, where $\mathcal{M}$ is defined here and Act was defined in the previous section. Thus a program can perform the actions in the set Act; recall that these were of the form $a?$, $a!$ or τ. The fork actions $\phi(p)$ are thus not amongst program actions.

We shall now define the transition relation: $\longrightarrow \subseteq \mathcal{M} \times Act \times \mathcal{M}$. We need the auxiliary function $rev : Com \rightarrow Com$, which for a given communication returns the complementary communication with which it can synchronise; i.e: $rev(a?) = a!$ and $rev(a!) = a?$.

The operational semantics of FC programs is then as follows.

Definition 3.2 (The transition relation $\longrightarrow$) *Let $\longrightarrow$ be the smallest subset of $\mathcal{M} \times Act \times \mathcal{M}$ closed under the following rules:*

$$(\textbf{Action}^{\{\!|\,|\!\}}) \qquad \frac{p \overset{\alpha}{\hookrightarrow} p'}{\{\!|p|\!\} \overset{\alpha}{\longrightarrow} \{\!|p'|\!\}}$$

$$(\textbf{Fork}^{\{\!|\,|\!\}}) \qquad \frac{p \overset{\phi(q)}{\hookrightarrow} p'}{\{\!|p|\!\} \overset{\tau}{\longrightarrow} \{\!|p', q|\!\}}$$

$$(\textbf{Parallel}_1^{\{\!|\,|\!\}}) \qquad \frac{P_1 \overset{\alpha}{\longrightarrow} P_1'}{P_1 \cup P_2 \overset{\alpha}{\longrightarrow} P_1' \cup P_2}$$

$$(\textbf{Parallel}_2^{\{\!|\,|\!\}}) \qquad \frac{P_1 \overset{c}{\longrightarrow} P_1' \,,\ P_2 \overset{rev(c)}{\longrightarrow} P_2'}{P_1 \cup P_2 \overset{\tau}{\longrightarrow} P_1' \cup P_2'}$$

The $\textbf{Action}^{\{\!|\,|\!\}}$-rule just says that if a process can perform an action, then a program containing that process can perform the action. Note that α ranges over Act, which does not contain $\phi(p)$ actions. The $\textbf{Fork}^{\{\!|\,|\!\}}$-rule explains how a $\phi(\ldots)$ action results in a τ action at program level: the forking process and the forked process will after the transition be in parallel. Finally, the rule $\textbf{Parallel}_1^{\{\!|\,|\!\}}$ explains how a "subset" of a program may perform actions on its own. Note, that we need only one such rule as $\cup$ is clearly commutative. The $\textbf{Parallel}_2^{\{\!|\,|\!\}}$–rule shows how two distinct subsets of a program may communicate, resulting in a τ action.

exactly the same structure. Formally, the multisets of A elements can be viewed as the elements in $MS(A) = A \rightarrow \mathbf{N}$. That is, a multiset of A elements is a total function from A to the natural numbers. Each A element is mapped to its number of occurrences. The union operator $\cup : (MS(A) \times MS(A)) \rightarrow MS(A)$ is defined by the equation $(S_1 \cup S_2)(a) = S_1(a) + S_2(a)$. A "finite" multiset S can be written $\{\!|p_1, \ldots, p_n|\!\}$ where the number of occurrences of a process q indicates the value $S(q)$. As an example, $\{\!|p|\!\}(p) = 1$ and $\{\!|p|\!\}(q) = 0$ whenever $p \neq q$.

Example 3.3 *As an example, let us deduce how the program* $\{\!|\boldsymbol{fork}(\alpha);\beta|\!\}$ *can perform the three transitions* $\tau\beta\alpha$ *(note that it is also possible for the program to perform the transitions* $\tau\alpha\beta$*). The first transition is deduced as follows (each deduction refers to the semantic rule used):*

$$\textbf{(Fork)}\quad \frac{}{\boldsymbol{fork}(\alpha) \overset{\phi(\alpha)}{\hookrightarrow} \boldsymbol{nil}}$$

$$(\textbf{Sequence}_1)\quad \frac{}{\boldsymbol{fork}(\alpha);\beta \overset{\phi(\alpha)}{\hookrightarrow} \boldsymbol{nil};\beta}$$

$$(\textbf{Fork}^{\{\!|\,|\!\}})\quad \frac{}{\{\!|\boldsymbol{fork}(\alpha);\beta|\!\} \xrightarrow{\tau} \{\!|\boldsymbol{nil};\beta,\alpha|\!\}}$$

The second transition is deduced as follows:

$$\textbf{(Action)}\quad \frac{}{\beta \overset{\beta}{\hookrightarrow} \boldsymbol{nil}}$$

$$(\textbf{Sequence}_2)\quad \frac{}{\boldsymbol{nil};\beta \overset{\beta}{\hookrightarrow} \boldsymbol{nil}}$$

$$(\textbf{Action}^{\{\!|\,|\!\}})\quad \frac{}{\{\!|\boldsymbol{nil};\beta|\!\} \xrightarrow{\beta} \{\!|\boldsymbol{nil}|\!\}}$$

$$(\textbf{Parallel}_1^{\{\!|\,|\!\}})\quad \frac{}{\{\!|\boldsymbol{nil};\beta,\alpha|\!\} \xrightarrow{\beta} \{\!|\boldsymbol{nil},\alpha|\!\}}$$

Finally, the third transition is deduced as follows:

$$\textbf{(Action)}\quad \frac{}{\alpha \overset{\alpha}{\hookrightarrow} \boldsymbol{nil}}$$

$$(\textbf{Action}^{\{\!|\,|\!\}})\quad \frac{}{\{\!|\alpha|\!\} \xrightarrow{\alpha} \{\!|\boldsymbol{nil}|\!\}}$$

$$(\textbf{Parallel}_1^{\{\!|\,|\!\}})\quad \frac{}{\{\!|\boldsymbol{nil},\alpha|\!\} \xrightarrow{\alpha} \{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\}}$$

The program can thus evaluate as follows:

$$\{\!|\boldsymbol{fork}(\alpha);\beta|\!\} \xrightarrow{\tau} \{\!|\boldsymbol{nil};\beta,\alpha|\!\} \xrightarrow{\beta} \{\!|\boldsymbol{nil},\alpha|\!\} \xrightarrow{\alpha} \{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\}$$

∎

3.2 Strong Congruence

We shall define a congruence relation, $\equiv$, between processes in such a way that for any two processes p_1 and p_2: $p_1 \equiv p_2$ if and only if the two processes behave *the same way*, and, whenever we place p_1 in some context, we get the same result as if we place p_2 in the same context. That is, for any context $C[.]$ being some FC-program with a hole in it, we want that $C[p_1] \equiv C[p_2]$ provided that $p_1 \equiv p_2$.

Section 3.2.1 presents a first natural quess of an equivalence relation. It, however, turns out not to be a congruence. Section 3.2.2 then presents a modification which is a congruence.

3.2.1 Process Equivalence

The purpose of this section is to define an equivalence relation $\sim \subseteq \mathcal{L} \times \mathcal{L}$ between FC processes. For this purpose we shall, however, first define an equivalence relation $\smile \subseteq \mathcal{M} \times \mathcal{M}$ between FC programs.

Program Equivalence

We shall first define $\smile$ in terms of the concept of bisimulation. We will then show that this relation is a congruence, a rather important property.

We note that the semantics of programs is a labelled transition system $(\mathcal{M}, Act, \longrightarrow)$. As described in chapter 2, such a labelled transition system generates a bisimulation equivalence between the programs in $\mathcal{M}$. We shall let $\smile$ denote this equivalence. Recall that $\smile$ is defined as the union of all bisimulations. As an example of a bisimulation for our labelled transitions system, consider the one containing the pairs:

$$\begin{array}{rcl} \{\!|\alpha; \beta|\!\} & , & \{\!|\alpha; \beta + \alpha; (\beta + \beta)|\!\} \\ \{\!|\boldsymbol{nil}; \beta|\!\} & , & \{\!|\boldsymbol{nil}; \beta|\!\} \\ \{\!|\boldsymbol{nil}; \beta|\!\} & , & \{\!|\boldsymbol{nil}; (\beta + \beta)|\!\} \\ \{\!|\boldsymbol{nil}|\!\} & , & \{\!|\boldsymbol{nil}|\!\} \end{array}$$

This bisimulation contains the pair $(\{\!|\alpha; \beta|\!\}, \{\!|\alpha; \beta+\alpha; (\beta+\beta)|\!\}$, so we can conclude that $\{\!|\alpha; \beta|\!\} \smile \{\!|\alpha; \beta+\alpha; (\beta+\beta)|\!\}$. The other pairs of this relation are obtained as results of performing actions. So the second and third pair are each obtained from the first by performing the α action, depending on which outermost $+$ branch is chosen ($\alpha; \beta$ or $\alpha; (\beta + \beta)$). The last pair is obtained from the second as well as the third by performing the β action.

As another example, suppose we want to prove that $\{\!|\alpha; (\beta+\gamma)|\!\} \not\smile \{\!|\alpha; \beta+\alpha; \gamma|\!\}$. We prove this by contradiction. Suppose there is a bisimulation S containing the pair $(\{\!|\alpha; (\beta + \gamma)|\!\}, \{\!|\alpha; \beta + \alpha; \gamma|\!\})$. Then S must contain also the pair $(\boldsymbol{nil}; (\beta + \gamma), \boldsymbol{nil}; \beta)$. But the left component can perform a γ action while the right component cannot. Therefore S is not a bisimulation, and our assumption was wrong.

Note that here we regard the τ action just as observable as the other actions ($a?$ and $a!$) in Act. We shall for example distinguish $\{\!|\alpha; \tau; \beta|\!\}$ and $\{\!|\alpha; \beta|\!\}$. This yields a rather strict equivalence (fewer programs are equivalent). We shall later discuss the situation where we no longer want to observe the τ actions, thus regarding them as internal. We have chosen to make τ observable in our first attempt, since it is the classical choice: first a strong notion of equivalence is defined (where τ is observable), and then one abstracts from τ.

We shall now state and prove the quite essential property of $\smile$ that it is a congruence: $\smile$ divides programs into equivalence classes since it is reflexive, symmetric and transitive, and the $\cup$ operator defined on programs can be applied to any representative of an equivalence class without changing what is the resulting equivalence class. The congruence property is expressed formally as follows:

Lemma 3.4 ($\smile$ is a congruence) *For all programs P_1, P_2 and Q,*

$$\textit{Whenever } P_1 \smile P_2 \textit{ then } P_1 \cup Q \smile P_2 \cup Q$$

Proof

We will show, that the following relation S is a bisimulation:

$$S \stackrel{def}{=} \{(P_1 \cup Q, P_2 \cup Q) \mid P_1 \smile P_2\}$$

Assume that $(P_1 \cup Q, P_2 \cup Q) \in S$ and let $P_1 \cup Q \xrightarrow{\alpha}$. There are three cases:

Case 1 $P_1 \xrightarrow{\alpha} P_1'$ and $P_1 \cup Q \xrightarrow{\alpha} P_1' \cup Q$. Then since $P_1 \smile P_2$ we have $P_2 \xrightarrow{\alpha} P_2'$ where $P_1' \smile P_2'$. So $P_2 \cup Q \xrightarrow{\alpha} P_2' \cup Q$ and $(P_1' \cup Q, P_2' \cup Q) \in S$.

Case 2 $Q \xrightarrow{\alpha} Q'$ and $P_1 \cup Q \xrightarrow{\alpha} P_1 \cup Q'$. Then $P_2 \cup Q \xrightarrow{\alpha} P_2 \cup Q'$ and $(P_1 \cup Q', P_2 \cup Q') \in S$.

Case 3 $\alpha = \tau$, $P1 \xrightarrow{c} P1'$, $Q \xrightarrow{rev(c)} Q'$ and $P1 \cup Q \xrightarrow{\tau} P1' \cup Q'$. Then since $P_1 \smile P_2$ we have $P_2 \xrightarrow{c} P_2'$ where $P_1' \smile P_2'$. So $P_2 \cup Q \xrightarrow{\tau} P_2' \cup Q'$ and $(P_1' \cup Q', P_2' \cup Q') \in S$.

A symmetric argument starting with $P_2 \cup Q \xrightarrow{\alpha}$ ends the proof.

■

Process Equivalence

In the previous section we introduced an equivalence on programs, and we showed it to be a congruence with respect to the basic operator on programs: $\cup$. What we are really interested in is, however, not programs, but rather processes in $\mathcal{L}$. Processes are the terms of our language, and programs are "just" semantic objects used for giving semantics to processes. So our definite goal is to define an equivalence on processes. This equivalence must additionally be a congruence with respect to the operators on processes ($+$; ***fork***).

In this section we shall come up with a process equivalence $\sim$, which seems rather natural and correct, but which, however, turns out not to be satisfactory due to lack of the congruence property. In section 3.2.2 we will define an equivalence $\equiv$ that is also a congruence, in fact the congruence induced by $\sim$; but the present exercise can hopefully motivate the final solution.

We shall define an equivalence relation $\sim \subseteq \mathcal{L} \times \mathcal{L}$ between FC processes. Two processes p and q are said to be equivalent if $(p, q) \in \sim$, which is written more conveniently as $p \sim q$.

Well, the idea is to say that two processes are equivalent, if they are equivalent when regarded as programs. Put differently: to see whether two processes are equivalent, "run" them (as programs) and see whether they behave the same way. Formally, we thus define $\sim$ as follows:

Definition 3.5 (Process equivalence) *The relation $\sim \subseteq \mathcal{L} \times \mathcal{L}$ is defined as:*

$$p \sim q \Leftrightarrow \{\!|p|\!\} \smile \{\!|q|\!\}$$

Let us give a couple of examples. Earlier we argued that $\{\!|\alpha;\beta|\!\} \smile \{\!|\alpha;\beta + \alpha;(\beta+\beta)|\!\}$. We can therefore conclude that $\alpha;\beta \sim \alpha;\beta + \alpha;(\beta+\beta)$. Likewise, we argued that $\{\!|\alpha;(\beta+\gamma)|\!\} \not\smile \{\!|\alpha;\beta + \alpha;\gamma|\!\}$, thus $\alpha;(\beta+\gamma) \not\sim \alpha;\beta + \alpha;\gamma$.

The relation $\sim$ is an equivalence, as stated in the following proposition:

Proposition 3.6 (Equivalence Property) *For all processes p, q and r it holds that:*

$$\begin{array}{ll} (1) & p \sim p \\ (2) & p \sim q \textit{ implies } q \sim p \\ (3) & p \sim q \textit{ and } q \sim r \textit{ implies } p \sim r \end{array}$$

Proof

(1) Since $\smile$ is an equivalence, it follows that $\{|p|\} \smile \{|p|\}$, and thereby it follows that $p \sim p$.

(2) Assume that $p \sim q$. Then $\{|p|\} \smile \{|q|\}$. Since $\smile$ is an equivalence it follows that $\{|q|\} \smile \{|p|\}$, and thereby that $q \sim p$.

(3) Assume that $p \sim q$ and $p \sim r$. Then $\{|p|\} \smile \{|q|\}$ and $\{|q|\} \smile \{|r|\}$. Since $\smile$ is an equivalence it follows that $\{|p|\} \smile \{|r|\}$, and thereby that $p \sim r$.

■

Common to these two examples of equivalence (respectively non equivalence) given above is that they involve only the operators $+$ and ;. The interesting operator is, however, the ***fork*** operator, and in the following we shall see that it (in combination with ;) causes $\sim$ not to be a congruence. In general, for $\sim$ to be a congruence, the following must hold for any operator $Op \in \{+, ;, \boldsymbol{fork}\}$ of the calculus: for all processes p_1, p_2,

$$p_1 \sim p_2 \text{ implies } Op(\ldots, p_1, \ldots) \sim Op(\ldots, p_2, \ldots)$$

The fact that $\sim$ is not a congruence is rather unusual since (strong) bisimilarity is normally preserved by operators. It is this semantical "melt-down", that originally caused the design, and investigation, of this calculus. Let us illustrate why it is not a congruence. Consider two processes that are related by $\sim$:

$$\tau; \alpha \quad \sim \quad \boldsymbol{fork}(\alpha)$$

To see this, consider the behaviours of the programs $\{|\tau; \alpha|\}$ and $\{|\boldsymbol{fork}(\alpha)|\}$:

$$\begin{array}{lllll} \{|\tau; \alpha|\} & \stackrel{\tau}{\hookrightarrow} & \{|\boldsymbol{nil}; \alpha|\} & \stackrel{\alpha}{\hookrightarrow} & \{|\boldsymbol{nil}|\} \\ \{|\boldsymbol{fork}(\alpha)|\} & \stackrel{\tau}{\hookrightarrow} & \{|\boldsymbol{nil}, \alpha|\} & \stackrel{\alpha}{\hookrightarrow} & \{|\boldsymbol{nil}, \boldsymbol{nil}|\} \end{array}$$

Clearly these behaviours are bisimilar (as they are identical). Now, suppose that we put the two processes into the same context $\_;\beta$. Will it then hold that $\tau;\alpha;\beta \sim \boldsymbol{fork}(\alpha);\beta$? Unfortunately no!. This can be seen from the behaviours of the programs $\{\!|\tau;\alpha;\beta|\!\}$ and $\{\!|\boldsymbol{fork}(\alpha);\beta|\!\}$, as illustrated by the transition trees for the two programs in figure 3.1.

$$
\begin{array}{c}
\{\!|\tau;\alpha;\beta|\!\} \\
\downarrow \tau \\
\{\!|\boldsymbol{nil};\alpha;\beta|\!\} \\
\downarrow \alpha \\
\{\!|\boldsymbol{nil};\beta|\!\} \\
\downarrow \beta \\
\{\!|\boldsymbol{nil}|\!\}
\end{array}
\qquad\qquad
\begin{array}{ccccc}
 & & \{\!|\boldsymbol{fork}(\alpha);\beta|\!\} & & \\
 & & \downarrow \tau & & \\
 & & \{\!|\boldsymbol{nil};\beta,\alpha|\!\} & & \\
 & \alpha \swarrow & & \searrow \beta & \\
\{\!|\boldsymbol{nil};\beta,\boldsymbol{nil}|\!\} & & & & \{\!|\boldsymbol{nil},\alpha|\!\} \\
\downarrow \beta & & & & \downarrow \alpha \\
\{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\} & & & & \{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\}
\end{array}
$$

Figure 3.1: Different Transition Trees

Thus, $\sim$ is not a congruence, and we have to repair this. In the following, we introduce a process relation that *is* a congruence. It is even the largest congruence induced by (contained in) $\sim$.

3.2.2 Process Congruence

In this section we introduce a new process equivalence $\equiv$ being strictly finer than the previously given equivalence $\sim$. As one of the main results, we prove that $\equiv$ is precisely the congruence induced by $\sim$; i.e. $\equiv$ is preserved by all operators of FC, and is the largest such equivalence included in $\sim$.

The fact that $\sim$ is not a congruence is illustrated by the previous example demonstrating the difference between the two processes $\tau;\alpha$ and $\boldsymbol{fork}(\alpha)$ when put into the context $\_;\beta$. The difference lies essentially in the ability of the action α to be in parallel with future computation, which is here represented by the action β.

It is this *ability to be in parallel with future computation* that we want to capture. Future computation is what computes after the *termination* of the observed process. When examining the two programs $\{\!|\tau;\alpha|\!\}$ and $\{\!|\boldsymbol{fork}(\alpha)|\!\}$, we cannot detect this termination: both programs perform a τ action and then an α action, and that's it.

The process $\tau;\alpha$, however, terminates after the second action (α), while the process $\boldsymbol{fork}(\alpha)$ already terminates after the first action (τ), when the forking has been performed — the forked process α can then execute after this termination.

We thus need to add some information that makes it possible to observe termination. This can be done by introducing a special event, π, that, when being performed, signals the successful termination of the observed process. For a given process p under observation, we shall examine $\{\!|p;\pi|\!\}$ rather than just $\{\!|p|\!\}$.

As an example, in order to observe the difference in terms of future computations between the processes $\tau;\alpha$ and $\boldsymbol{fork}(\alpha)$, we "run" $\tau;\alpha;\pi$ and $\boldsymbol{fork}(\alpha);\pi$ as programs. That is, we examine $\{\!|\tau;\alpha;\pi|\!\}$ and $\{\!|\boldsymbol{fork}(\alpha);\pi|\!\}$. Both programs can perform the trace $\tau\alpha\pi$, but only the program $\{\!|\boldsymbol{fork}(\alpha);\pi|\!\}$ can perform the trace $\tau\pi\alpha$, where the α action occurs after the π action. This may be illustrated by the transition trees for the two programs, which will be identical to the trees in figure 3.1 with π replacing β.

We are now able to give the following formal definition of the process equivalence $\equiv$:

Definition 3.7 (Process congruence $\equiv$) *The relation $\equiv\,\subseteq\,\mathcal{L}\times\mathcal{L}$ is defined as* [1]*:*

$$p \equiv q \Leftrightarrow \{\!|p;\pi|\!\} \smile \{\!|q;\pi|\!\}$$

Example 3.8 *In this example, we shall demonstrate that the following equivalence holds:*

$$\boldsymbol{fork}(\boldsymbol{fork}(\alpha)) \equiv \boldsymbol{fork}(\tau;\alpha)$$

Due to definition 3.7 we must simply argue for the program equivalence:

$$\{\!|\boldsymbol{fork}(\boldsymbol{fork}(\alpha));\pi|\!\} \smile \{\!|\boldsymbol{fork}(\tau;\alpha);\pi|\!\}$$

To see that this equation holds, we exhibit the following bisimulation S containing the pair $(\{\!|\boldsymbol{fork}(\boldsymbol{fork}(\alpha));\pi|\!\}, \{\!|\boldsymbol{fork}(\tau;\alpha);\pi|\!\})$

$$\begin{array}{rcl}
\{\!|\boldsymbol{fork}(\boldsymbol{fork}(\alpha));\pi|\!\} & , & \{\!|\boldsymbol{fork}(\tau;\alpha);\pi|\!\} \\
\{\!|\boldsymbol{nil};\pi,\boldsymbol{fork}(\alpha)|\!\} & , & \{\!|\boldsymbol{nil};\pi,\tau;\alpha|\!\} \\
\{\!|\boldsymbol{nil},\boldsymbol{fork}(\alpha)|\!\} & , & \{\!|\boldsymbol{nil},\tau;\alpha|\!\} \\
\{\!|\boldsymbol{nil},\boldsymbol{nil},\alpha|\!\} & , & \{\!|\boldsymbol{nil},\boldsymbol{nil};\alpha|\!\} \\
\{\!|\boldsymbol{nil},\boldsymbol{nil},\boldsymbol{nil}|\!\} & , & \{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\} \\
\{\!|\boldsymbol{nil};\pi,\boldsymbol{nil},\alpha|\!\} & , & \{\!|\boldsymbol{nil};\pi,\boldsymbol{nil};\alpha|\!\} \\
\{\!|\boldsymbol{nil};\pi,\boldsymbol{nil},\boldsymbol{nil}|\!\} & , & \{\!|\boldsymbol{nil};\pi,\boldsymbol{nil}|\!\}
\end{array}$$

It is easy to verify that S is indeed a bisimulation. ■

[1] This equivalence is similar to one of the equivalences defined in [PGM90]; however, no general congruence result is stated in [PGM90].

As one of the main results of this chapter we will in the remainder of this section prove that $\equiv$ is the congruence induced by $\sim$. First of all, it is straightforward to show that $\equiv$ is an equivalence relation.

Theorem 3.9 *For all processes p, q and r it holds that:*

$$\begin{array}{ll} (1) & p \equiv p \\ (2) & p \equiv q \textit{ implies } q \equiv p \\ (3) & p \equiv q \textit{ and } q \equiv r \textit{ implies } p \equiv r \end{array}$$

Proof

(1) Since $\smile$ is an equivalence, it follows that $\{\!|p;\pi|\!\} \smile \{\!|p;\pi|\!\}$, and thereby it follows that $p \equiv p$.

(2) Assume that $p \equiv q$. Then $\{\!|p;\pi|\!\} \smile \{\!|q;\pi|\!\}$. Since $\smile$ is an equivalence it follows that $\{\!|q;\pi|\!\} \smile \{\!|p;\pi|\!\}$, and thereby that $q \equiv p$.

(3) Assume that $p \equiv q$ and $q \equiv r$. Then $\{\!|p;\pi|\!\} \smile \{\!|q;\pi|\!\}$ and $\{\!|q;\pi|\!\} \smile \{\!|r;\pi|\!\}$. Since $\smile$ is an equivalence it follows that $\{\!|p;\pi|\!\} \smile \{\!|r;\pi|\!\}$, and thereby that $p \equiv r$.

■

Also, it is important to note that $\equiv$ is indeed included in $\sim$, (a fact which we shall exploit heavily in future proofs):

Lemma 3.10 (Inclusion Lemma) *For all processes p_1, p_2,*

$$\textit{Whenever } p_1 \equiv p_2 \textit{ then } \{\!|p_1|\!\} \smile \{\!|p_2|\!\}$$

Proof

We will show that the following relation S is a strong bisimulation:

$$S = \{(\{\!|p_1|\!\} \cup R_1, \{\!|p_2|\!\} \cup R_2) \mid \{\!|p_1;\pi|\!\} \cup R_1 \smile \{\!|p_2;\pi|\!\} \cup R_2\}$$

We consider the pair $(\{\!|p_1|\!\} \cup R_1, \{\!|p_2|\!\} \cup R_2)$. Note that, as a condition for being in S, $\{\!|p_1;\pi|\!\} \cup R_1 \smile \{\!|p_2;\pi|\!\} \cup R_2$.

Assume that $\{\!|p_1|\!\} \cup R_1 \xrightarrow{\alpha} \{\!|p_1'|\!\} \cup R_1'$. Then we can also conclude that $\{\!|p_1;\pi|\!\} \cup R_1 \xrightarrow{\alpha} \{\!|p_1';\pi|\!\} \cup R_1'$. Note that $\alpha \neq \pi$. Since $\{\!|p_1;\pi|\!\} \cup R_1 \smile \{\!|p_2;\pi|\!\} \cup R_2$ it follows that $\{|p_2;\pi|\} \cup R_2 \xrightarrow{\alpha} \{|p_2';\pi|\} \cup R_2'$ where $\{|p_1';\pi|\} \cup R_1' \smile \{|p_2';\pi|\} \cup R_2'$ and further $\{|p_2|\} \cup R_2 \xrightarrow{\alpha} \{|p_2'|\} \cup R_2'$. Since $\{|p_1';\pi|\} \cup R_1' \smile \{|p_2';\pi|\} \cup R_2'$ we have that $(\{|p_1'|\} \cup R_1', \{|p_2'|\} \cup R_2') \in S$.

A symmetric argument starting with $\{|p_2|\} \cup R_2$ ends the proof.

■

Now to the main result — $\equiv$ is a congruence, i.e. $\equiv$ is preserved by the operators of FC.

Theorem 3.11 (Congruence Property) *Assume for two processes p_1, p_2 that $p_1 \equiv p_2$. Then for any process q:*

$$\begin{aligned} \boldsymbol{fork}(p_1) &\equiv \boldsymbol{fork}(p_2) && (3.1)\\ p_1 + q &\equiv p_2 + q && (3.2)\\ q; p_1 &\equiv q; p_2 && (3.3)\\ p_1; q &\equiv p_2; q && (3.4) \end{aligned}$$

Proof

For each equation $t_1 \equiv t_2$ we shall prove $\{|t_1;\pi|\} \smile \{|t_2;\pi|\}$. Then the equation follows from definition 3.7. Note that our assumption is that $p_1 \equiv p_2$ and therefore that $\{|p_1;\pi|\} \smile \{|p_2;\pi|\}$.

(3.1) $\boldsymbol{fork}(p_1) \equiv \boldsymbol{fork}(p_2)$

We show that $\{|\boldsymbol{fork}(p_1);\pi|\} \smile \{|\boldsymbol{fork}(p_2);\pi|\}$ by showing that if one of the programs can perform an action, then the other program can perform the same action, and the two resulting programs are equivalent.

Assume that $\{|\boldsymbol{fork}(p_1);\pi|\} \xrightarrow{\tau} \{|\boldsymbol{nil};\pi, p_1|\}$ (note that no other move is possible for the left hand side of the equation). We also have that $\{|\boldsymbol{fork}(p_2);\pi|\} \xrightarrow{\tau} \{|\boldsymbol{nil};\pi, p_2|\}$. Since $p_1 \equiv p_2$, lemma 3.10 yields that $\{|p_1|\} \smile \{|p_2|\}$, but then $\{|\boldsymbol{nil};\pi, p_1|\} \smile \{|\boldsymbol{nil};\pi, p_2|\}$ due to the congruence property of $\smile$ (lemma 3.4).

The case where we start with $\{|\boldsymbol{fork}(p_2);\pi|\}$ is similar.

(3.2) $p_1 + q \equiv p_2 + q$

We shall show that $\{\!|(p_1+q);\pi|\!\} \smile \{\!|(p_2+q);\pi|\!\}$. Assume that $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\alpha}$. Then there are five cases.

Case 1 $Stop(p_1+q)$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$. But then $Stop(p_1)$ (as well as $Stop(q)$) so $\{\!|p_1;\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$. Then, since $\{\!|p_1;\pi|\!\} \smile \{\!|p_2;\pi|\!\}$, also $\{\!|p_2;\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$. In this way we know that also $Stop(p_2)$. This means that $\{\!|(p_2+q);\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$, and due to reflexivity of $\smile$, $\{\!|\boldsymbol{nil}|\!\} \smile \{\!|\boldsymbol{nil}|\!\}$.

Case 2 $q \overset{\alpha}{\hookrightarrow} q'$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\alpha} \{\!|q';\pi|\!\}$. Then also $\{\!|(p_2+q);\pi|\!\} \xrightarrow{\alpha} \{\!|q';\pi|\!\}$ and reflexivity yields $\{\!|q';\pi|\!\} \smile \{\!|q';\pi|\!\}$.

Case 3 $q \overset{\phi(r)}{\hookrightarrow} q'$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\tau} \{\!|q';\pi,r|\!\}$. Then also $\{\!|(p_2+q);\pi|\!\} \xrightarrow{\tau} \{\!|q';\pi,r|\!\}$ and reflexivity yields $\{\!|q';\pi,r|\!\} \smile \{\!|q';\pi,r|\!\}$.

Case 4 $p_1 \overset{\alpha}{\hookrightarrow} p_1'$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\alpha} \{\!|p_1';\pi|\!\}$. Then $\{\!|p_1;\pi|\!\} \xrightarrow{\alpha} \{\!|p_1';\pi|\!\}$. Then since $\{\!|p_1;\pi|\!\} \smile \{\!|p_2;\pi|\!\}$, also $\{\!|p_2;\pi|\!\} \xrightarrow{\alpha} R$ for some R where $\{\!|p_1';\pi|\!\} \smile R$. But then $\{\!|(p_2+q);\pi|\!\} \xrightarrow{\alpha} R$.

Case 5 $p_1 \overset{\phi(r)}{\hookrightarrow} p_1'$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\tau} \{\!|p_1';\pi,r|\!\}$. Then $\{\!|p_1;\pi|\!\} \xrightarrow{\tau} \{\!|p_1';\pi,r|\!\}$. Then since $\{\!|p_1;\pi|\!\} \smile \{\!|p_2;\pi|\!\}$, also $\{\!|p_2;\pi|\!\} \xrightarrow{\tau} R$ for some R where $\{\!|p_1';\pi,r|\!\} \smile R$. But then $\{\!|(p_2+q);\pi|\!\} \xrightarrow{\tau} R$.

This shows (1) of proposition 2.6, and (2) is similar.

(3.3) $q;p_1 \equiv q;p_2$

In the proofs of (3.1) and (3.2) we have used proposition 2.6 (from "right to left"). In the proof here (and in the proof of (3.4) that follows) we shall find a proper relation S and show that it is a bisimulation. We will show that the following relation S is a bisimulation:

$$S \;=\; \{(\{\!|q;p_1;\pi|\!\} \cup R, \{\!|q;p_2;\pi|\!\} \cup R) \mid p_1 \equiv p_2\} \cup \smile$$

We don't consider the pairs in $\smile$ (recall that S is defined to contain $\smile$) since $\smile$ is a bisimulation.

Let $(\{\!|q; p_1; \pi|\!\} \cup R, \{\!|q; p_2; \pi|\!\} \cup R) \in S$. Then we know that $p_1 \equiv p_2$ and therefore that $\{\!|p_1; \pi|\!\} \smile \{\!|p_2; \pi|\!\}$. Now there are two cases to analyse, depending on whether q is stopped or not.

Case 1 *Stop*(q). In this case we can prove that $\{\!|q; p_1; \pi|\!\} \smile \{\!|q; p_2; \pi|\!\}$, and that therefore $\{\!|q; p_1; \pi|\!\} \cup R \smile \{\!|q; p_2; \pi|\!\} \cup R$ due to the congruence property of $\smile$. But then the case is covered by the $\smile$ part of S.

To prove $\{\!|q; p_1; \pi|\!\} \smile \{\!|q; p_2; \pi|\!\}$ we proceed as follows. Assume that *Stop*(q) and $\{\!|q; p_1; \pi|\!\} \xrightarrow{\alpha} P_1$ for some P_1. Then also $\{\!|p_1; \pi|\!\} \xrightarrow{\alpha} P_1$ since it must be $p_1; \pi$ that moves. But since $\{\!|p_1; \pi|\!\} \smile \{\!|p_2; \pi|\!\}$, also $\{\!|p_2; \pi|\!\} \xrightarrow{\alpha} P_2$ for some P_2 where $P_1 \smile P_2$. But then $\{\!|q; p_2; \pi|\!\} \xrightarrow{\alpha} P_2$.

A symmetrical argument starting with $\{\!|q; p_2; \pi|\!\}$ ends the proof.

Case 2 $\neg$*Stop*(q). Assume that $\{\!|q; p_1; \pi|\!\} \cup R \xrightarrow{\alpha}$. Since $\neg$*Stop*(q) we know that α origins from q and (or) R, so we conclude that $\{\!|q; p_1; \pi|\!\} \cup R \xrightarrow{\alpha} \{\!|q'; p_1; \pi|\!\} \cup R'$. But then obviously $\{\!|q; p_2; \pi|\!\} \cup R \xrightarrow{\alpha} \{\!|q'; p_2; \pi|\!\} \cup R'$, and clearly $(\{\!|q'; p_1; \pi|\!\} \cup R', \{\!|q'; p_2; \pi|\!\} \cup R') \in S$

A symmetric argument starting with $\{\!|q; p_2; \pi|\!\} \cup R$ ends the proof.

(3.4) $p_1; q \equiv p_2; q$

We will show that the following relation S is a bisimulation:

$$
\begin{aligned}
S \stackrel{def}{=} \; & \{(\{\!|p_1; q; \pi|\!\} \cup R_1, \{\!|p_2; q; \pi|\!\} \cup R_2) \mid \\
& \quad \{\!|p_1; \pi|\!\} \cup R_1 \smile \{\!|p_2; \pi|\!\} \cup R_2\} \\
& \cup \smile
\end{aligned}
$$

We don't consider the pairs in $\smile$ (recall that S is defined to contain $\smile$) since $\smile$ is itself a bisimulation.

Let $(\{\!|p_1; q; \pi|\!\} \cup R_1, \{\!|p_2; q; \pi|\!\} \cup R_2) \in S$. Then we know that $\{\!|p_1; \pi|\!\} \cup R_1 \smile \{\!|p_2; \pi|\!\} \cup R_2$. We investigate two cases:

Case 1 *Stop*(p_1). In this case $\{\!|p_1; \pi|\!\} \cup R_1 \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_1$. Since $\{\!|p_1; \pi|\!\} \cup R_1 \smile \{\!|p_2; \pi|\!\} \cup R_2$ we then also know that $\{\!|p_2; \pi|\!\} \cup R_2 \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_2$ where

$\{|\boldsymbol{nil}|\} \cup R_1 \smile \{|\boldsymbol{nil}|\} \cup R_2$ and therefore $R_1 \smile R_2$. But then also $Stop(p_2)$. Since thus $Stop(p_1)$ and $Stop(p_2)$ and $R_1 \smile R_2$ we can easily conclude that $\{|p_1; q; \pi|\} \cup R_1 \smile \{|p_2; q; \pi|\} \cup R_2$ due to the congruence property of $\smile$. But then the pair belongs to the $\smile$ part of S and these pairs we need not check.

Case 2 $\neg Stop(p_1)$. Assume that $\{|p_1; q; \pi|\} \cup R_1 \xrightarrow{\alpha}$. Since $\neg Stop(p_1)$ we know that α origins from p_1 and (or) R_1, so we conclude that $\{|p_1; q; \pi|\} \cup R_1 \xrightarrow{\alpha} \{|p_1'; q; \pi|\} \cup R_1'$. But then $\{|p_1; \pi|\} \cup R_1 \xrightarrow{\alpha} \{|p_1'; \pi|\} \cup R_1'$. Since $\{|p_1; \pi|\} \cup R_1 \smile \{|p_2; \pi|\} \cup R_2$ we also have that $\{|p_2; \pi|\} \cup R_2 \xrightarrow{\alpha} \{|p_2'; \pi|\} \cup R_2'$ where $\{|p_1'; \pi|\} \cup R_1' \smile \{|p_2'; \pi|\} \cup R_2'$. But then $\{|p_2; q; \pi|\} \cup R_2 \xrightarrow{\alpha} \{|p_2'; q; \pi|\} \cup R_2'$ and finally $(\{|p_1'; q; \pi|\} \cup R_1', \{|p_2'; q; \pi|\} \cup R_2') \in S$.

A symmetric argument starting with $\{|p_2; q; \pi|\} \cup R_2$ ends the proof.

■

We are now finally able to state (and prove) the following main result:

Theorem 3.12 (Induced Congruence Property) $\equiv$ *is the largest congruence contained in* $\sim$.

Proof Using the previous theorem and lemma it is clear that $\equiv$ is a congruence included in $\sim$. To see that $\equiv$ is the largest such, assume that $\simeq$ is an arbitrary congruence included in $\sim$. Now assume $p \simeq q$. As $\simeq$ is assumed to be a congruence it follows that $p; \pi \simeq q; \pi$, and — as $\simeq$ is supposed to be included in $\sim$ — also $p; \pi \sim q; \pi$. However, then $p \equiv q$ according to the definition of $\equiv$ and it follows that $\simeq$ is included in $\equiv$. ■

3.3 Weak Congruence

We shall define a congruence relation, $\cong$, between processes, which is not sensitive to τ-transitions. The structure of the section follows the structure for the strong case: section 3.3.1 presents a first natural quess of an equivalence relation which, however, is not a congruence. Section 3.3.2 then presents a modification which is a congruence.

3.3.1 Process Equivalence

The purpose of this section is to define a weak equivalence relation $\approx \subseteq \mathcal{L} \times \mathcal{L}$ between FC processes, that ignores τ actions. For example, we want that $\tau; \alpha \approx$

α. For this purpose we shall, however, first define a weak equivalence relation $\asymp \subseteq \mathcal{M} \times \mathcal{M}$ between FC programs. We thus follow the same strategy as when we defined the strong process equivalence $\sim$ in terms of the strong program equivalence $\smile$.

Program Equivalence

We shall first define $\asymp$ in terms of the concept of weak bisimulation. We will then show that this relation is a congruence.

The semantics of programs is a labelled transition system $(\mathcal{M}, Act, \longrightarrow)$. As described in chapter 2, such a labelled transition system generates a weak bisimulation equivalence between the programs in $\mathcal{M}$. We shall let $\asymp$ denote this equivalence. Recall that $\asymp$ is defined as the union of all weak bisimulations. As an example of a weak bisimulation for our labelled transitions system, consider the one containing the pairs:

$$\begin{array}{rcl} \{\!|\tau;\alpha|\!\} & , & \{\!|\alpha|\!\} \\ \{\!|\boldsymbol{nil};\alpha|\!\} & , & \{\!|\alpha|\!\} \\ \{\!|\boldsymbol{nil}|\!\} & , & \{\!|\boldsymbol{nil}|\!\} \end{array}$$

This bisimulation contains the pair $(\{\!|\tau;\alpha|\!\}, \{\!|\alpha|\!\})$, so we can conclude that $\{\!|\tau;\alpha|\!\} \asymp \{\!|\alpha|\!\}$.

We shall now state and prove the property of $\asymp$ that it is a congruence. Note that we know $\asymp$ to be an equivalence. The congruence property is expressed formally as follows:

Lemma 3.13 ($\asymp$ is a congruence) *For all programs P_1, P_2 and Q,*

$$\textit{Whenever } P_1 \asymp P_2 \textit{ then } P_1 \cup Q \asymp P_2 \cup Q$$

Proof

We will show, that the following relation S is a weak bisimulation:

$$S \stackrel{def}{=} \{(P_1 \cup Q, P_2 \cup Q) \mid P_1 \asymp P_2\}$$

Assume that $(P_1 \cup Q, P_2 \cup Q) \in S$ and let $P_1 \cup Q \stackrel{\alpha}{\longrightarrow}$. There are three cases:

Case 1 $P_1 \xrightarrow{\alpha} P_1'$ and $P_1 \cup Q \xrightarrow{\alpha} P_1' \cup Q$. Then since $P_1 \asymp P_2$ we have $P_2 \overset{\alpha}{\Longrightarrow} P_2'$ where $P_1' \asymp P_2'$. So $P_2 \cup Q \overset{\alpha}{\Longrightarrow} P_2' \cup Q$ and $(P_1' \cup Q, P_2' \cup Q) \in S$.

Case 2 $Q \xrightarrow{\alpha} Q'$ and $P_1 \cup Q \xrightarrow{\alpha} P_1 \cup Q'$. Then $P_2 \cup Q \overset{\alpha}{\Longrightarrow} P_2 \cup Q'$ and $(P_1 \cup Q', P_2 \cup Q') \in S$.

Case 3 $\alpha = \tau$, $P1 \xrightarrow{c} P1'$, $Q \xrightarrow{rev(c)} Q'$ and $P1 \cup Q \xrightarrow{\tau} P1' \cup Q'$. Then since $P_1 \asymp P_2$ we have $P_2 \overset{c}{\Longrightarrow} P_2'$ where $P_1' \asymp P_2'$. So $P_2 \cup Q \overset{\tau}{\Longrightarrow} P_2' \cup Q'$ and $(P_1' \cup Q', P_2' \cup Q') \in S$.

A symmetric argument starting with $P_2 \cup Q \xrightarrow{\alpha}$ ends the proof.

■

Process Equivalence

In the previous section we introduced a weak equivalence on programs, and we showed it to be a congruence with respect to the basic operator on programs: $\cup$. In this section we shall come up with a weak process equivalence $\approx$, which in a natural way equates two processes if they behave the same but which, however, is not satisfactory due to lack of the congruence property. In the section to follow we will define an equivalence $\cong$ that is also a congruence.

The idea is the same as for $\sim$: two processes are equivalent, if they are equivalent when regarded as programs. Formally, we define $\approx$ as follows:

Definition 3.14 (Weak Process Equivalence) *The relation $\approx \subseteq \mathcal{L} \times \mathcal{L}$ is defined as:*

$$p \approx q \Leftrightarrow \{\!|p|\!\} \asymp \{\!|q|\!\}$$

Let us give an example. Earlier we argued that $\{\!|\tau;\alpha|\!\} \asymp \{\!|\alpha|\!\}$. We can therefore conclude that $\tau;\alpha \approx \alpha$.

The relation $\approx$ is an equivalence, as stated in the following proposition:

Proposition 3.15 (Weak Equivalence Property) *For all processes p, q and r it holds that:*

$$\begin{array}{ll} (1) & p \approx p \\ (2) & p \approx q \text{ implies } q \approx p \\ (3) & p \approx q \text{ and } q \approx r \text{ implies } p \approx r \end{array}$$

Proof

(1) Since $\asymp$ is an equivalence, it follows that $\{|p|\} \asymp \{|p|\}$, and thereby it follows that $p \approx p$.

(2) Assume that $p \approx q$. Then $\{|p|\} \asymp \{|q|\}$. Since $\asymp$ is an equivalence it follows that $\{|q|\} \asymp \{|p|\}$, and thereby that $q \approx p$.

(3) Assume that $p \approx q$ and $p \approx r$. Then $\{|p|\} \asymp \{|q|\}$ and $\{|q|\} \asymp \{|r|\}$. Since $\asymp$ is an equivalence it follows that $\{|p|\} \asymp \{|r|\}$, and thereby that $p \approx r$.

■

One can verify that $\approx$ is not a congruence with respect to the operators of FC in a similar way as it was verified that $\sim$ is not a congruence. For example, $\alpha \approx \textit{\textbf{fork}}(\alpha)$ but $\alpha; \beta \not\approx \textit{\textbf{fork}}(\alpha); \beta$. Although this example illustrates that $\approx$ is not a congruence, it does not really depend on the fact that τ is unobserved. An example that specifically focuses on the unobservability of τ actions, is the following (well known from process algebra): it holds that $\tau; \alpha \approx \alpha$, but $(\alpha + \beta) \not\approx ((\tau; \alpha) + \beta)$. The latter inequation is easily verifiable using the definitions — the rightmost process can perform a τ action and go into a state where only α is possible. This (unobserved) τ transition cannot be matched by the leftmost process, which at any time will offer either an α or a β until one has been performed.

The ***fork*** example and the '+' example illustrate the two kinds of congruence problems that we are faced with. In the following chapter we shall focus on the problem with ***fork***, while the problem with '+' is not dealt with since it is well known how to fix it within traditional process algebra, see for example [Mil89].

3.3.2 Process Congruence

In this section we introduce a weak process equivalence $\cong$ being strictly finer than the previously given equivalence $\approx$. We prove that $\cong$ has the congruence property under certain conditions; i.e. $\cong$ is preserved by (nearly) all operators of FC. However, the problem with '+' is dealt with in terms of a condition on the property, and we thus do not get a pure congruence.

The fact that $\approx$ is not a congruence is illustrated by the previous example demonstrating the difference between the two processes α and $\textit{\textbf{fork}}(\alpha)$ when put into the context $\_; \beta$. The difference lies essentially in the ability of the action α to be in parallel with future computation, which is here represented by the action β. We proceed exactly as in the strong case and introduce the special event π to represent termination. The formal definition of the process equivalence $\cong$ is as follows:

Definition 3.16 (Weak process congruence $\cong$) *The relation $\cong \subseteq \mathcal{L} \times \mathcal{L}$ is defined as:*

$$p \cong q \Leftrightarrow \{\!|p; \pi|\!\} \asymp \{\!|q; \pi|\!\}$$

In the remainder of this section we will prove that $\cong$ is nearly a congruence. First of all, it is straightforward to show that $\cong$ is an equivalence relation.

Theorem 3.17 *For all processes p, q and r it holds that:*

$$\begin{array}{ll} (1) & p \cong p \\ (2) & p \cong q \textit{ implies } q \cong p \\ (3) & p \cong q \textit{ and } q \cong r \textit{ implies } p \cong r \end{array}$$

Proof

(1) Since $\asymp$ is an equivalence, it follows that $\{\!|p; \pi|\!\} \asymp \{\!|p; \pi|\!\}$, and thereby it follows that $p \cong p$.

(2) Assume that $p \cong q$. Then $\{\!|p; \pi|\!\} \asymp \{\!|q; \pi|\!\}$. Since $\asymp$ is an equivalence it follows that $\{\!|q; \pi|\!\} \asymp \{\!|p; \pi|\!\}$, and thereby that $q \cong p$.

(3) Assume that $p \cong q$ and $q \cong r$. Then $\{\!|p; \pi|\!\} \asymp \{\!|q; \pi|\!\}$ and $\{\!|q; \pi|\!\} \asymp \{\!|r; \pi|\!\}$. Since $\asymp$ is an equivalence it follows that $\{\!|p; \pi|\!\} \asymp \{\!|r; \pi|\!\}$, and thereby that $p \cong r$.

■

Also, it is important to note that $\cong$ is indeed included in $\approx$:

Lemma 3.18 (Weak Inclusion Lemma) *For all processes p_1, p_2,*

$$\textit{Whenever } p_1 \cong p_2 \textit{ then } \{\!|p_1|\!\} \asymp \{\!|p_2|\!\}$$

Proof

We will show that the following relation S is a bisimulation:

$$S = \{(\{\!|p_1|\!\} \cup R_1, \{\!|p_2|\!\} \cup R_2) \mid \{\!|p_1; \pi|\!\} \cup R_1 \asymp \{\!|p_2; \pi|\!\} \cup R_2\}$$

We consider the pair $(\{\!|p_1|\!\} \cup R_1, \{\!|p_2|\!\} \cup R_2)$. Note that, as a condition for being in S, $\{\!|p_1;\pi|\!\} \cup R_1 \asymp \{\!|p_2;\pi|\!\} \cup R_2$.

Assume that $\{\!|p_1|\!\} \cup R_1 \xrightarrow{\alpha} \{\!|p_1'|\!\} \cup R_1'$. Then we can also conclude that $\{\!|p_1;\pi|\!\} \cup R_1 \xrightarrow{\alpha} \{\!|p_1';\pi|\!\} \cup R_1'$. Since $\{\!|p_1;\pi|\!\} \cup R_1 \asymp \{\!|p_2;\pi|\!\} \cup R_2$ it follows that $\{\!|p_2;\pi|\!\} \cup R_2 \stackrel{\alpha}{\Longrightarrow} \{\!|p_2';\pi|\!\} \cup R_2'$ where $\{\!|p_1';\pi|\!\} \cup R_1' \asymp \{\!|p_2';\pi|\!\} \cup R_2'$ and further $\{\!|p_2|\!\} \cup R_2 \stackrel{\alpha}{\Longrightarrow} \{\!|p_2'|\!\} \cup R_2'$. Since $\{\!|p_1';\pi|\!\} \cup R_1' \asymp \{\!|p_2';\pi|\!\} \cup R_2'$ we have that $(\{\!|p_1'|\!\} \cup R_1', \{\!|p_2'|\!\} \cup R_2') \in S$.

A symmetric argument starting with $\{\!|p_2|\!\} \cup R_2$ ends the proof.

■

Before we prove that $\cong$ is a congruence, we need to introduce the concept of a stable process.

Definition 3.19 (Stable process) *A process p is stable, iff:*

$$\textit{Whenever } p \stackrel{l}{\hookrightarrow} p' \textit{ for some } l \textit{ and } p', \textit{ then } l \in \mathit{Com}$$

So a process is stable if it can only start with a communication $a?$ or $a!$. That is, it cannot start with a τ nor with a $\phi(\ldots)$ – note that fork actions give rise to τ actions at the program level.

Now to the main result — $\cong$ is nearly a congruence, i.e. $\cong$ is preserved by nearly all the operators of FC.

Theorem 3.20 (Weak congruence property) *Assume for two processes p_1, p_2 that $p_1 \cong p_2$. Then for any process q:*

$$\begin{aligned} \textbf{\textit{fork}}(p_1) &\cong \textbf{\textit{fork}}(p_2) && (3.5)\\ p_1 + q &\cong p_2 + q \qquad \text{provided } p_1 \text{ and } p_2 \text{ are both stable} && (3.6)\\ q; p_1 &\cong q; p_2 && (3.7)\\ p_1; q &\cong p_2; q && (3.8) \end{aligned}$$

Proof

For each equation $t_1 \cong t_2$ we shall prove $\{\!|t_1;\pi|\!\} \asymp \{\!|t_2;\pi|\!\}$. Then the equation follows from definition 3.16. Note that our assumption is that $p_1 \cong p_2$ and therefore that $\{\!|p_1;\pi|\!\} \asymp \{\!|p_2;\pi|\!\}$.

(3.5) $\boldsymbol{fork}(p_1) \cong \boldsymbol{fork}(p_2)$

We show that $\{\!|\boldsymbol{fork}(p_1);\pi|\!\} \asymp \{\!|\boldsymbol{fork}(p_2);\pi|\!\}$ by showing that if one of the programs can perform an action, then the other program can perform the same action, and the two resulting programs are equivalent.

Assume that $\{\!|\boldsymbol{fork}(p_1);\pi|\!\} \xrightarrow{\tau} \{\!|\boldsymbol{nil};\pi,p_1|\!\}$ (note that no other move is possible for the left hand side of the equation). We also have that $\{\!|\boldsymbol{fork}(p_2);\pi|\!\} \stackrel{\tau}{\Longrightarrow} \{\!|\boldsymbol{nil};\pi,p_2|\!\}$. Since $p_1 \cong p_2$, lemma 3.18 yields that $\{\!|p_1|\!\} \asymp \{\!|p_2|\!\}$, but then $\{\!|\boldsymbol{nil};\pi,p_1|\!\} \asymp \{\!|\boldsymbol{nil};\pi,p_2|\!\}$ due to the congruence property of $\asymp$ (lemma 3.13).

The case where we start with $\{\!|\boldsymbol{fork}(p_2);\pi|\!\}$ is similar.

(3.6) $p_1 + q \cong p_2 + q$

We shall show that $\{\!|(p_1+q);\pi|\!\} \asymp \{\!|(p_2+q);\pi|\!\}$. Assume that $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\alpha}$. Then there are four cases.

Case 1 $Stop(p_1+q)$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$. But then $Stop(p_1)$ (as well as $Stop(q)$) so $\{\!|p_1;\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$. Then, since $\{\!|p_1;\pi|\!\} \asymp \{\!|p_2;\pi|\!\}$, also $\{\!|p_2;\pi|\!\} \stackrel{\pi}{\Longrightarrow} \{\!|\boldsymbol{nil}|\!\} \cup R$ for some R, where $\{\!|\boldsymbol{nil}|\!\} \asymp \{\!|\boldsymbol{nil}|\!\} \cup R$. This means that $\{\!|(p_2+q);\pi|\!\} \stackrel{\pi}{\Longrightarrow} \{\!|\boldsymbol{nil}|\!\} \cup R$.

Case 2 $q \stackrel{\alpha}{\hookrightarrow} q'$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\alpha} \{\!|q';\pi|\!\}$. Then also $\{\!|(p_2+q);\pi|\!\} \stackrel{\alpha}{\Longrightarrow} \{\!|q';\pi|\!\}$ and reflexivity yields $\{\!|q';\pi|\!\} \asymp \{\!|q';\pi|\!\}$.

Case 3 $q \stackrel{\phi(r)}{\hookrightarrow} q'$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\tau} \{\!|q';\pi,r|\!\}$. Then also $\{\!|(p_2+q);\pi|\!\} \stackrel{\tau}{\Longrightarrow} \{\!|q';\pi,r|\!\}$ and reflexivity yields $\{\!|q';\pi,r|\!\} \asymp \{\!|q';\pi,r|\!\}$.

Case 4 $p_1 \stackrel{c}{\hookrightarrow} p_1'$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{c} \{\!|p_1';\pi|\!\}$. Then $\{\!|p_1;\pi|\!\} \xrightarrow{c} \{\!|p_1';\pi|\!\}$. Then since $\{\!|p_1;\pi|\!\} \asymp \{\!|p_2;\pi|\!\}$, also $\{\!|p_2;\pi|\!\} \stackrel{c}{\Longrightarrow} R$ for some R where $\{\!|p_1';\pi|\!\} \asymp R$. But then $\{\!|(p_2+q);\pi|\!\} \stackrel{c}{\Longrightarrow} R$.

This shows (1) of proposition 2.12, and (2) is similar.

(3.7) $q;p_1 \cong q;p_2$

We will show that the following relation S is a weak bisimulation:

$$S \quad = \quad \{(\{\!|q;p_1;\pi|\!\} \cup R, \{\!|q;p_2;\pi|\!\} \cup R) \mid p_1 \cong p_2\} \cup \asymp$$

We don't consider the pairs in $\asymp$ (recall that S is defined to contain $\asymp$) since $\asymp$ is a weak bisimulation.

Let $(\{\!|q;p_1;\pi|\!\} \cup R, \{\!|q;p_2;\pi|\!\} \cup R) \in S$. Then we know that $p_1 \cong p_2$ and therefore that $\{\!|p_1;\pi|\!\} \asymp \{\!|p_2;\pi|\!\}$. Now there are two cases to analyse, depending on whether q is stopped or not.

Case 1 $\mathit{Stop}(q)$. In this case we can prove that $\{\!|q;p_1;\pi|\!\} \asymp \{\!|q;p_2;\pi|\!\}$, and that therefore $\{\!|q;p_1;\pi|\!\} \cup R \asymp \{\!|q;p_2;\pi|\!\} \cup R$ due to the congruence property of $\asymp$. But then the case is covered by the $\asymp$ part of S.

To prove $\{\!|q;p_1;\pi|\!\} \asymp \{\!|q;p_2;\pi|\!\}$ we proceed as follows. Assume that $\mathit{Stop}(q)$ and $\{\!|q;p_1;\pi|\!\} \xrightarrow{\alpha} P_1$ for some P_1. Then also $\{\!|p_1;\pi|\!\} \xrightarrow{\alpha} P_1$ since it must be $p_1;\pi$ that moves. But since $\{\!|p_1;\pi|\!\} \asymp \{\!|p_2;\pi|\!\}$, also $\{\!|p_2;\pi|\!\} \xRightarrow{\alpha} P_2$ for some P_2 where $P_1 \asymp P_2$. But then $\{\!|q;p_2;\pi|\!\} \xRightarrow{\alpha} P_2$.

A symmetrical argument starting with $\{\!|q;p_2;\pi|\!\}$ ends the proof.

Case 2 $\neg\mathit{Stop}(q)$. Assume that $\{\!|q;p_1;\pi|\!\} \cup R \xrightarrow{\alpha}$. Since $\neg\mathit{Stop}(q)$ we know that α origins from q and (or) R, so we conclude that $\{\!|q;p_1;\pi|\!\} \cup R \xrightarrow{\alpha} \{\!|q';p_1;\pi|\!\} \cup R'$. But then obviously $\{\!|q;p_2;\pi|\!\} \cup R \xRightarrow{\alpha} \{\!|q';p_2;\pi|\!\} \cup R'$, and clearly $(\{\!|q';p_1;\pi|\!\} \cup R', \{\!|q';p_2;\pi|\!\} \cup R') \in S$

A symmetric argument starting with $\{\!|q;p_2;\pi|\!\} \cup R$ ends the proof.

(3.8) $p_1;q \cong p_2;q$

We will show that the following relation S is a weak bisimulation:

$$\begin{aligned} S \overset{def}{=} \; & \{(\{\!|p_1;q;\pi|\!\} \cup R_1, \{\!|p_2;q;\pi|\!\} \cup R_2) \mid \\ & \quad \{\!|p_1;\pi|\!\} \cup R_1 \asymp \{\!|p_2;\pi|\!\} \cup R_2\} \\ & \cup \asymp \end{aligned}$$

We don't consider the pairs in $\asymp$ (recall that S is defined to contain $\asymp$) since $\asymp$ is itself a weak bisimulation.

Let $(\{\!|p_1;q;\pi|\!\} \cup R_1, \{\!|p_2;q;\pi|\!\} \cup R_2) \in S$. Then we know that $\{\!|p_1;\pi|\!\} \cup R_1 \asymp \{\!|p_2;\pi|\!\} \cup R_2$. Assume that $\{\!|p_1;q;\pi|\!\} \cup R_1 \xrightarrow{\alpha} P_1$.

We investigate two cases:

Case 1 $\mathit{Stop}(p_1)$. Then $\{\!|p_1;\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$ and therefore $\{\!|p_1;\pi|\!\} \cup R_1 \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_1$. But since $\{\!|p_1;\pi|\!\} \cup R_1 \asymp \{\!|p_2;\pi|\!\} \cup R_2$, also $\{\!|p_2;\pi|\!\} \cup R_2 \stackrel{\pi}{\Longrightarrow} \{\!|\boldsymbol{nil}|\!\} \cup R_3$ where $\{\!|\boldsymbol{nil}|\!\} \cup R_1 \asymp \{\!|\boldsymbol{nil}|\!\} \cup R_3$. By applying property (5) of lemma B.1, and using transitivity of $\asymp$, we get that $R_1 \asymp R_3$. Now, since $\{\!|p_2;\pi|\!\} \cup R_2 \stackrel{\pi}{\Longrightarrow} \{\!|\boldsymbol{nil}|\!\} \cup R3$ we have that $\{\!|p_2|\!\} \cup R_2 \stackrel{\tau}{\Longrightarrow} \{\!|\boldsymbol{nil}|\!\} \cup R_3$ and therefore that $\{\!|p_2;q;\pi|\!\} \cup R_2 \stackrel{\tau}{\Longrightarrow} \{\!|\boldsymbol{nil};q;\pi|\!\} \cup R_3$.

Now since $\mathit{Stop}(p_1)$ and since $R_1 \asymp R_3$, we can easily verify that $\{\!|p_1;q;\pi|\!\} \cup R_1 \asymp \{\!|\boldsymbol{nil};q;\pi|\!\} \cup R_3$. So since $\{\!|p_1;q;\pi|\!\} \cup R_1 \xrightarrow{\alpha} P_1$ we conclude that $\{\!|\boldsymbol{nil};q;\pi|\!\} \cup R_3 \stackrel{\alpha}{\Longrightarrow} P_2$ where $P_1 \asymp P_2$. But then $\{\!|p_2;q;\pi|\!\} \cup R_2 \stackrel{\alpha}{\Longrightarrow} P_2$, and $(P_1, P_2) \in S$ since S includes $\asymp$.

Case 2 $\neg \mathit{Stop}(p_1)$. Since $\neg \mathit{Stop}(p_1)$ we know that α origins from p_1 and (or) R_1, so we conclude that $\{\!|p_1;q;\pi|\!\} \cup R_1 \xrightarrow{\alpha} \{\!|p_1';q;\pi|\!\} \cup R_1'$. But then $\{\!|p_1;\pi|\!\} \cup R_1 \xrightarrow{\alpha} \{\!|p_1';\pi|\!\} \cup R_1'$. Since $\{\!|p_1;\pi|\!\} \cup R_1 \asymp \{\!|p_2;\pi|\!\} \cup R_2$ we also have that $\{\!|p_2;\pi|\!\} \cup R_2 \stackrel{\alpha}{\Longrightarrow} \{\!|p_2';\pi|\!\} \cup R_2'$ where $\{\!|p_1';\pi|\!\} \cup R_1' \asymp \{\!|p_2';\pi|\!\} \cup R_2'$. But then $\{\!|p_2;q;\pi|\!\} \cup R_2 \stackrel{\alpha}{\Longrightarrow} \{\!|p_2';q;\pi|\!\} \cup R_2'$ and finally $(\{\!|p_1';q;\pi|\!\} \cup R_1', \{\!|p_2';q;\pi|\!\} \cup R_2') \in S$.

A symmetric argument starting with $\{\!|p_2;q;\pi|\!\} \cup R_2$ ends the proof.

■

We are able to state (and prove) the following:

Theorem 3.21 (Induced Congruence Property) $\cong$ *contains any congruence contained in* $\approx$*.*

Proof Assume that $\simeq$ is an arbitrary congruence included in $\approx$. Now assume $p \simeq q$. As $\simeq$ is assumed to be a congruence it follows that $p;\pi \simeq q;\pi$, and — as $\simeq$ is supposed to be included in $\approx$ — also $p;\pi \approx q;\pi$. However, then $p \cong q$ according to the definition of $\cong$ and it follows that $\simeq$ is included in $\cong$. ■

Chapter 4

Reasoning about Processes

This chapter presents two different theories for reasoning about processes. The first theory, presented in section 4.1, establishes a way of performing equational reasoning. The theory is basically an axiomatisation of FC, in terms of which one can determine if two process terms are congruent. The second theory, presented in section 4.2, provides a modal logic for FC. A language of formulae is defined, together with a satisfaction relation between processes and formulae.

These two ways of reasoning about processes are both present for example in the Concurrency Workbench [CPS93], an automated verification tool for CCS.

4.1 Axiomatisation of Strong Equivalence

In this section we present a sound and complete axiomatisation for the process congruence $\equiv$ presented in definition 3.7. The completeness of the axiomatisation is obtained in a classical manner through the use of normal forms — being process terms with a very restricted use of the ***fork***–operator. More precisely, the axiomatisation consists of a collection of basic equational axioms, and two expansion laws. The basic axioms achieve completeness for normal form processes, and the expansion laws (together with the basic axioms) enable arbitrary process terms to be transformed into normal form, thus yielding completeness for the full calculus.

In more classical process calculi, expansion laws allow parallel composition to be replaced by non-determinism. In our calculus, the expansion laws will have an analogous purpose, namely that of replacing as much as possible forking with non–deterministic choice. However — as we shall see in the next section — our calculus seems to lack expressive power for suitable expansion laws to exist. To overcome this problem, we shall extend the calculus with an extra operator to be described below [1].

4.1.1 Searching for an Expansion Law

Axioms involving only the operators '+', '***nil***' and ';' can easily be verified (replacing = with $\equiv$); for example:

$$\begin{array}{rcl} p+q & = & q+p \\ \boldsymbol{nil}; p & = & p \end{array}$$

We will, however, also need more interesting axioms involving the ***fork*** operator. Basically, we need an expansion law, that describes how forking may be expanded into non-determinism — similar to the expansion law of CCS that expands parallel composition into nondeterminism. For example, consider the following instance of the CCS expansion law:

$$a.\mathbf{o} \mid b.\mathbf{o} \quad = \quad a.b.\mathbf{o} + b.a.\mathbf{o}$$

Let us try to search for a similar expansion law for the FC ***fork*** operator. We will simplify the problem and just search for an expansion of a simple process, similar to what we did for the CCS term $a.\mathbf{o} \mid b.\mathbf{o}$ above. We will thus try to expand the corresponding FC–process $\boldsymbol{fork}(\alpha); \beta$. An initial guess can be the following:

$$\boldsymbol{fork}(\alpha); \beta \quad = \quad \alpha; \beta + \beta; \alpha$$

However, this equation is not sound (it does not hold when replacing = with $\equiv$), and can therefore immediately be ignored. Intuitively, the reason is that the right hand side contains no information about α's capability to be in parallel with "the rest of the program". In the next seemingly correct equation we try to take this into account:

$$\boldsymbol{fork}(\alpha); \beta = \alpha; \beta + \beta; \boldsymbol{fork}(\alpha) \tag{4.1}$$

This equation is not sound either: the left hand side can perform a τ action as the first thing, while the right hand side can perform either an α action or a β action (note that since we want to axiomatise strong process congruence $\equiv$, τ actions are observable). Trying to repair this problem we obtain:

$$\boldsymbol{fork}(\alpha); \beta \quad = \quad \tau; (\alpha; \beta + \beta; \boldsymbol{fork}(\alpha)) \tag{4.2}$$

[1] This phenomenon is similar to the necessity of the leftmerge operator in PL in order to obtain a finite sound and complete equational axiomatisation as shown in [Mol90].

This equation is not sound either: the left hand side can only perform one τ action, while the right hand side can perform two (the explicitly mentioned and the one caused by ***fork***).

So it seems that we cannot in general expand forking! This leaves open the problem of how to establish equality between process terms involving the ***fork***-operator. The latter two attempts above suggest that we need a version of ***fork*** which is instantaneous without the initial internal transition caused by ***fork***. Put alternatively, we lack the ability to express that something *has been* forked, with the initial τ action already having occurred previously. To gain this expressivity, we add a new instantaneous forking operator to our calculus. We call this operator ***forked***, and we write ***forked***(p) to mean 'fork p without a τ'. One can regard this operator alternatively by reading ***forked***(p) as 'p has been forked', or shorter: 'forked p'. The latter reading suggests the following relationship between ***fork*** and ***forked***:

$$\boldsymbol{fork}(p) \quad = \quad \tau; \boldsymbol{forked}(p)$$

Let us now try to write new versions of the equations (4.1) and (4.2), that contain the ***forked*** operator in appropriate positions. These equations can be shown sound on basis of the formal definition we give of the ***forked*** operator in the following section. The equations are:

$$\begin{aligned} \boldsymbol{forked}(\alpha); \beta &= \alpha; \beta + \beta; \boldsymbol{forked}(\alpha) \\ \boldsymbol{fork}(\alpha); \beta &= \tau; (\alpha; \beta + \beta; \boldsymbol{forked}(\alpha)) \end{aligned}$$

However, it remains to settle whether it is possible to find an operator ***forked*** satisfying the above equations. In the next section we give an affirmative answer by providing an operational semantics for this new operator.

4.1.2 Adding the 'Forked'-operator

The syntax is extended with the ***forked***(p) alternative:

$$p \quad ::= \quad \boldsymbol{nil} \mid \alpha \mid p_1 + p_2 \mid p_1; p_2 \mid \boldsymbol{fork}(p) \mid \boldsymbol{forked}(p) \mid N$$

The language generated is referred to as $\mathcal{L}^\Phi$. The domain of $\mathcal{L}^\Phi$ multisets is called $\mathcal{M}^\Phi$, and the extended calculus is called FC^Φ.

The set *Lab* of actions that a process can perform is extended accordingly:

$$Lab \quad = \quad Act \cup \{\phi(p) \mid p \in \mathcal{L}^\Phi\} \cup \{\Phi(p) \mid p \in \mathcal{L}^\Phi\}$$

The new (local) semantics of processes is obtained by adding the following rule:

$$(\textbf{Forked}) \quad \frac{}{\boldsymbol{forked}(p) \overset{\Phi(p)}{\hookrightarrow} \boldsymbol{nil}}$$

Similarly, the new (global) semantics of programs is obtained by adding the following rule:

$$(\textbf{Forked}^{\{\|\}}) \quad \frac{p \overset{\Phi(q)}{\hookrightarrow} p', \{\!|p', q|\!\} \overset{\alpha}{\longrightarrow} R}{\{\!|p|\!\} \overset{\alpha}{\longrightarrow} R}$$

The previous semantic definitions from section 3.1 — the sets *Com* and *Act* (however not *Lab*), and the operational semantic rules for the original operators — carry over unchanged to the extended calculus. Likewise do the definitions of the program equivalence $\smile$ and the process equivalences $\sim$ and $\equiv$ from sections 3.2.1 and 3.2.2. The proofs of the following properties also carry over: (1) $\smile$ is a congruence with respect to $\cup$, (2) $\sim$ is an equivalence (but not a congruence, as before), (3) $\equiv$ is an equivalence.

The congruence property of $\equiv$ does, however, not apply to the extended calculus. The reason for this can be illustrated by the following example. It is easily shown that $\boldsymbol{forked}(\boldsymbol{nil}) \equiv \boldsymbol{nil}$. That is, according to our definition of $\equiv$ it holds since $\{\!|\boldsymbol{forked}(\boldsymbol{nil}); \pi|\!\} \smile \{\!|\boldsymbol{nil}; \pi|\!\}$, which is easily verifiable: the only action that any of the two programs can perform is π. Now, consider the two processes $\boldsymbol{forked}(nil) + \alpha$ and $\boldsymbol{nil} + \alpha$. In order for $\equiv$ to be a congruence, these two processes should be equivalent (show the same behaviour). They are, however, not equivalent since the following programs are not: $\{\!|(\boldsymbol{forked}(\boldsymbol{nil}) + \alpha); \pi|\!\}$ and $\{\!|(\boldsymbol{nil} + \alpha); \pi|\!\}$. If we consider the first program, we can deduce the following behaviour:

$$(\textbf{Forked}^{\{\|\}}) \quad \frac{}{\boldsymbol{forked}(\boldsymbol{nil}) \overset{\Phi(\boldsymbol{nil})}{\hookrightarrow} \boldsymbol{nil}}$$

$$(\textbf{Choice}_2) \quad \frac{}{(\boldsymbol{forked}(\boldsymbol{nil}) + \alpha) \overset{\Phi(\boldsymbol{nil})}{\hookrightarrow} \boldsymbol{nil}}$$

$$(\textbf{Sequence}_1) \quad \frac{}{(\boldsymbol{forked}(\boldsymbol{nil}) + \alpha); \pi \overset{\Phi(\boldsymbol{nil})}{\hookrightarrow} \boldsymbol{nil}; \pi}$$

This deduction we can combine with the obvious fact that $\{\!|\boldsymbol{nil};\pi,\boldsymbol{nil}|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\}$:

$$(\textbf{Forked}^{\{\!|\,|\!\}})\quad \frac{(\boldsymbol{forked}(\boldsymbol{nil})+\alpha);\pi \overset{\Phi(\boldsymbol{nil})}{\hookrightarrow} \boldsymbol{nil};\pi\ ,\ \{\!|\boldsymbol{nil};\pi,\boldsymbol{nil}|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\}}{\{\!|(\boldsymbol{forked}(\boldsymbol{nil})+\alpha);\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\}}$$

The program $\{\!|(\boldsymbol{nil}+\alpha);\pi|\!\}$ can, however only perform a π action after having performed an α action, so the two programs are not equivalent. What happens in the first program is that the effect of $\boldsymbol{forked}(\boldsymbol{nil})$ is to bypass the α action, an effect that $\boldsymbol{nil}$ cannot have.

Rather than changing the definition of $\equiv$, we choose to loosen the requirement that $\equiv$ is a congruence. It is though nearly a congruence as stated in the following theorem, where the only deviation is concerned with the '$+$' operator and the new $\boldsymbol{forked}$ operator. Let $Active(q)$, for any process $q \in \mathcal{L}^\Phi$, mean that q can perform an action. That is: $Active(q) \overset{def}{=} \exists \alpha \in Act, P \in \mathcal{M}^\Phi \cdot \{\!|q|\!\} \xrightarrow{\alpha} P$. Then the loosened congruence property can be stated as follows:

Theorem 4.1 (Loosened Congruence Property) *Assume for two processes p_1, p_2 that $p_1 \equiv p_2$. Then for any process q:*

$$\begin{aligned}
\boldsymbol{fork}(p_1) &\equiv \boldsymbol{fork}(p_2) && (4.3)\\
\boldsymbol{forked}(p_1) &\equiv \boldsymbol{forked}(p_2) && (4.4)\\
p_1 + q &\equiv p_2 + q \quad \text{provided } Active(q) \Rightarrow (Stop(p_1) \Leftrightarrow Stop(p_2)) && (4.5)\\
q;p_1 &\equiv q;p_2 && (4.6)\\
p_1;q &\equiv p_2;q && (4.7)
\end{aligned}$$

The condition in equation (4.5) is motivated by our previous example, where q is α, and p_1 is $\boldsymbol{forked}(\boldsymbol{nil})$ and p_2 is $\boldsymbol{nil}$, thus the condition is not satisfied in this case. Note that $Stop(\boldsymbol{forked}(\boldsymbol{nil}))$ does not hold according the definition of $Stop$ which is unchanged for the extended calculus.

Proof

For each equation $t_1 \equiv t_2$ we shall prove $\{\!|t_1;\pi|\!\} \smile \{\!|t_2;\pi|\!\}$. Then the equation follows from definition 3.7. Note that our assumption is that $p_1 \equiv p_2$ and therefore that $\{\!|p_1;\pi|\!\} \smile \{\!|p_2;\pi|\!\}$.

In the proof we shall use an unlabelled transition relation $\Longrightarrow \subseteq \mathcal{M}^\Phi \times \mathcal{M}^\Phi$ which is the least relation closed under the following rules:

$$(1)\quad \frac{p \overset{\Phi(r)}{\hookrightarrow} p'}{P \cup \{\!|p|\!\} \Longrightarrow P \cup \{\!|p',r|\!\}}$$

$$(2) \qquad \frac{}{P \Longrightarrow P}$$

$$(3) \qquad \frac{P \Longrightarrow P' \,,\; P' \Longrightarrow P''}{P \Longrightarrow P''}$$

We shall use the $\Longrightarrow$ relation to write proofs of $P \xrightarrow{\alpha} Q$ transitions more conveniently. That is, suppose for example that we can prove $\{|p|\} \xrightarrow{\alpha} Q$ from the fact that $p \overset{\Phi(r)}{\hookrightarrow} p'$ and $\{|p', r|\} \xrightarrow{\alpha} Q$ by using the ($\mathbf{Forked}^{\{|\,|\}}$) rule. We shall typically write this as $\{|p|\} \Longrightarrow \{|p', r|\} \xrightarrow{\alpha} Q$. This notation becomes even more convenient in the case where several applications of the ($\mathbf{Forked}^{\{|\,|\}}$) rule are involved.

(4.3) $\boldsymbol{fork}(p_1) \equiv \boldsymbol{fork}(p_2)$

As the proof of theorem 3.11 (3.1).

(4.4) $\boldsymbol{forked}(p_1) \equiv \boldsymbol{forked}(p_2)$

We shall show that $\{|\boldsymbol{forked}(p_1); \pi|\} \smile \{|\boldsymbol{forked}(p_2); \pi|\}$.
Assume that $\{|\boldsymbol{forked}(p_1); \pi|\} \xrightarrow{\alpha} P_1$.

This transition can be split into an unlabelled and a labelled transition: $\{|\boldsymbol{forked}(p_1); \pi|\} \Longrightarrow \{|\boldsymbol{nil}; \pi, p_1|\} \xrightarrow{\alpha} P_1$. On the other hand we have that $\{|\boldsymbol{forked}(p_2); \pi|\} \Longrightarrow \{|\boldsymbol{nil}; \pi, p_2|\}$. Now, since $p_1 \equiv p_2$, lemma 3.10 yields that $\{|p_1|\} \smile \{|p_2|\}$, and due to congruence this implies that $\{|\boldsymbol{nil}; \pi, p_1|\} \smile \{|\boldsymbol{nil}; \pi, p_2|\}$. So since $\{|\boldsymbol{nil}; \pi, p_1|\} \xrightarrow{\alpha} P_1$ we therefore get that $\{|\boldsymbol{nil}; \pi, p_2|\} \xrightarrow{\alpha} P_2$ where $P_1 \smile P_2$. Finally we can conclude that $\{|\boldsymbol{forked}(p_2); \pi|\} \xrightarrow{\alpha} P_2$.

A symmetric argument starting with $\{|\boldsymbol{forked}(p_2); \pi|\}$ ends the proof.

(4.5) $p_1 + q \equiv p_2 + q$

We shall show that $\{\!|(p_1{+}q);\pi|\!\} \smile \{\!|(p_2{+}q);\pi|\!\}$ on the condition that $Active(q) \Rightarrow (Stop(p_1) \Leftrightarrow Stop(p_2))$.

Assume that $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\alpha} P_1$. Then there are 5 cases:

Case 1 $Stop(p_1+q)$ and $\{\!|(p_1+q);\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$. But then $Stop(p_1)$ (as well as $Stop(q)$) so $\{\!|p_1;\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$. Then, since $\{\!|p_1;\pi|\!\} \smile \{\!|p_2;\pi|\!\}$, also $\{\!|p_2;\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R$ for some R where $\{\!|\boldsymbol{nil}|\!\} \smile \{\!|\boldsymbol{nil}|\!\} \cup R$. Now there are two cases:

Case 1.1 $Stop(p_2)$, in which case $\{\!|(p_2{+}q);\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$, and we are done.

Case 1.2 $p_2 \overset{\Phi(r)}{\hookrightarrow} p_2'$. In this case $\{\!|p_2;\pi|\!\} \Longrightarrow \{\!|p_2';\pi|\!\} \cup \{\!|r|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R$, so $\{\!|(p_2+q);\pi|\!\} \Longrightarrow \{\!|p_2';\pi|\!\} \cup \{\!|r|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R$, and since we earlier concluded that $\{\!|\boldsymbol{nil}|\!\} \smile \{\!|\boldsymbol{nil}|\!\} \cup R$ we are done.

Case 2 $q \overset{\alpha}{\hookrightarrow} q'$ or $q \overset{\phi(r)}{\hookrightarrow} q'$. As the proof of theorem 3.11 (3.2) cases 2 and 3.

Case 3 $q \overset{\Phi(r)}{\hookrightarrow} q'$. In this case $(p_1+q);\pi \overset{\Phi(r)}{\hookrightarrow} q';\pi$ and therefore $\{\!|(p_1+q);\pi|\!\} \Longrightarrow \{\!|q';\pi|\!\} \cup \{\!|r|\!\} \xrightarrow{\alpha} P_1$. Then also $\{\!|(p_2+q);\pi|\!\} \Longrightarrow \{\!|q';\pi|\!\} \cup \{\!|r|\!\} \xrightarrow{\alpha} P_1$ and reflexivity yields $P_1 \smile P_1$.

Case 4 $p_1 \overset{\alpha}{\hookrightarrow} p_1'$ or $p_1 \overset{\phi(r)}{\hookrightarrow} p_1'$. As the proof of theorem 3.11 (3.2) cases 4 and 5.

Case 5 $p_1 \overset{\Phi(r)}{\hookrightarrow} p_1'$. In this case $\{\!|(p_1+q);\pi|\!\} \Longrightarrow \{\!|p_1';\pi|\!\} \cup \{\!|r|\!\} \xrightarrow{\alpha} P_1$. In particular $\{\!|p_1;\pi|\!\} \Longrightarrow \{\!|p_1';\pi|\!\} \cup \{\!|r|\!\} \xrightarrow{\alpha} P_1$. Since $\{\!|p_1;\pi|\!\} \smile \{\!|p_2;\pi|\!\}$ we then have that $\{\!|p_2;\pi|\!\} \xrightarrow{\alpha} P_2$ where $P_1 \smile P_2$.

Now, there are two cases to consider:

Case 5.1 $\alpha \neq \pi$. In this case it is p_2 that performs the action α and therefore $\{\!|(p_2+q);\pi|\!\} \xrightarrow{\alpha} P_2$.

Case 5.2 $\alpha = \pi$. In this case we have that $\{\!|(p_1+q);\pi|\!\} \Longrightarrow \{\!|p_1';\pi|\!\} \cup \{\!|r|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_1$. More specifically $\{\!|p_1;\pi|\!\} \Longrightarrow \{\!|p_1';\pi|\!\} \cup \{\!|r|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_1$. Since $\{\!|p_1;\pi|\!\} \smile \{\!|p_2;\pi|\!\}$ we then have that $\{\!|p_2;\pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_2$ where $\{\!|\boldsymbol{nil}|\!\} \cup R_1 \smile \{\!|\boldsymbol{nil}|\!\} \cup R_2$.

There are two cases:

Case 5.2.1 $Stop(p_2)$. Since $Active(q) \Rightarrow (Stop(p_1) \Leftrightarrow Stop(p_2))$ and since $\neg Stop(p_1)$ and $Stop(p_2)$ we must have that $\neg Active(q)$. Further, since $Stop(p_2)$ we can easily verify that $\neg Active(p_1)$, and therefore that $R_1 \smile \{\!|\boldsymbol{nil}|\!\}$.
There are thus two cases:

Case 5.2.1.1 $Stop(q)$. Then $Stop(p_2 + q)$ and therefore $\{\!|(p_2 + q); \pi|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\}$.

Case 5.2.1.2 $q \overset{\Phi(s)}{\hookrightarrow} q'$. In this case $\{\!|(p_2 + q); \pi|\!\} \Longrightarrow \{\!|q'; \pi|\!\} \cup \{\!|s|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_3$. The π action is possible since we know that $\neg Active(q)$. Since $\neg Active(q)$ it must also hold that $\{\!|\boldsymbol{nil}|\!\} \cup R_1 \smile \{\!|\boldsymbol{nil}|\!\} \cup R_3$.

Case 5.2.2 $p_2 \overset{\Phi(s)}{\hookrightarrow} p_2'$. Then $\{\!|p_2; \pi|\!\} \Longrightarrow \{\!|p_2'; \pi|\!\} \cup \{\!|s|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_2$ and therefore $\{\!|(p_2 + q); \pi|\!\} \Longrightarrow \{\!|p_2'; \pi|\!\} \cup \{\!|s|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_2$.

A symmetric argument starting with $\{\!|(p_2 + q); \pi|\!\}$ ends the proof.

(4.6) $q; p_1 \equiv q; p_2$

We will show that the following relation S is a bisimulation:

$$S \;=\; \{(\{\!|q; p_1; \pi|\!\} \cup R, \{\!|q; p_2; \pi|\!\} \cup R) \mid p_1 \equiv p_2\} \cup \smile$$

Let $(\{\!|q; p_1; \pi|\!\} \cup R, \{\!|q; p_2; \pi|\!\} \cup R) \in S$ and assume that $\{\!|q; p_1; \pi|\!\} \cup R \xrightarrow{\alpha} P_1$. Now there are two cases to analyse, depending on whether q during the evaluation reaches a state q' where $Stop(q')$ (q gets exhausted) or not (note the similarity between these two cases and the two cases $Stop(q)$ versus $\neg Stop(q)$ in the proof of theorem 3.11 (3.3)).

Case 1: q gets exhausted Since q gets exhausted it must hold that $\{\!|q; p_1; \pi|\!\} \cup R \Longrightarrow \{\!|q'; p_1; \pi|\!\} \cup R' \xrightarrow{\alpha} P_1$ where $Stop(q')$. But then $\{\!|q; p_2; \pi|\!\} \cup R \Longrightarrow \{\!|q'; p_2; \pi|\!\} \cup R'$ and since $Stop(q')$ it is easy verifiable that $\{\!|q'; p_1; \pi|\!\} \smile \{\!|q'; p_2; \pi|\!\}$. Due to congruence it follows that $\{\!|q'; p_1; \pi|\!\} \cup R' \smile \{\!|q'; p_2; \pi|\!\} \cup R'$, and since $\{\!|q'; p_1; \pi|\!\} \cup R' \xrightarrow{\alpha} P_1$ we therefore get that $\{\!|q'; p_2; \pi|\!\} \cup R' \xrightarrow{\alpha} P_2$ where $P_1 \smile P_2$. Finally $\{\!|q; p_2; \pi|\!\} \cup R \xrightarrow{\alpha} P_2$ and $(P_1, P_2) \in S$.

Case 2: q does not get exhausted As the proof of theorem 3.11 (3.3) in the case $\neg Stop(q)$.

(4.7) $p_1; q \equiv p_2; q$

We will show that the following relation S is a bisimulation:

$$\begin{array}{rl} S \stackrel{def}{=} & \{(\{\!|p_1; q; \pi|\!\} \cup R_1, \{\!|p_2; q; \pi|\!\} \cup R_2) \mid \\ & \quad \{\!|p_1; \pi|\!\} \cup R_1 \smile \{\!|p_2; \pi|\!\} \cup R_2\} \\ & \cup \smile \end{array}$$

Let $(\{\!|p_1; q; \pi|\!\} \cup R_1, \{\!|p_2; q; \pi|\!\} \cup R_2) \in S$ and assume that $\{\!|p_1; q; \pi|\!\} \cup R_1 \xrightarrow{\alpha} P_1$. Now there are two cases to analyse, depending on whether p_1 during the evaluation reaches a state p_1' where $Stop(p_1')$ (p_1 gets exhausted) or not.

Case 1: p_1 gets exhausted Since p_1 gets exhausted it must hold that: $\{\!|p_1; q; \pi|\!\} \cup R_1 \Longrightarrow \{\!|p_1'; q; \pi|\!\} \cup R_1' \xrightarrow{\alpha} P_1$ where $Stop(p_1')$. But then $\{\!|p_1; \pi|\!\} \cup R_1 \Longrightarrow \{\!|p_1'; \pi|\!\} \cup R_1' \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_1'$. Since $\{\!|p_1; \pi|\!\} \cup R_1 \smile \{\!|p_2; \pi|\!\} \cup R_2$ it must then hold that $\{\!|p_2; \pi|\!\} \cup R_2 \Longrightarrow \{\!|p_2'; \pi|\!\} \cup R_2' \xrightarrow{\pi} \{\!|\boldsymbol{nil}|\!\} \cup R_2'$ where $Stop(p_2')$ and $R_1' \smile R_2'$. This means that $\{\!|p_2; q; \pi|\!\} \cup R_2 \Longrightarrow \{\!|p_2'; q; \pi|\!\} \cup R_2'$. Since $Stop(p_1')$ and $Stop(p_2')$ and since $R_1' \smile R_2'$ we conclude easily that $\{\!|p_1'; q; \pi|\!\} \cup R_1' \smile \{\!|p_2'; q; \pi|\!\} \cup R_2'$ and therefore since $\{\!|p_1'; q; \pi|\!\} \cup R_1' \xrightarrow{\alpha} P_1$ also $\{\!|p_2'; q; \pi|\!\} \cup R_2' \xrightarrow{\alpha} P_2$ where $P_1 \smile P_2$. But this means that $\{\!|p_2; q; \pi|\!\} \cup R_2 \xrightarrow{\alpha} P_2$ and $(P_1, P_2) \in S$.

Case 2: p_1 does not get exhausted As the proof of theorem 3.11 (3.4) in the case $\neg Stop(p_1)$.

A symmetric argument starting with $\{\!|p_2; q; \pi|\!\} \cup R_2$ ends the proof.

■

4.1.3 Basic Axioms and Inference Rules

The final axiom system $\mathcal{AS}$ consists of a set $\mathcal{AX}$ of equational axioms, two expansion laws $\mathcal{E}_1$ and $\mathcal{E}_2$, using normal forms, and a set $\mathcal{I}$ of inference rules. In this section we shall present the basic equational axioms $\mathcal{AX}$ and the inference rules $\mathcal{I}$.

Definition 4.2 (The basic axioms $\mathcal{AX}$) *The set $\mathcal{AX}$ contains the following axioms:*

$$p; \boldsymbol{nil} \;=\; p \tag{4.8}$$

$$\begin{aligned}
\textbf{\textit{nil}}; p &= p && (4.9)\\
(p; q); r &= p; (q; r) && (4.10)\\
(p + q); r &= p; r + q; r && (4.11)
\end{aligned}$$

$$\begin{aligned}
p + (q + r) &= (p + q) + r && (4.12)\\
p + q &= q + p && (4.13)\\
p + \textbf{\textit{nil}} &= p && (4.14)\\
p + p &= p && (4.15)
\end{aligned}$$

$$\begin{aligned}
\textbf{\textit{fork}}(p) &= \tau; \textbf{\textit{forked}}(p) && (4.16)\\
\textbf{\textit{forked}}((\alpha; \textbf{\textit{forked}}(p)) + q) &= \textbf{\textit{forked}}((\alpha; p) + q) && (4.17)\\
\textbf{\textit{forked}}(\textbf{\textit{nil}}) &= \textbf{\textit{nil}} && (4.18)
\end{aligned}$$

Proposition 4.3 ($\mathcal{AX}$ is sound) *The axioms $\mathcal{AX}$ are sound with respect to strong process congruence.*

Proof See in Appendix B, proposition B.2. ■

Definition 4.4 (The inference rules $\mathcal{I}$) *The inference rules $\mathcal{I}$ are:*

(**Reflexive**) $$\frac{}{p = p}$$

(**Symmetric**) $$\frac{p = q}{q = p}$$

(**Transitive**) $$\frac{p = q, q = r}{p = r}$$

(**Congruence$_1$**) $$\frac{p_1 = p_2}{\textbf{\textit{fork}}(p_1) = \textbf{\textit{fork}}(p_2)}$$

(**Congruence$_2$**) $$\frac{p_1 = p_2}{\textbf{\textit{forked}}(p_1) = \textbf{\textit{forked}}(p_2)}$$

(**Congruence$_3$**) $$\frac{p_1 = p_2}{p_1 + q = p_2 + q}\ {\scriptstyle Active(q) \Rightarrow (Stop(p_1) \Leftrightarrow Stop(p_2))}$$

(**Congruence$_4$**) $$\frac{p_1 = p_2}{q; p_1 = q; p_2}$$

(**Congruence$_5$**) $$\frac{p_1 = p_2}{p_1; q = p_2; q}$$

Proposition 4.5 ($\mathcal{I}$ is sound) *The inference rules $\mathcal{I}$ are sound with respect to strong process congruence.*

Proof The rules must hold when replacing $=$ with $\equiv$. For the three first equivalence rules, this is the case as stated in theorem 3.9 which also holds for the extended calculus. Concerning the remaining congruence rules, theorem 4.1 states this for the operators in the extended calculus. ∎

4.1.4 Normal Forms

Our way to completeness is classical in that it is based on *normal forms*. For processes in normal form, the basic axioms $\mathcal{A}X$ will be shown to be complete, and we shall in the next section present two expansion laws that will enable any process term to be transformed into a provable equivalent normal form process term.

Processes in normal form contain no applications of the ***fork*** operator, and applications of the ***forked*** operator occur last. One intuition behind this is that when observing a process we cannot observe the individual forkings; and as long as the original process has not terminated, we cannot observe which actions come from it and which actions come from the forked processes. We can, however, in contexts observe when the process terminates and what at that time has been forked. In principle a process is brought into normal form by first replacing applications $\boldsymbol{fork}(p)$ by $\tau;\boldsymbol{forked}(p)$ (applying axiom 4.16), and then by moving the applications of ***forked*** as much as possible "to the right". Note that we must deal with the ***forked*** operator in normal forms, since we must keep track of what is potentially in parallel with future computation.

As an example, consider the process $\boldsymbol{fork}(\alpha);\beta$. First, we can replace $\boldsymbol{fork}(\alpha)$ with $\tau;\boldsymbol{forked}(\alpha)$, thus obtaining $\tau;\boldsymbol{forked}(\alpha);\beta$. Then we can consider the process $\boldsymbol{forked}(\alpha);\beta$, and bring it into the form $\alpha;\beta+\beta;\boldsymbol{forked}(\alpha)$. As result we obtain $\tau;(\alpha;\beta+\beta;\boldsymbol{forked}(\alpha))$. We see that there is no application of the ***fork*** operator, and that the application of the ***forked*** operator occurs last.

We can also put some semantic light on normal forms. Consider a process N in normal form, and consider the transition tree having $\{|N|\}$, as root. Then this tree can be pictured as in figure 4.1.

Each path through the tree can be considered as consisting of two parts: a *before termination* part followed by a *potential in parallel with future computation* part. The latter part coming from an argument to the ***forked*** operator.

Let us now formally introduce normal forms:

Definition 4.6 (Normal form) *A process p is in normal form, if it is a term*

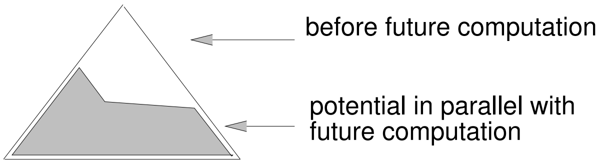

Figure 4.1: Computation Tree

in N, where:

$$\begin{aligned} N &::= \sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j) \\ B &::= \sum_k \beta_k; B_k \end{aligned}$$

A process p is in simple normal form, if it is a term in B.

Note that simple normal forms contain no applications of the ***forked*** operator. That is, once we have reached the termination of a process (second alternative in the definition of N), what remains has all been forked, and to express this we only need one application of the ***forked*** operator. That is, we do not care about any "further" forking.

4.1.5 Expansion Laws

The two expansion laws to be presented in this section will enable any process term to be transformed into normal form. To motivate the two expansion laws consider the case of a sequential composition of two process terms N and M already in normal form. Now assume N is of the following form:

$$N = \sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j)$$

Then, using the laws for distributing ; over + and laws of associativity for ;, we obtain:

$$\vdash N; M = \sum_i \alpha_i; (N_i; M) + \sum_j \beta_j; (\boldsymbol{forked}(B_j); M)$$

To enable this term to be transformed into a normal form we introduce the following expansion law:

Definition 4.7 (Expansion law $\mathcal{E}_1$)

Let:

$$A = \sum_i \alpha_i; A_i$$

$$\begin{aligned}
N &= \sum_j \beta_j; N_j \\
L &= \sum_k \gamma_k; \boldsymbol{forked}(C_k) \\
M &= N + L
\end{aligned}$$

*where M is not **nil**, then:*

$$\boldsymbol{forked}(A); M = A' + B' + C' + D' + E'$$

where:

$$\begin{aligned}
A' &= \sum_i \alpha_i; \boldsymbol{forked}(A_i); M \\
B' &= \sum_j \beta_j; \boldsymbol{forked}(A); N_j \\
C' &= \sum_k \gamma_k; \boldsymbol{forked}(A); \boldsymbol{forked}(C_k) \\
D' &= \sum_{\alpha_i = rev(\beta_j)} \tau; \boldsymbol{forked}(A_i); N_j \\
E' &= \sum_{\alpha_i = rev(\gamma_k)} \tau; \boldsymbol{forked}(A_i); \boldsymbol{forked}(C_k)
\end{aligned}$$

Proposition 4.8 ($\mathcal{E}_1$ is sound) *The expansion law $\mathcal{E}_1$ is sound with respect to strong process congruence.*

Proof

See Appendix B proposition B.3 ∎

As an example illustrating expansion law $\mathcal{E}_1$, we shall prove that $\vdash \boldsymbol{forked}(\alpha); \beta = \alpha; \beta + \beta; \boldsymbol{forked}(\alpha)$. As a motivation for proving this, recall that we in section 4.1.4 indicated how the process $\boldsymbol{fork}(\alpha); \beta$ could be brought into the normal form $\tau; (\alpha; \beta + \beta; \boldsymbol{forked}(\alpha))$. The first step was to apply axiom (4.16), thus obtaining $\tau; \boldsymbol{forked}(\alpha); \beta$. The next step was to bring $\boldsymbol{forked}(\alpha); \beta$ into the form $\alpha; \beta + \beta; \boldsymbol{forked}(\alpha)$, a step that we can now verify is legal:

$\boldsymbol{forked}(\alpha); \beta$

$=$

$\boldsymbol{forked}(\alpha; \boldsymbol{nil}); \beta; \boldsymbol{nil}$

– *axiom 4.8*

$=$

$\alpha; \boldsymbol{forked}(\boldsymbol{nil}); \beta; \boldsymbol{nil} + \beta; \boldsymbol{forked}(\alpha; \boldsymbol{nil}); \boldsymbol{nil}$

– *expansion law $\mathcal{E}_1$*

$=$

$\alpha; \beta + \beta; \boldsymbol{forked}(\alpha)$

– *axioms (4.8), (4.18)*

To be able to inductively transform $\boldsymbol{forked}(A); M$ into normal form, we see in definition 4.7 that we need to be able to handle terms of the form $\boldsymbol{forked}(A); \boldsymbol{forked}(B)$. This motivates the second expansion law below:

Definition 4.9 (Expansion law $\mathcal{E}_2$)

Let:

$$\begin{aligned} A &= \sum_i \alpha_i; A_i \\ B &= \sum_j \beta_j; B_j \end{aligned}$$

Then:

$$\boldsymbol{forked}(A); \boldsymbol{forked}(B) = \boldsymbol{forked}(A' + B' + C')$$

where:

$$\begin{aligned} A' &= \sum_i \alpha_i; \boldsymbol{forked}(A_i); \boldsymbol{forked}(B) \\ B' &= \sum_j \beta_j; \boldsymbol{forked}(A); \boldsymbol{forked}(B_j) \\ C' &= \sum_{\alpha_i = rev(\beta_j)} \tau; \boldsymbol{forked}(A_i); \boldsymbol{forked}(B_j) \end{aligned}$$

Proposition 4.10 ($\mathcal{E}_2$ is sound) *The expansion law $\mathcal{E}_2$ is sound with respect to strong process congruence.*

Proof

See Appendix B proposition B.4 ∎

As an example illustrating the application of expansion law $\mathcal{E}_2$ we prove in figure 4.2 that
$\vdash \boldsymbol{forked}(\alpha); \boldsymbol{forked}(\beta) = \boldsymbol{forked}(\beta); \boldsymbol{forked}(\alpha)$.

The basic axioms $\mathcal{AX}$, the two expansion laws $\mathcal{E}_1$ and $\mathcal{E}_2$ and the inference rules $\mathcal{I}$ constitute our proof system $\mathcal{AS}$, which we shall show to be (limited) complete. We write $\vdash p = q$ if the equivalence of p and q can be entailed within the proof system $\mathcal{AS}$.

4.1.6 Soundness and Limited Completeness

In this section we shall show, that the axiom system $\mathcal{AS}$ is sound and complete with respect to strong process congruence. That is, we shall prove that $\vdash p =$

$$
\begin{array}{l}
\boldsymbol{forked}(\alpha); \boldsymbol{forked}(\beta) \\
= \\
\boldsymbol{forked}(\alpha; \boldsymbol{nil}); \boldsymbol{forked}(\beta; \boldsymbol{nil}) \\
\text{– } \textit{axiom (4.8)} \\
= \\
\boldsymbol{forked}(\alpha; \boldsymbol{forked}(\boldsymbol{nil}); \boldsymbol{forked}(\beta; \boldsymbol{nil}) + \\
\qquad \beta; \boldsymbol{forked}(\alpha; \boldsymbol{nil}); \boldsymbol{forked}(\boldsymbol{nil})) \\
\text{– } \textit{expansion law } \mathcal{E}_2 \\
= \\
\boldsymbol{forked}(\alpha; \boldsymbol{forked}(\beta) + \beta; \boldsymbol{forked}(\alpha)) \\
\text{– } \textit{axioms (4.18),(4.8)} \\
= \\
\boldsymbol{forked}(\alpha; \beta + \beta; \alpha) \\
\text{– } \textit{axiom (4.17) applied twice} \\
= \\
\boldsymbol{forked}(\beta; \alpha + \alpha; \beta) \\
\text{– } \textit{axiom (4.13)} \\
= \\
\vdots \\
= \\
\boldsymbol{forked}(\beta); \boldsymbol{forked}(\alpha) \\
\text{– } \textit{by the same reasoning}
\end{array}
$$

Figure 4.2: A proof using expansion law $\mathcal{E}_2$.

$q \Leftrightarrow p \equiv q$. The soundness ($\Rightarrow$) has already been proved since the axioms $\mathcal{A}X$, $\mathcal{E}_1$ and $\mathcal{E}_2$ have been proved to hold when replacing '$=$' with '$\equiv$', and so have the inference rules $\mathcal{I}$.

The basic technique for showing completeness is classical; i.e. first to prove that each process term in the original ***forked***-free calculus of section 3.1 is equivalent to a term in normal form, and then to prove completeness for normal forms. The fact that the completeness result only will apply to ***forked***-free processes makes it "limited".

$\mathcal{A}S$ is Sound

Proposition 4.11 ($\mathcal{A}S$ is sound) *For all p, q in* FC^{Φ},

$$\textit{Whenever} \vdash p = q \textit{ then } p \equiv q.$$

Proof

Soundness of $\mathcal{A}S$ reduces to soundness of the basic axioms $\mathcal{A}X$, soundness of the expansion laws $\mathcal{E}_1$ and $\mathcal{E}_2$ and soundness of the inference rules $\mathcal{I}$. This was proved in propositions 4.3, 4.8, 4.10 and 4.5.

■

Existence of Normal Forms

This subsection states four lemmas and a proposition. The first two lemmas state, that in certain cases, two applications of the ***forked*** operator can be reduced to one application. The next two lemmas state the existence of normal forms for subcategories of process terms. These four lemmas lead to the final proposition stating that there exists a normal form for every process term in FC. Note that there does not exist a normal form for every process term in FC^{Φ}; as an example, the process ***forked***(α) has no normal form.

First we need a lemma saying that nested ***forked*** applications in certain cases can be removed, only keeping the outermost application.

Lemma 4.12 (Nested *forkeds*) *For any normal form N, there exists a simple normal form A such that:*

$$\vdash \boldsymbol{forked}(N) = \boldsymbol{forked}(A)$$

Proof

See Appendix B lemma B.5. ■

The second lemma says that two ***forked*** applications (one after the other) in certain cases can be reduced to a single ***forked*** application.

Lemma 4.13 (Sequential *forkeds*) *For any simple normal forms A and B, there exists a simple normal form C such that:*

$$\vdash \boldsymbol{forked}(A); \boldsymbol{forked}(B) = \boldsymbol{forked}(C)$$

Proof

See Appendix B lemma B.6. ■

Now to the two lemmas stating the existence of normal forms for subcategories of process terms.

Lemma 4.14 (Normal forms for a subcategory) *For any simple normal form A and normal form M (M different from **nil**), there exists a normal form N such that:*

$$\vdash \boldsymbol{forked}(A); M = N$$

Proof

See Appendix B lemma B.7. ∎

Lemma 4.15 (Normal forms for a subcategory) *For any two normal forms M, N, there exists a normal form K such that:*

$$\vdash M; N = K$$

Proof

See Appendix B lemma B.8. ∎

We can finally state the following important proposition:

Proposition 4.16 (Normal forms for FC) *For any p in* FC*, there exists a normal form N such that:*

$$\vdash p = N$$

Proof

The proof is by structural induction. Assume that any subterm has a normal form. That is, if p and q are subterms, then $\vdash p = M$ and $\vdash q = N$ where:

$$\begin{aligned} M &= \sum_i \alpha_i; M_i + \sum_k \gamma_k; \boldsymbol{forked}(C_k) \\ N &= \sum_j \beta_j; N_j + \sum_l \delta_l; \boldsymbol{forked}(D_l) \end{aligned}$$

Then we proceed by case analysis:

Case *nil*: ***nil*** is on normal form.

Case α: $\alpha = \alpha; \boldsymbol{nil}$ using axiom (4.8).

Case $p + q$:

$$
\begin{array}{l}
p + q \\
= \\
M + N \\
\textit{– induction hypothesis} \\
= \\
(\sum_i \alpha_i; M_i + \sum_k \gamma_k; \boldsymbol{forked}(C_k)) + \\
(\sum_j \beta_j; N_j + \sum_l \delta_l; \boldsymbol{forked}(D_l)) \\
\textit{– unfolding of } M \textit{ and } N \\
= \\
(\sum_i \alpha_i; M_i + \sum_j \beta_j; N_j) + \\
(\sum_k \gamma_k; \boldsymbol{forked}(C_k) + \sum_l \delta_l; \boldsymbol{forked}(D_l)) \\
\textit{– axiom (4.13)}
\end{array}
$$

Case $p; q$: If N is $\boldsymbol{nil}$, the normal form is M. Otherwise use the normal form lemma 4.15:

$$
\begin{array}{ll}
p; q & \\
= & \\
M; N & \textit{– induction hypothesis} \\
= & \\
K & \textit{– lemma 4.15}
\end{array}
$$

Case $\boldsymbol{fork}(p)$: Use lemma 4.12:

$$
\begin{array}{ll}
\boldsymbol{fork}(p) & \\
= & \\
\boldsymbol{fork}(M) & \textit{– induction hypothesis} \\
= & \\
\tau; \boldsymbol{forked}(M) & \textit{– axiom (4.16)} \\
= & \\
\tau; \boldsymbol{forked}(A) & \textit{– lemma 4.12 applied to } \boldsymbol{forked}(M) \\
 & \textit{– yielding } \boldsymbol{forked}(A) \textit{ where } A \textit{ is on} \\
 & \textit{– simple normal form}
\end{array}
$$

∎

Limited Completeness

We are now able to formally present our completeness result. First, we shall state the important completeness result for normal forms. The proof, however, requires the following lemma, stating completeness for a subcategory of processes.

Lemma 4.17 *For all simple normal forms A, B in* FC^{Φ}*,*

$$\begin{array}{ll} \textit{Whenever} & \boldsymbol{forked}(A) \equiv \boldsymbol{forked}(B) \\ \textit{then} & \vdash \boldsymbol{forked}(A) = \boldsymbol{forked}(B) \end{array}$$

Proof

We will show that $\boldsymbol{forked}(A) \equiv \boldsymbol{forked}(B)$ implies that $\vdash A = B$. Then the result follows from the fact that congruence is one of the inference rules for '='.

Assume that $\boldsymbol{forked}(A) \equiv \boldsymbol{forked}(B)$. We use induction on the sum n of the depths of A and B.

Base case $n = 0$: In this case both A and B are $\boldsymbol{nil}$, and reflexivity yields that $\vdash \boldsymbol{nil} = \boldsymbol{nil}$.

Induction step $n > 0$: Assume that A and B have the forms:

$$\begin{array}{rcl} A & = & \sum_i \alpha_i; A_i \\ B & = & \sum_j \beta_j; B_j \end{array}$$

As a step towards showing that $\vdash A = B$, we shall show that for each summand S_A of A, there is a summand S_B of B, such that $\vdash S_A = S_B$; and vice versa. By definition 3.7, $\{\!|\boldsymbol{forked}(A); \pi|\!\} \smile \{\!|\boldsymbol{forked}(B); \pi|\!\}$. Due to axiom (B.1), we the have that $\{\!|A, \pi|\!\} \smile \{\!|B, \pi|\!\}$. Now let $\{\!|A, \pi|\!\}$ make a move.

$\{\!|A, \pi|\!\} \xrightarrow{\alpha_i} \{\!|\boldsymbol{nil}; A_i, \pi|\!\}$ where α_i is different from π. Then since $\smile$ is a bisimulation, $\{\!|B, \pi|\!\} \xrightarrow{\alpha_i} \{\!|\boldsymbol{nil}; B_j, \pi|\!\}$ where $\{\!|\boldsymbol{nil}; A_i, \pi|\!\} \smile \{\!|\boldsymbol{nil}; B_j, \pi|\!\}$. But then again due to axiom (B.1) we have that $\{\!|\boldsymbol{forked}(\boldsymbol{nil}; A_i); \pi|\!\} \smile \{\!|\boldsymbol{forked}(\boldsymbol{nil}; B_j); \pi|\!\}$ which means that $\boldsymbol{forked}(\boldsymbol{nil}; A_i) \equiv \boldsymbol{forked}(\boldsymbol{nil}; B_j)$. From the latter we can deduce that $\boldsymbol{forked}(A_i) \equiv \boldsymbol{forked}(B_j)$ since equation (4.9) is known to be sound.

Since $\boldsymbol{forked}(A_i) \equiv \boldsymbol{forked}(B_j)$, the induction hypothesis yields that $\vdash A_i = B_j$ and therefore that $\vdash \alpha_i; A_i = \alpha_i; B_j$ due to the congruence property.

So for every summand $\alpha_i; A_i$ of A, there is a summand $\alpha_i; B_j$ of B such that $\vdash \alpha_i; A_i = \alpha_i; B_j$:

$$\begin{aligned} A &= \ldots + \alpha_i; A_i + \ldots \\ B &= \ldots + \alpha_i; B_j + \ldots \end{aligned}$$

Similarly for every summand of B.

We can now deduce that $\vdash A = A + B$ and $\vdash A + B = B$ and therefore by transitivity that $\vdash A = B$. The technique for showing for example that $\vdash A = A + B$ is to apply the following transformation for each summand $\beta_j; B_j$ of B. We know that there exists a summand $\alpha_i; A_i$ of A such that $\vdash \alpha_i; A_i = \beta_j; B_j$. Then we can perform the following deduction:

A

$=$

$A + \alpha_i; A_i$

– *axioms (4.12),(4.13) and (4.15)*

$=$

$A + \beta_j; B_j$

– *since* $\vdash \alpha_i; A_i = \beta_j; B_j$

– *and then congruence property*

∎

We can now state a completeness result for the normal forms of FC^{Φ}.

Proposition 4.18 (Normal form completeness) *For all normal forms M, N in* FC^{Φ},

$$\textit{Whenever } M \equiv N \textit{ then } \vdash M = N$$

Proof

Assume that $M \equiv N$. We use induction on the maximum depth n of M and N.

Base case $n = 0$: In this case both M and N are ***nil***, and reflexivity yields that $\vdash$ ***nil*** $=$ ***nil***.

Induction step $n > 0$: Assume that M and N have the forms:

$$
\begin{aligned}
M &= \sum_i \alpha_i; M_i + \sum_k \gamma_k; \boldsymbol{forked}(C_k) \\
N &= \sum_j \beta_j; N_j + \sum_l \delta_l; \boldsymbol{forked}(D_l)
\end{aligned}
$$

We assume that the C_k's and D_l's are not $\boldsymbol{nil}$. As a step towards showing that $\vdash M = N$, we shall show that for each summand of M, there is an equivalent summand of N, and vice versa. By definition 3.7, $\{|M;\pi|\} \smile \{|N;\pi|\}$. Now let $\{|M;\pi|\}$ make a move. We get two cases depending on whether M choses an $\alpha_i; M_i$ alternative or an $\gamma_k; \boldsymbol{forked}(C_k)$ alternative.

Case 1 $M \overset{\alpha_i}{\hookrightarrow} \boldsymbol{nil}; M_i$ and $\{|M;\pi|\} \xrightarrow{\alpha_i} \{|\boldsymbol{nil}; M_i;\pi|\}$, then since $\smile$ is a bisimulation, $\{|N;\pi|\} \xrightarrow{\alpha_i} \{|\boldsymbol{nil}; N_j;\pi|\}$ where $\{|\boldsymbol{nil}; M_i;\pi|\} \smile \{|\boldsymbol{nil}; N_j;\pi|\}$. Note that N cannot choose an $\delta_l; \boldsymbol{forked}(D_l)$ alternative since it can never hold that $\{|\boldsymbol{nil}; M_i;\pi|\} \smile \{|\boldsymbol{nil}; \boldsymbol{forked}(D_l);\pi|\}$. But then $M \overset{\alpha_i}{\hookrightarrow} \boldsymbol{nil}; M_i$ and $N \overset{\alpha_i}{\hookrightarrow} \boldsymbol{nil}; N_j$, and $\boldsymbol{nil}; M_i \equiv \boldsymbol{nil}; N_j$. From the latter we can deduce that $M_i \equiv N_j$ since equation (4.9) is known to be sound.

Since $M_i \equiv N_j$ the induction hypothesis yields that $\vdash M_i = N_j$ and therefore that $\vdash \alpha_i; M_i = \alpha_i; N_j$ due to the congruence property. So for every summand $\alpha_i; M_i$ of M, there is a summand $\alpha_i; N_j$ of N such that $\vdash \alpha_i; M_i = \alpha_i; N_j$:

$$
\begin{aligned}
M &= \ldots + \alpha_i; M_i + \ldots \\
N &= \ldots + \alpha_i; N_j + \ldots
\end{aligned}
$$

Case 2 $M \overset{\gamma_k}{\hookrightarrow} \boldsymbol{nil}; \boldsymbol{forked}(C_k)$ and $\{|M;\pi|\} \xrightarrow{\gamma_k} \{|\boldsymbol{nil}; \boldsymbol{forked}(C_k);\pi|\}$, then since $\smile$ is a bisimulation, $\{|N;\pi|\} \xrightarrow{\gamma_k} \{|\boldsymbol{nil}; \boldsymbol{forked}(D_l);\pi|\}$ where $\{|\boldsymbol{nil}; \boldsymbol{forked}(C_k);\pi|\} \smile \{|\boldsymbol{nil}; \boldsymbol{forked}(D_l);\pi|\}$. Note that N cannot choose an $\beta_j; N_j$ alternative since it can never hold that $\{|\boldsymbol{nil}; \boldsymbol{forked}(C_k);\pi|\} \smile \{|\boldsymbol{nil}; N_j;\pi|\}$. But then $M \overset{\gamma_k}{\hookrightarrow} \boldsymbol{nil}; \boldsymbol{forked}(C_k)$ and $N \overset{\gamma_k}{\hookrightarrow} \boldsymbol{nil}; \boldsymbol{forked}(D_l)$, and $\boldsymbol{nil}; \boldsymbol{forked}(C_k) \equiv \boldsymbol{nil}; \boldsymbol{forked}(D_l)$. From the latter we can deduce that $\boldsymbol{forked}(C_k) \equiv \boldsymbol{forked}(D_l)$ since equation (4.9) is known to be sound.

Since $\boldsymbol{forked}(C_k) \equiv \boldsymbol{forked}(D_l)$, lemma 4.17 yields that $\vdash \boldsymbol{forked}(C_k) = \boldsymbol{forked}(D_l)$ and therefore that $\vdash \gamma_k; \boldsymbol{forked}(C_k) = \gamma_k; \boldsymbol{forked}(D_l)$ due to the congruence property. So for every summand $\gamma_k; \boldsymbol{forked}(C_k)$

of M, there is a summand $\gamma_k;\boldsymbol{forked}(D_l)$ of N such that $\vdash \gamma_k;\boldsymbol{forked}(C_k) = \gamma_k;\boldsymbol{forked}(D_l)$:

$$\begin{array}{rcl} M & = & \ldots + \gamma_k;\boldsymbol{forked}(C_k) + \ldots \\ N & = & \ldots + \gamma_k;\boldsymbol{forked}(D_l) + \ldots \end{array}$$

From the above case analysis, we can conclude that for every summand S_M of M there exists a summand S_N of N such that $\vdash S_M = S_N$. Similarly, for every summand of N there exists an equivalent summand of M.

We can now deduce that $\vdash M = M + N$ and $\vdash M + N = N$ and therefore by transitivity that $\vdash M = N$. The technique for showing for example that $\vdash M = M + N$ is to apply the following transformation for each summand S_N of N. We know that there exists a summand S_M of M such that $\vdash S_M = S_N$. Then we can perform the following deduction:

$$\begin{array}{ll} M & \\ = & \\ M + S_M & \textit{– axioms (4.12),(4.13) and (4.15)} \\ = & \\ M + S_N & \textit{–} \vdash S_M = S_N \textit{ and then congruence property} \end{array}$$

■

The completeness result we arrive at is limited in the sense that it only holds for process terms of FC (and not FC^{Φ}).

Theorem 4.19 ($\mathcal{AS}$ is limited Complete) *For all p, q in* FC*,*

$$\textit{Whenever } p \equiv q \textit{ then } \vdash p = q$$

Proof

Assume that $p \equiv q$. Now, due to proposition 4.16 there exist normal forms, say M and N such that $\vdash p = M$ and $\vdash q = N$. Then, due to the soundness proposition 4.11, we deduce $p \equiv M$ and $q \equiv N$. The initial assumption $p \equiv q$ together with the latter two facts then yields $M \equiv N$ by transitivity. But proposition 4.18 states completeness for normal forms, so $\vdash M = N$. Now together with the facts that $\vdash p = M$ and $\vdash q = N$, transitivity then yields the result $\vdash p = q$.

■

4.2 An Adequate Logic

In this section we shall define a logic, in which properties of FC processes can be stated. A property about a process is some, possibly partial, characterisation of the behaviour of the process. As an example, consider the following process:

$$p \stackrel{def}{=} \alpha_1; (\beta + \gamma) + \alpha_2; \textbf{\textit{fork}}(\beta + \gamma) \tag{4.19}$$

One property is: *an α_1 action is possible, and if performed, then a β action is thereafter possible, but an α_2 action is not possible*. Another property is: *an α_2 action is possible, and if performed, the process may thereafter terminate (by performing a τ action), leaving forked a process that can peform a β action*. Note that the latter property reflects that ***fork*** results in a τ, and in the case of p, the ***fork*** expression terminates p, although the action $\beta + \gamma$ is left forked.

We shall choose a particular kind of logic, often referred to as Hennessy-Milner logic [HM85, Mil81], which is particularly designed for stating properties of labelled transition systems. In very general terms, the logic consists of a set Ψ of properties (defined by a grammar) and of a satisfaction relation $\models \subseteq \mathcal{L} \times \Psi$ between processes and properties. We will say that a process p satisfies a property ψ, if $(p, \psi) \in \models$. Using infix notation we shall write the latter as $p \models \psi$.

Our choice of Hennessy-Milner Logic as a basis is mainly motivated by its focus on action modalities; the fact that Hennessy-Milner logic is well-suited for stating properties about the actions of a *labelled* transition system, ignoring properties of the individual states. We could, however, also have used some variant of Temporal Logic [Eme90, MP89, Sti88, Sti87]. These logics are though typically defined for *unlabelled* transition systems (Kripke-structures), where it is the states between the transitions that are observed, normally via the values of variables in each state. Hence, Temporal Logics are normally directly applicable for parallel systems based on shared variables. There are though Temporal Logics for labelled transition systems as well, see for example [NV90, NFGR92, FGR92].

Since the interesting thing to observe from FC processes are the actions they perform, just as is the case for CCS, Hennessy-Milner Logic seems a natural choice. In any case, Hennessy-Milner Logic is sufficient for characterising the notion of equivalence that we have defined in the sense that equivalent processes satisfy the same formulae. This property, referred to as adequacy, is a formal argument for the sufficiency of Hennessy-Milner Logic.

The first section introduces the logic. The second section is concerned with the adequacy of the logic with respect to the congruence relation $\equiv$ defined on processes. That is, the logic is adequate if it holds that: two processes are equivalent if and only if they satisfy the same properties. We shall show that the defined logic is indeed adequate in the above sense, thus obtaining a result similar to that for CCS. Note that Hennessy-Milner logic was originally proposed in [HM85] as an adequate logic for CCS. It was introduced as an alternative way of characterising equivalence.

4.2.1 A Logic

The logic Ψ is the propositional logic (with its usual meaning) extended with modal properties of the form $<\alpha>\psi$, which for a process p asserts that: *it is possible for p to do an α and thereby reach a state which satisfies ψ*. The syntax, generating the language Ψ, is as follows:

$$\psi \quad ::= \quad \boldsymbol{tt} \mid \psi_1 \,\wedge\, \psi_2 \mid \neg\psi \mid \; <\alpha>\psi$$

Note that α was defined in section 3.1 to range over $\mathcal{A}$ where $\alpha ::= \tau \mid a? \mid a!$. As an example, consider the process 4.19 defined in the beginning of the chapter. One property that were formulated in words was: *an α_1 action is possible, and if performed, then a β action is thereafter possible, but an α_2 action is not possible.* This is now stated as:

$$<\alpha_1>(<\beta>\boldsymbol{tt} \wedge \neg <\alpha_2>\boldsymbol{tt})$$

We shall define the satisfaction relation $\models$ between processes and properties in two steps; first we define a satisfaction relation $\models\ \subseteq \mathcal{M} \times \Psi$ between programs and properties.

The logic were originally developed for labelled transition systems in general, without depending on the particular language behind ([Mil89]). Since the semantics of programs is a labelled transitions system $lts = (\mathcal{M}, Lab, \longrightarrow)$, it then follows directly how to define the satisfaction relation between programs and properties:

Definition 4.20 (The satisfaction relation $\models$) *The relation $\models\ \subseteq \mathcal{M}\times\Psi$ is the least relation such that for any program P in $\mathcal{M}$, and for any properties ψ, ψ_1 and ψ_2 in Ψ, and for any action α in $\mathcal{A}$,*

$$\begin{array}{lll} P \models \boldsymbol{tt} & & \\ P \models \psi_1 \wedge \psi_2 & \textit{iff} & P \models \psi_1 \textit{ and } P \models \psi_2 \\ P \models \neg\psi & \textit{iff} & \textit{not } P \models \psi \\ P \models <\alpha>\psi & \textit{iff} & P \xrightarrow{\alpha} P' \textit{ for some } P' \textit{ and } P' \models \psi \end{array}$$

Recall now the definition of $\equiv$: for any processes p, q in $\mathcal{L}$: $p \equiv q$ iff $\{|p;\pi|\} \smile \{|q;\pi|\}$). This definition gives us a hint of how to define the satisfaction relation $\models$

between processes and properties in terms of the satisfaction relation $\models$ between programs and properties:

Definition 4.21 (The satisfaction relation $\models$) *The relation $\models \subseteq \mathcal{L} \times \Psi$ is defined as:*

$$p \models \psi \quad \textit{iff} \quad \{\!|p;\pi|\!\}\models\psi$$

As an example, consider the two processes:

$$\begin{array}{rcl} p & \stackrel{def}{=} & \alpha;(\beta+\gamma) \\ q & \stackrel{def}{=} & \alpha;\beta+\alpha;\gamma \end{array}$$

Surely it is not the case that $p \equiv q$. The two processes can be distinguished by the following property ψ_1 in the sense that $p \models \psi_1$ but $q \not\models \psi_1$:

$$\psi_1 \stackrel{def}{=} <\alpha>(<\beta>\, \boldsymbol{tt} \wedge\ <\gamma>\, \boldsymbol{tt})$$

That is, $\{\!|p;\pi|\!\}\models\psi_1$, but $\{\!|q;\pi|\!\} \not\models \psi_1$. The property ψ_2 below is such that $q \models \psi_2$ but $p \not\models \psi_2$:

$$\psi_2 \stackrel{def}{=} <\alpha>\neg <\beta>\, \boldsymbol{tt}$$

We shall also illustrate how forking can be specified. Consider the processes:

$$\begin{array}{rcl} r & \stackrel{def}{=} & \tau;\alpha \\ s & \stackrel{def}{=} & \boldsymbol{fork}(\alpha) \end{array}$$

We recall that the two processes are not congruent, $r \not\equiv s$, since $\{\!|\tau;\alpha;\pi|\!\} \not\sim \{\!|\boldsymbol{fork}(\alpha);\pi|\!\}$. Recall that the program $\{\!|\boldsymbol{fork}(\alpha);\pi|\!\}$ can perform the π action before the α action which is not the case for the other program. That is,

$$\{\!|\boldsymbol{fork}(\alpha);\pi|\!\} \xrightarrow{\tau} \{\!|\boldsymbol{nil};\pi,\alpha|\!\} \xrightarrow{\pi} \{\!|\boldsymbol{nil},\alpha|\!\} \xrightarrow{\alpha} \{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\}$$

The property ψ_3 below distinguishes the two processes r and s in the sense that $r \not\models \psi_3$ but $s \models \psi_3$:

$$\psi_3 \stackrel{def}{=} <\tau><\pi><\alpha>\, \boldsymbol{tt}$$

That is, $\{\!|\tau;\alpha;\pi|\!\} \not\models \psi_3$ but $\{\!|\boldsymbol{fork}(\alpha);\pi|\!\})\models\psi_3$. Note that the π-event by convention is used to signal termination, so we can read property ψ_3 as follows: *process q can perform a τ, whereupon it terminates (π), leaving forked a process that can peform an α action.* We regard properties of the form $<\pi>\psi$ as having a special status since they state termination, and we shall adopt the following abbreviation:

$$\bullet\psi \stackrel{def}{=} <\pi> \psi$$

The property ψ_3 can thus be written as $<\tau> \bullet <\alpha> \boldsymbol{tt}$. We read this as: *the process s can perform a τ action, and then terminate ($\bullet$), leaving forked a process which can perform an α action.*

For the purpose of writing some more practical examples, we additionally define the following derived forms:

$$\begin{array}{rcl} \boldsymbol{ff} & \stackrel{def}{=} & \neg \boldsymbol{tt} \\ \psi_1 \vee \psi_2 & \stackrel{def}{=} & \neg(\neg\psi_1 \wedge \neg\psi_2) \\ <\alpha_1 \ldots \alpha_n> \psi & \stackrel{def}{=} & <\alpha_1> \ldots <\alpha_n> \psi \\ [\alpha_1 \ldots \alpha_n]\psi & \stackrel{def}{=} & \neg <\alpha_1 \ldots \alpha_n> \neg\psi \end{array}$$

A property of the form $[\alpha_1 \ldots \alpha_n]\psi$ means that: if the actions $\alpha_1 \ldots \alpha_n$ are performed, then ψ will hold. Note, however, that the property holds if the sequence $\alpha_1 \ldots \alpha_n$ is not possible. As a special case, the property $[\alpha_1 \ldots \alpha_n]\boldsymbol{ff}$ means: *if the actions $\alpha_1 \ldots \alpha_n$ are performed, then $\boldsymbol{ff}$ will hold.* Since $\boldsymbol{ff}$ cannot hold for any program, it means that $\alpha_1 \ldots \alpha_n$ is not possible. As a final example, consider the process p (4.19) from the introduction. It should now be easy to see that:

$$\begin{array}{rcl} p & \models & <\alpha_1> (<\beta> \boldsymbol{tt} \wedge \neg <\alpha_2> \boldsymbol{tt}) \\ p & \models & [\alpha_2 \tau] \bullet <\beta> \boldsymbol{tt} \\ p & \models & [\alpha_1] \neg \bullet \boldsymbol{tt} \end{array}$$

4.2.2 Adequateness

In this section we shall show that the logic Ψ is adequate with respect to the congruence $\equiv$ on processes. That is, we shall show that two processes are related by $\equiv$ if and only if they satisfy (via $\models$) the same properties in Ψ. Since the satisfaction relation $\models$ between processes and properties is defined in terms of the satisfaction relation $\models$ between programs and properties, we shall, however, first show adequateness with respect to the congruence $\smile$ on programs. Then adequateness with respect to $\equiv$ will follow easily.

Adequateness wrt. $\smile$

We shall show that two programs are related by $\smile$ if and only if they satisfy (via $\models$) the same properties in Ψ. Following [Mil89], this is a consequence of the facts that $lts = (\mathcal{M}, Lab, \longrightarrow)$ is a labelled transition system, and $\smile$ is the largest bisimulation induced by lts, and the logic $(\Psi, \models)$ is the standard Hennessy-Milner logic for lts.

We repeat the reasoning of [Mil89] below. One needs to reformulate the definition of the congruence $\smile$, and one also needs to reformulate the definition of the set Ψ of properties. The basic principle in the reformulations is to define the concepts as limits of certain chains of relations respectively logics. Let us first redefine $\smile$.

Stratification of $\smile$

We define a sequence[2] of relations $\smile_n \subseteq \mathcal{M} \times \mathcal{M}$ for $n \in \mathbf{N}_0$, in such a way that $\smile_{n+1} \subseteq \smile_n$:

Definition 4.22 (The relations $\smile_n$) *For any programs P, Q in $\mathcal{M}$,*

1. $P \smile_0 Q$. *That is,* $\smile_0 = \mathcal{M} \times \mathcal{M}$.
2. $P \smile_{n+1} Q$ *iff*
   (a) *Whenever* $P \xrightarrow{\alpha} P'$ *for some* P' *then* $Q \xrightarrow{\alpha} Q'$ *for some* Q' *and* $P' \smile_n Q'$
   (b) *Whenever* $Q \xrightarrow{\alpha} Q'$ *for some* Q' *then* $P \xrightarrow{\alpha} P'$ *for some* P' *and* $P' \smile_n Q'$

Note that $P \smile_n Q$ means: if we are only able to observe the n first transitions of a program, then we will not be able to distinguish P and Q.

Proposition 4.23

$$\smile = \bigcap_{n \in \boldsymbol{N}} \smile_n$$

[2] In [Mil89] n ranges over the ordinal numbers, but this is not necessary in our case since the programs we consider are finite branching. That is, for any program P obtained as a derivative from a program $\{|p|\}$, (p is a process in $\mathcal{L}$) and for any action α, there are only finitely many programs P' such that $P \xrightarrow{\alpha} P'$.

Stratification of Ψ

Recalling what $P \smile_n Q$ means, we easily see its implication: P and Q cannot be distinguished by formulae of "depth" less than n, where "depth" means the degree of modality nesting. To make this more formal, we define a function $depth : \Psi \rightarrow \mathbf{N}$ as follows:

$$\begin{array}{rcl} depth(\boldsymbol{tt}) & = & 0 \\ depth(\psi_1 \wedge \psi_2) & = & max(depth(\psi_1), depth(\psi_2)) \\ depth(\neg\psi) & = & depth(\psi) \\ depth(<\alpha>\psi) & = & 1 + depth(\psi) \end{array}$$

The function $\max : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ returns the biggest og its two arguments.

Definition 4.24 (The logics Ψ_n) *For every $n \in \boldsymbol{N}$ the logic Ψ_n is defined as:*

$$\Psi_n = \{\psi \in \Psi \mid depth(\psi) \leq n\}$$

Now, obviously it holds that $\Psi = \bigcup_{n \in \mathbf{N}} \Psi_n$.

Stratified Adequateness

We shall now, as an intermediate result, show that for each $n \in \mathbf{N}$, the logic Ψ_n is adequate wrt. $\smile_n$. Let for any program P, $\bar{\bar{\Psi}}_n(P)$ be defined as the set of properties of maximum depth n that P satisfies:

$$\bar{\bar{\Psi}}_n(P) = \{\psi \in \Psi_n \mid P \models \psi\}$$

Then we have following lemma:

Lemma 4.25 *For any two programs $P, Q \in \mathcal{M}$ and for any $n \in \boldsymbol{N}$,*

$$P \smile_n Q \quad \textit{iff} \quad \bar{\bar{\Psi}}_n(P) = \bar{\bar{\Psi}}_n(Q)$$

Proof

We use induction on n.

Base case $n = 0$: The left hand side $P \smile_0 Q$ obviously holds since $\smile = \mathcal{M} \times \mathcal{M}$. The right hand side holds since $\overset{=}{\Psi}_0\,(P)$ is just the set of properties logically equivalent to $\boldsymbol{tt}$ (no occurrences of the modal operator $<\_>\_$), and this is also the case for $\overset{=}{\Psi}_0\,(Q)$.

Induction step $n = k + 1$: We consider the two directions ('only if' and 'if') in turn.

1. $P \smile_{k+1} Q$ implies $\overset{=}{\Psi}_{k+1}\,(P) = \overset{=}{\Psi}_{k+1}\,(Q)$:
   We show by structural induction on $\psi \in \Psi_{k+1}$ that, if $P \smile_{k+1} Q$ and $P \models \psi$ then $Q \models \psi$.
   (a) ψ **is** $\boldsymbol{tt}$: this case is trivial.
   (b) ψ **is** $\psi_1 \wedge \psi_2$: then $P \models \psi_1$ and $P \models \psi_2$. Structural induction gives that $Q \models \psi_1$ and $Q \models \psi_2$, and thereby that $Q \models \psi$.
   (c) ψ **is** $\neg\psi'$: then not $P \models \psi'$. Now if $Q \models \psi'$ then by structural induction $P \models \psi'$ which is a contradiction. Therefore not $Q \models \psi'$ and thus $Q \models \psi$.
   (d) ψ **is** $<\alpha>\psi'$: then $P \xrightarrow{\alpha} P'$ for some P' where $P' \models \psi'$. Since $P \smile_{k+1} Q$, there exists a Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \smile_k Q'$. Now $\psi' \in \Psi_k$ so by induction $(k < n)$ $Q' \models \psi'$, and therefore $Q \models \psi$.

2. $P \not\smile_{k+1} Q$ implies $\overset{=}{\Psi}_{k+1}\,(P) \neq \overset{=}{\Psi}_{k+1}\,(Q)$:
   Assume that $P \not\smile_{k+1} Q$. Then, without loss of generality, we can assume that there exists an α and a P' such that $P \xrightarrow{\alpha} P'$, and such that if $Q \xrightarrow{\alpha} Q'$ for some Q', then $P' \not\smile_k Q'$. Let $\{Q_1, \ldots, Q_d\} = \{Q' \in \mathcal{M} \mid Q \xrightarrow{\alpha} Q'\}$. By induction, $\overset{=}{\Psi}_k\,(P') \neq \overset{=}{\Psi}_k\,(Q_i)$ for each i, $1 \leq i \leq d$. So there are properties $\psi_1, \ldots, \psi_d \in \Psi_k$ such that for each i, $1 \leq i \leq d$: $P' \models \psi_i$ and not $Q_i \models \psi_i$. Now let $\overline{\psi}$ denote the property $\psi_1 \wedge \ldots \wedge \psi_d$. Then $P' \models \overline{\psi}$ but not $Q_i \models \overline{\psi}$. Therefore $P \models <\alpha>\overline{\psi}$ and not $Q \models <\alpha>\overline{\psi}$. That is, $\overset{=}{\Psi}_{k+1}\,(P) \neq \overset{=}{\Psi}_{k+1}\,(Q)$.

■

Adequateness

We can now easily prove that Ψ is adequate wrt. $\smile$. Let for any program P, $\overset{=}{\Psi}\,(P)$ be defined as the set of properties that P satisfies:

$$\overset{=}{\Psi}\,(P) = \{\psi \in \Psi \mid P \models \psi\}$$

Then we have following proposition:

Proposition 4.26 (Adequateness wrt. $\smile$) *The logic Ψ is adequate wrt. $\smile$. That is, for any two programs $P, Q \in \mathcal{M}$,*

$$P \smile Q \quad \textit{iff} \quad \bar{\bar{\Psi}}(P) = \bar{\bar{\Psi}}(Q)$$

Proof

Assume that $P \smile Q$. Then since $\smile$ is defined as the intersection of all $\smile_n$, $n \geq 0$, we have that $P \smile_n Q$ for all $n \geq 0$. But then lemma 4.25 says that $\bar{\bar{\Psi}}_n(P) = \bar{\bar{\Psi}}_n(Q)$ for all $n \geq 0$. Now, it is easily seen that $\bar{\bar{\Psi}}(P) = \bigcup_{n \in \mathbf{N}} \bar{\bar{\Psi}}_n(P)$; and due to the previous fact, the latter equals $\bigcup_{n \in \mathbf{N}} \bar{\bar{\Psi}}_n(Q)$; which again equals $\bar{\bar{\Psi}}(Q)$.

In the other direction, assume that $\bar{\bar{\Psi}}(P) = \bar{\bar{\Psi}}(Q)$. Put differently, $\bigcup_{n \in \mathbf{N}} \bar{\bar{\Psi}}_n(P) = \bigcup_{n \in \mathbf{N}} \bar{\bar{\Psi}}_n(Q)$. We can easily from this conclude that for any $n \geq 0$, $\bar{\bar{\Psi}}_n(P) = \bar{\bar{\Psi}}_n(Q)$, and thereby that $P \smile_n Q$. Then since $\smile$ is defined as the intersection of all $\smile_n$, $n \geq 0$, we have that $P \smile Q$.

∎

Adequateness wrt. $\equiv$

We are now ready to state the main theorem in this section, namely that our logic is adequate wrt. the process congruence $\equiv$. Let for any process p, $\bar{\Psi}(p)$ be defined as the set of properties that p satisfies:

$$\bar{\Psi}(p) = \{\psi \in \Psi \mid p \models \psi\}$$

Note that for any $p \in \mathcal{L}$, $\bar{\Psi}(p) = \{\psi \in \Psi \mid \{\!|p; \pi|\!\} \models \psi\} = \bar{\bar{\Psi}}(\{\!|p; \pi|\!\})$. Then we have:

Theorem 4.27 (Adequateness wrt. $\equiv$) *The logic Ψ is adequate wrt. $\equiv$. That is, for any two processes $p, q \in \mathcal{L}$,*

$$p \equiv q \quad \textit{iff} \quad \bar{\Psi}(p) = \bar{\Psi}(q)$$

Proof Assume that $p \equiv q$. Then by definition of $\equiv$, $\{\!|p;\pi|\!\} \smile \{\!|q;\pi|\!\}$. Since Ψ is adequate wrt. $\smile$ (proposition 4.26), this means that $\bar{\bar{\Psi}}\,(\{\!|p;\pi|\!\}) = \bar{\bar{\Psi}}\,(\{\!|q;\pi|\!\})$. This again means that $\bar{\Psi}\,(p) = \bar{\Psi}\,(q)$.

Now repeating the argument in the reverse direction ends the proof.

$\blacksquare$

Part III

Extensions of the Fork Calculus

Chapter 5

A Fork Calculus with Restriction and guarded Choice

We shall now extend and modify the Fork Calculus in order to obtain a more realistic calculus in which non-trivial programs can be written. We shall as before define its syntax and semantics, and a congruence relation. Then an axiomatisation is provided, and finally we present a refinement calculus in which specifications can be formulated and refined into programs in a stepwise verified-correct manner.

First, we add a recursion operator, allowing us to define processes with infinite behaviour. Recursion is not difficult to deal with (thanks to the use of operational semantics), except that the axiomatisation cannot be shown complete in presence of recursion. So although we introduce recursion now, it will be temporarily "thrown out" when formulating the axiomatisation.

Second, we introduce a 'channel allocation' operator like in CML, or a restriction operator to be in the vocabular of CCS, that makes channels local to processes. The allocation of a channel, which this operator gives rise to, results in a τ action – as does the ***fork***-operator. So just as we needed two forking operators (***fork*** and ***forked***) we also need two restriction operators in order for the axiomatisation, that we later give, to work. Recall that our calculus is designed for the purpose of theoretical investigation, and this is the reason that we allow two versions of these operators. In a "final" calculus, we would only have one version of each. As we shall see, the restriction operator has a more complex semantics than the corresponding restriction operator in CCS. This is basically forced by the presence og the fork operator.

Finally, we introduce guarded choice where the alternatives in the choice are guarded by actions. This commitment to guardedness is necessary in order to achieve nice properties of the refinement logic that we later define. In addition we obtain a congruence result that is not associated with conditions in the case of choice.

Section 5.1 defines the syntax and semantics of the new calculus. Section 5.2 defines a congruence relation.

5.1 Syntax and Semantics

5.1.1 Syntax and Informal Semantics

The syntax of the calculus is as follows.

$$\begin{array}{lcl} p & ::= & \sum_{i \in I} \alpha_i ; p_i \mid p_1 ; p_2 \mid \textbf{\textit{fork}}(p) \mid \textbf{\textit{forked}}(p) \mid (a)p \mid \{a\}p \mid \textbf{\textit{fix}}\, x \cdot p \mid x \\ \alpha & ::= & \tau \mid a? \mid a! \end{array}$$

The $\sum_{i \in I} \alpha_i ; p_i$ construct represents an action guarded choice between a finite number of processes p_i, each guarded by an action α_i. Formally, the choice construct is a mapping from indices to pairs of actions and processes. When writing choice expressions we use $+$ to combine the pairs, leaving out the indices, and we use ';' to separate actions from processes. An example is $a!; p_1 + \tau; p_2$. We shall use the constant ***nil*** to represent the empty choice where the index set $I = \emptyset$. This is the inactive process.

The term $(a)p$ is the equivalent of CCS channel restriction $p \backslash a$. It means that a is made an internal channel to p, so communication on this channel can only be done within p between local processes. Although the effect is the same as the CCS-restriction operator, we shall give our operator a different operational semantics, basically forced by the existence of the ***fork*** operator. Its meaning will be that a new unique internal channel is allocated and a will then refer to this new channel. The term $\{a\}p$ is the silent version of $(a)p$ which does not result in a τ-action.

Finally $\textbf{\textit{fix}}\, x \cdot p$ is the way to introduce recursion. For example, the process $\textbf{\textit{fix}}\, x \cdot \alpha; x$ performs α infinitely.

To avoid too many parentheses we adopt the convention that the operators have decreasing binding power, in the following order: Sequential composition (tightest binding), Choice, Recursion, Restriction. We shall further use the convention to interpretate a process $\alpha; p$ as $\sum_{i \in \{1\}} \alpha; p$. Finally, the process α is short for $\alpha;$ ***nil***.

As an example of a program in FC_r, consider the following two-place buffer.

$$(i)\textbf{\textit{fork}}(\underbrace{\textbf{\textit{fix}}\, x \cdot a?; i!; x}_{p}); \underbrace{\textbf{\textit{fix}}\, y \cdot i?; d!; y}_{q}$$

Two processes, for illustration identified as p and q, are made to run in parallel. Process p can perform an $a?$-action (accept), then communicate with q over the internal channel i, where after it returns to its initial state, ready for another $a?$-action. When q has performed the complementary $i?$-action it performs a $d!$-action (deliver), and returns to its initial state. The system performs sequences of $a?$ $d!$ actions (mixed with τ-actions), with no more that two consecutive $a?$-actions and with every $d!$-action corresponding to a preceding $a?$-action. The communications over the i-channel are non-observable from outside.

We shall later need to refer to the set of well-guarded and guarded processes. For the definition of these terms we need the following auxiliary definitions. The predicate *Silent* holds on a process p, if $p; \pi$ can perform π as the first thing, where π (as usual) represents "any computation". *Silent* is the least predicate satisfying:

$$\begin{array}{rcl}
Silent(\textbf{\textit{nil}}) & & \\
Silent(\textbf{\textit{forked}}(p)) & & \\
Silent(p_1; p_2) & \Leftarrow & Silent(p_1) \wedge Silent(p_2) \\
Silent(\{a\}p) & \Leftarrow & Silent(p) \\
Silent(\textbf{\textit{fix}}\, x \cdot p) & \Leftarrow & Silent(p)
\end{array}$$

The second definition we need is that of the function UG, that for a process returns the set of variables that are unguarded (not preceded by some action):

$$\begin{array}{rcl}
UG(\sum_i \alpha_i; p_i) & = & \emptyset \\
UG(p_1; p_2) & = & \left\{ \begin{array}{ll} UG(p_1) & \text{if } \neg Silent(p_1) \\ UG(p_1) \cup UG(p_2) & \text{if } Silent(p_1) \end{array} \right. \\
UG(\textbf{\textit{forked}}(p)) & = & UG(p) \\
UG(\{a\}p) & = & UG(p) \\
UG(\textbf{\textit{fix}}\, x \cdot p) & = & UG(p) - \{x\} \\
UG(x) & = & \{x\}
\end{array}$$

We say that a variable x is guarded in a process p if $x \notin UG(p)$. We let $\mathcal{L}_o$ stand for the set of all processes including the open ones with free variables. We shall, however, only consider well-guarded processes, where a process is well-guarded if for every occurrence of a $\textbf{\textit{fix}} x \cdot p$ construct, x is guarded in p. We let $\mathcal{L}_w$ stand for the set of these well-guarded processes. If moreover the free variables of a process p are guarded in p, we say that p is guarded. The set og guarded processes is denoted by $\mathcal{L}_g$. Finally, the set of closed well-guarded processes is denoted by $\mathcal{L}$. The defined sublanguages are related in the following way: $\mathcal{L} \subseteq \mathcal{L}_g \subseteq \mathcal{L}_w \subseteq \mathcal{L}_o$.

5.1.2 Formal Semantics

The semantics is as before divided into two layers, one for processes in isolation, and one for multisets of processes.

Processes

We define the labelled transition system $(\mathcal{L}, Lab, \hookrightarrow)$. Concerning the definition of the labels Lab, this is defined as follows:

$$\begin{array}{lcl} Com & = & \{a? \mid a \in Chan\} \cup \{a! \mid a \in Chan\} \\ Act & = & Com \cup \{\tau\} \\ Lab & = & Act \cup \{\phi(p) \mid p \in \mathcal{L}\} \cup \{\Phi(p) \mid p \in \mathcal{L}\} \cup \\ & & \{\lambda(k) \mid k \in Chan\} \cup \{\Lambda(k) \mid k \in Chan\} \end{array}$$

The sets Com and Act are as before. The set Lab now additionally includes labels of the form $\lambda(k)$ and $\Lambda(k)$ ($k \in Chan$) which arise from evaluation of processes of the form $(a)p$ and $\{a\}p$. These labels will not be observable at the second layer of semantics.

We now define the transition relation $\hookrightarrow \subseteq \mathcal{L} \times Lab \times \mathcal{L}$. Before defining this transition relation we define the predicate $Stop \subseteq \mathcal{L}$. $Stop$ is defined as the least subset of $\mathcal{L}$ satisfying the following:

$$\begin{array}{rcl} Stop(\boldsymbol{nil}) & & \\ Stop(p_1; p_2) & \Leftarrow & Stop(p_1) \wedge Stop(p_2) \\ Stop(\boldsymbol{fix}\, x \cdot p) & \Leftarrow & Stop(p) \end{array}$$

The operational semantics of FC_r processes is then as follows.

Definition 5.1 (The transition relation $\hookrightarrow$) *Let $\hookrightarrow$ be the smallest subset of $\mathcal{L} \times Lab \times \mathcal{L}$ closed under the following rules:*

$$(\textbf{Choice}) \qquad \frac{}{\sum_i \alpha_i; p_i \overset{\alpha_i}{\hookrightarrow} p_i}$$

$$(\textbf{Sequence}_1) \qquad \frac{p_1 \overset{l}{\hookrightarrow} p_1'}{p_1; p_2 \overset{l}{\hookrightarrow} p_1'; p_2}$$

$$(\textbf{Sequence}_2) \quad \frac{p_2 \overset{l}{\hookrightarrow} p_2'}{p_1; p_2 \overset{l}{\hookrightarrow} p_2'} \; Stop(p_1)$$

$$(\textbf{Fork}) \quad \frac{}{\boldsymbol{fork}(p) \overset{\phi(p)}{\hookrightarrow} \boldsymbol{nil}}$$

$$(\textbf{Forked}) \quad \frac{}{\boldsymbol{forked}(p) \overset{\Phi(p)}{\hookrightarrow} \boldsymbol{nil}}$$

$$(\textbf{Channel}) \quad \frac{}{(a)p \overset{\lambda(k)}{\hookrightarrow} p[k/a]} \; k \in Chan$$

$$(\textbf{Allocated}) \quad \frac{}{\{a\}p \overset{\Lambda(k)}{\hookrightarrow} p[k/a]} \; k \in Chan$$

$$(\textbf{Recursion}) \quad \frac{p[(\boldsymbol{fix}\, x \cdot p)/x] \overset{l}{\hookrightarrow} p'}{\boldsymbol{fix}\, x \cdot p \overset{l}{\hookrightarrow} p'}$$

The rules should be fairly simple to read. The (**Choice**)-rule says that any of the α_i actions can be performed where after p_i will continue. The (**Channel**) and (**Allocated**) rules show how the higher order labels $\lambda(k)$ and $\Lambda(k)$ are created. When we come to the second layer of semantics their use will be explained. The (**Recursion**) rule is the standard one.

We shall need to refer to the set $CV[p]$ of free channels occurring in a process p:

$$\begin{aligned}
CV[\sum_{i \in I} \alpha_i; p_i] &= \bigcup \{ CV[\alpha_i] \cup CV[p_i] \mid i \in I \} \\
CV[p_1; p_2] &= CV[p_1] \cup CV[p_2] \\
CV[\boldsymbol{fork}(p)] &= CV[p] \\
CV[\boldsymbol{forked}(p)] &= CV[p] \\
CV[(a)p] &= CV[p] - \{a\} \\
CV[\{a\}p] &= CV[p] - \{a\} \\
CV[\boldsymbol{fix}\, x \cdot p] &= CV[p] \\
CV[x] &= \{\} \\
CV[\tau] &= \{\} \\
CV[a?] &= \{a\} \\
CV[a!] &= \{a\}
\end{aligned}$$

The channels that are occurring free in a multiset P is calculated as follows:

$$CV[P] \stackrel{def}{=} \bigcup\{ CV[p] \mid p \in P\}$$

Configurations

A program is a multiset of processes. We let $\mathcal{M}$ denote the set of programs ($\mathcal{M} = MS(\mathcal{L})$).

In order to give semantics to programs that allocate (internal) channels, we introduce a component into the semantics, that explicitly keeps track of 'already allocated channels'. We refer to this component as the *record*, and we represent it as a set of (the already allocated) channels: $Record \stackrel{def}{=} \mathcal{P}(Chan)$. We let K range over *Record*. A channel allocation yields a new channel that is not already in the record, and the record is thereafter updated (extended) with the new channel.

A K-configuration $K \triangleright P$ consists of a record K and a program P. Note that we shall only consider well formed K-Configurations, where the record includes all the channels occurring in the program:

$$KCon \stackrel{def}{=} \{K \triangleright P \in Record \times \mathcal{M} \mid CV[P] \subseteq K\}$$

The semantics of K-configurations is given in terms of the labelled transition system
($KCon, Act, \longrightarrow\!\!\succ$), where Act was defined in the previous section. Thus a K-configuration can perform the actions in the set Act; recall that these were of the form $a?$, $a!$ or τ. The fork actions $\phi(p)$ and $\Phi(p)$ and the channel allocation actions $\lambda(k)$ and $\Lambda(k)$ are thus not amongst these actions. The transition relation $\longrightarrow\!\!\succ KCon \times Act \times KCon.$ is defined as follows:

Definition 5.2 (The transition relation $\longrightarrow\!\!\succ$) *Let $\longrightarrow\!\!\succ$ be the smallest subset of $KCon \times Act \times KCon$ closed under the following rules:*

$$(\textbf{Action}^{\{\!|\,|\!\}}) \qquad \frac{p \stackrel{\alpha}{\hookrightarrow} p'}{K \triangleright \{\!|p|\!\} \stackrel{\alpha}{\longrightarrow\!\!\succ} K \triangleright \{\!|p'|\!\}}$$

$$(\textbf{Fork}^{\{\!|\,|\!\}}) \qquad \frac{p \stackrel{\phi(q)}{\hookrightarrow} p'}{K \triangleright \{\!|p|\!\} \stackrel{\tau}{\longrightarrow\!\!\succ} K \triangleright \{\!|p', q|\!\}}$$

$$(\mathbf{Forked}^{\{\!|\,|\!\}}) \qquad \frac{p \stackrel{\Phi(q)}{\hookrightarrow} p'\ ,\ K \triangleright \{\!|p', q|\!\} \stackrel{\alpha}{\longrightarrow} K' \triangleright R}{K \triangleright \{\!|p|\!\} \stackrel{\alpha}{\longrightarrow} K' \triangleright R}$$

$$(\mathbf{Channel}^{\{\!|\,|\!\}}) \qquad \frac{p \stackrel{\lambda(k)}{\hookrightarrow} p'}{K \triangleright \{\!|p|\!\} \stackrel{\tau}{\longrightarrow} K \cup \{k\} \triangleright \{\!|p'|\!\}}\ {\scriptstyle k \notin K}$$

$$(\mathbf{Allocated}^{\{\!|\,|\!\}}) \qquad \frac{p \stackrel{\Lambda(k)}{\hookrightarrow} p'\ ,\ K \cup \{k\} \triangleright \{\!|p'|\!\} \stackrel{\alpha}{\longrightarrow} K' \triangleright R}{K \triangleright \{\!|p|\!\} \stackrel{\alpha}{\longrightarrow} K' \triangleright R}\ {\scriptstyle k \notin K}$$

$$(\mathbf{Parallel}_1^{\{\!|\,|\!\}}) \qquad \frac{K \triangleright P_1 \stackrel{\alpha}{\longrightarrow} K' \triangleright P_1'}{K \triangleright P_1 \cup P_2 \stackrel{\alpha}{\longrightarrow} K' \triangleright P_1' \cup P_2}$$

$$(\mathbf{Parallel}_2^{\{\!|\,|\!\}}) \qquad \frac{K \triangleright P_1 \stackrel{c}{\longrightarrow} K' \triangleright P_1'\ ,\ K \triangleright P_2 \stackrel{rev(c)}{\longrightarrow} K'' \triangleright P_2'}{K \triangleright P_1 \cup P_2 \stackrel{\tau}{\longrightarrow} K' \cup K'' \triangleright P_1' \cup P_2'}\ {\scriptstyle K = K' \cap K''}$$

The rules $(\mathbf{Action}^{\{\!|\,|\!\}})$, $(\mathbf{Fork}^{\{\!|\,|\!\}})$, $(\mathbf{Forked}^{\{\!|\,|\!\}})$ and $(\mathbf{Parallel}_1^{\{\!|\,|\!\}})$ need no further explanation since they are quite similar to the rules for the simpler calculus, except for the K-component. The $(\mathbf{Channel}^{\{\!|\,|\!\}})$ rule describes how a new (not in K) channel k is allocated. Note how the new channel is remembered in the record. Likewise, the $(\mathbf{Allocated}^{\{\!|\,|\!\}})$ rule explains how a $\Lambda(k)$ is used: if a process p can ("silently") allocate a new channel k and thereby go into p', and if p' with an updated K can go into R, then p can go into R (with a corresponding K-transformation). The $(\mathbf{Parallel}_2^{\{\!|\,|\!\}})$ rule shows how two distinct subsets of a program may communicate, resulting in a τ action. The condition on this rule states that if P_1 and P_2 allocate new channels "on the way", resulting in K' and K'', then none of these new channels must be in common (that is, the only common channels can be those in K).

We need to extend the semantics further. In order to internalise dynamic channels (the channels that are introduced by $(\_)\_$ and $\{\_\}\_$), we need a component in the semantics, that identifies the static channels (channels not introduced by these two operators). Note that the record initially contains all the static channels, but updating the record makes it no longer possible to identify the initial record. We refer to the new component as the *window*, and we represent it as the set of static channels: $\mathit{Window} \stackrel{def}{=} \mathcal{P}(\mathit{Chan})$. We let W range over Window. The window never changes throughout the execution of a program. A window together with a record is referred to an an *environment*: $\mathit{Env} \stackrel{def}{=} \mathit{Window} \times \mathit{Record}$.

A configuration $(W, K) \triangleright P$ consists of an environment (W, K) and a program P. We shall only consider well formed configurations, where the window (and the set of free channels in the program) is a subset of the record:

$$Con \stackrel{def}{=} \{(W,K) \triangleright P \in Env \times \mathcal{M} \mid (W \cup CV[P]) \subseteq K\}$$

The semantics of configurations is given in terms of the labelled transition system $(Con, Act, \longrightarrow)$. In order to define $\longrightarrow$ we need the following predicate $\_\, allows\, \_ :$ $Window \times \mathcal{A}$:

$$\begin{array}{lcl} W\ allows\ \tau & = & true \\ W\ allows\ k? & = & k \in W \\ W\ allows\ k! & = & k \in W \end{array}$$

Then we can define the dynamic behaviour of configurations.

Definition 5.3 (The transition relation $\longrightarrow$) *Let $\longrightarrow$ be the smallest subset of $Con \times Act \times Con$ satisfying the following rule:*

$$\frac{K \triangleright P \stackrel{\alpha}{\longrightarrow\!\!\!\!\succ} K' \triangleright P'}{(W,K) \triangleright P \stackrel{\alpha}{\longrightarrow} (W,K') \triangleright P'}\ W\ allows\ \alpha$$

As an example, let us observe the execution of a configuration corresponding to the process $\{i\}\boldsymbol{forked}(i!);(i?;a!+i?;b!)$. The example is chosen to illustrate the silent versions of the fork and channel allocation operators. The process is equivalent to $\tau;a!+\tau;b!$, that is: it non-deterministically executes either $a!$ or $b!$ (after a τ). We choose the initial environment (window and record) to contain the free channel names a and b. The channel allocation will give rise to the allocation of a new channel k different from a and b. Let $I \stackrel{def}{=} \{a,b\}$ represent the initial set of channels and let $E \stackrel{def}{=} I \cup \{k\}$ represent the initial set extended with the new channel k. We will show that:

$$\begin{array}{l} (I,I) \triangleright \{\!|\{i\}\boldsymbol{forked}(i!);(i?;a!+i?;b!)|\!\} \stackrel{\tau}{\longrightarrow} \\ (I,E) \triangleright \{\!|a!,\boldsymbol{nil}|\!\} \stackrel{a!}{\longrightarrow} \\ (I,E) \triangleright \{\!|\boldsymbol{nil},\boldsymbol{nil}|\!\} \end{array}$$

We start with the τ-transition, and to show this we split our deductions into three groups, corresponding to process transitions ($\hookrightarrow$), K-configuration transitions ($\longrightarrow\!\!\!\!\succ$) and finally the above configuration transition ($\longrightarrow$).

First we state the involved process transitions. Each step is numbered (to the left) and is followed by a justification referring to a semantic inference rule and occasionally to previous steps.

(1)	$\{i\}\boldsymbol{forked}(i!);(i?;a!+i?;b!) \overset{\Lambda(k)}{\hookrightarrow} \boldsymbol{forked}(k!);(k?;a!+k?;b!)$		(**Allocated**)
(2)	$\boldsymbol{forked}(k!);(k?;a!+k?;b!) \overset{\Phi(k!)}{\hookrightarrow} \boldsymbol{nil};(k?;a!+k?;b!)$		(**Forked**)
(3)	$k?;a!+k?;b! \overset{k?}{\hookrightarrow} a!$		(**Choice**)
(4)	$\boldsymbol{nil};(k?;a!+k?;b!) \overset{k?}{\hookrightarrow} a!$	(3)	(**Sequence**$_2$)
(5)	$k! \overset{k!}{\hookrightarrow} \boldsymbol{nil}$		(**Choice**)

We can then perform the following K-configuration transitions:

(6)	$E \triangleright \{\!\mid \boldsymbol{nil};(k?;a!+k?;b!) \mid\!\} \overset{k?}{\longrightarrow} E \triangleright \{\!\mid a! \mid\!\}$	(4)	(**Action**$^{\{\mid\mid\}}$)
(7)	$E \triangleright \{\!\mid k! \mid\!\} \overset{k!}{\longrightarrow} E \triangleright \{\!\mid \boldsymbol{nil} \mid\!\}$	(5)	(**Action**$^{\{\mid\mid\}}$)
(8)	$E \triangleright \{\!\mid \boldsymbol{nil};(k?;a!+k?;b!), k! \mid\!\} \overset{\tau}{\longrightarrow} E \triangleright \{\!\mid a!, \boldsymbol{nil} \mid\!\}$	(6,7)	(**Parallel**$_2^{\{\mid\mid\}}$)
(9)	$E \triangleright \{\!\mid \boldsymbol{forked}(k!);(k?;a!+k?;b!) \mid\!\} \overset{\tau}{\longrightarrow} E \triangleright \{\!\mid a!, \boldsymbol{nil} \mid\!\}$	(2,8)	(**Forked**$^{\{\mid\mid\}}$)
(10)	$I \triangleright \{\!\mid \{i\}\boldsymbol{forked}(i!);(i?;a!+i?;b!) \mid\!\} \overset{\tau}{\longrightarrow} E \triangleright \{\!\mid a!, \boldsymbol{nil} \mid\!\}$	(1,9)	(**Allocated**$^{\{\mid\mid\}}$)

Finally we can lift the evaluation of the K-configuration to the configuration with window I:

(11)	$(I,I) \triangleright \{\!\mid \{i\}\boldsymbol{forked}(i!);(i?;a!+i?;b!) \mid\!\} \overset{\tau}{\longrightarrow} (I,E) \triangleright \{\!\mid a!, \boldsymbol{nil} \mid\!\}$	(10)	I *allows* τ

To end the example we further show that $(I,E) \triangleright \{\!\mid a!, \boldsymbol{nil} \mid\!\} \overset{a!}{\longrightarrow} (I,E) \triangleright \{\!\mid \boldsymbol{nil}, \boldsymbol{nil} \mid\!\}$:

(12)	$a! \overset{a!}{\hookrightarrow} \boldsymbol{nil}$		(**Choice**)
(13)	$E \triangleright \{\!\mid a! \mid\!\} \overset{a!}{\longrightarrow} E \triangleright \{\!\mid \boldsymbol{nil} \mid\!\}$	(12)	(**Action**$^{\{\mid\mid\}}$)
(14)	$E \triangleright \{\!\mid a!, \boldsymbol{nil} \mid\!\} \overset{a!}{\longrightarrow} E \triangleright \{\!\mid \boldsymbol{nil}, \boldsymbol{nil} \mid\!\}$	(13)	(**Parallel**$_1^{\{\mid\mid\}}$)
(15)	$(I,E) \triangleright \{\!\mid a!, \boldsymbol{nil} \mid\!\} \overset{a!}{\longrightarrow} (I,E) \triangleright \{\!\mid \boldsymbol{nil}, \boldsymbol{nil} \mid\!\}$	(14)	(I *allows* $a!$)

We end this section by introducing some operations for creating and manipulating environments. Let $Env[\_] : \mathcal{L} \rightarrow Env$ be defined as follows $Env[p] \overset{def}{=} (CV[p], CV[p])$, the environment generated by the process p. This environment contains (in window as well as in record) all the channels that occur free in p. Let $W_{\_} : Env \rightarrow Window$ be defined as follows $W_{(W,K)} \overset{def}{=} W$, extracting the window of an environment. Finally, let $\cup : Env \times Env \rightarrow Env$ be defined as follows $(W_1, K_1) \cup (W_2, K_2) \overset{def}{=} (W_1 \cup W_2, K_1 \cup K_2)$, the union of two environments.

5.2 Process Congruence

We shall now define an equivalence relation $\equiv \subseteq \mathcal{L} \times \mathcal{L}$ between processes, and we shall state this also to be a congruence. For this purpose we shall, however, first define an equivalence relation $\smile \subseteq Con \times Con$ between configurations. We define $\smile$ in terms of the concept of bisimulation [Mil89].

We note that the semantics of configurations is a labelled transition system $(Con, Act, \longrightarrow)$. As described in chapter 2, such a labelled transition system generates a bisimulation equivalence between the configurations in Con. We shall let $\smile$ denote this equivalence. Recall that $\smile$ is defined as the union of all bisimulations.

Now, two processes are equivalent, if they are equivalent when "lifted" to configurations. Formally, we associate to each process its 'initial configuration':

Definition 5.4 (Initial configuration of a process) *The initial configuration $Config[p]$ of a process p is defined as:*

$$Config[p] \stackrel{def}{=} (CV[p], CV[p]) \triangleright \{\!|p; \pi|\!\}$$

We are now able to give the following formal definition of the process congruence $\equiv$:

Definition 5.5 (Process congruence $\equiv$) *The relation $\equiv \subseteq \mathcal{L} \times \mathcal{L}$ is defined as:*

$$p \equiv q \Leftrightarrow Config[p] \smile Config[q]$$

Surely $\equiv$ is an equivalence, and the proof is basically as the proof of theorem 3.9. In the remainder of this section we prove that $\equiv$ is a congruence. Before we proceed, we show a number of properties which are needed to show that $\equiv$ is a congruence. First, we define an equivalence $\sim$ between processes that equates more processes than the congruence $\equiv$. For this purpose we introduce the concept of a direct initial configuration.

Definition 5.6 (Direct initial configuration of a process) *The direct initial configuration $config[p]$ of a process p is defined as:*

$$config[p] \stackrel{def}{=} (CV[p], CV[p]) \triangleright \{\!|p|\!\}$$

Note that no π event appears. We now define the process equivalence $\sim$:

Definition 5.7 (Process equivalence $\sim$) *The relation $\sim \subseteq \mathcal{L} \times \mathcal{L}$ is defined as:*

$$p \sim q \Leftrightarrow \mathit{config}[p] \smile \mathit{config}[q]$$

Now, $\equiv$ is indeed included in $\sim$ (a fact exploited in future proofs):

Lemma 5.8 (Inclusion Lemma) *For all processes p_1, p_2,*

$$\textit{Whenever } p_1 \equiv p_2 \textit{ then } p_1 \sim p_2$$

Proof

Left for the reader.

∎

As another property, we shall show that the relation $\smile$ between configurations is a kind of congruence. Let $W : P\sharp Q \overset{def}{=} (CV[P] - W) \cap (CV[Q] - W) = \emptyset$. That is, $W : P\sharp Q$ if P and Q do not communicate via internal channels. We can now state the congruence lemma:

Lemma 5.9 (Configuration congruence) *For all configurations $(W_1, K_1) \triangleright P_1 \cup Q_1$ and $(W_2, K_2) \triangleright P_2 \cup Q_2$,*

$$\begin{array}{ll} \textit{Whenever} & (W_1, K_1) \triangleright P_1 \smile (W_2, K_2) \triangleright P_2 \\ \textit{and} & (W_1, K_1) \triangleright Q_1 \smile (W_2, K_2) \triangleright Q_2 \\ \textit{and} & W_1 : P_1 \sharp Q_1 \\ \textit{and} & W_2 : P_2 \sharp Q_2 \\ & \\ \textit{then} & (W_1, K_1) \triangleright P_1 \cup Q_1 \smile (W_2, K_2) \triangleright P_2 \cup Q_2 \end{array}$$

Proof

Left for the reader.

∎

The following lemma states that we can extend the environment (W, K) of a configuration with a set E of channels, obtaining $(W \cup E, K \cup E)$, if none of the channels in E are local in P wrt. W.

Lemma 5.10 (Window extension) *For any configuration $(W, K) \triangleright P$ and for any $L \subseteq \mathit{Chan}$,*

$$\begin{array}{ll} \textit{Whenever} & L \cap (CV[P] - W) = \emptyset \\ \textit{then} & (W, K) \triangleright P \smile (W \cup L, K \cup L) \triangleright P \end{array}$$

Proof

Left for the reader.

■

The following lemma states that $\smile$ basically is a congruence wrt. window restriction.

Lemma 5.11 (Window restriction) *For all configurations* $(W_1, K_1) \triangleright P_1$ *and* $(W_2, K_2) \triangleright P_2$*, and for any* $L \subseteq$ *Chan,*

$$\begin{array}{ll} \textit{Whenever} & (W_1, K_1) \triangleright P_1 \smile (W_2, K_2) \triangleright P_2 \\ \textit{then} & (W_1 - L, K_1) \triangleright P_1 \smile (W_2 - L, K_2) \triangleright P_2 \end{array}$$

Proof

We show that the following relation is a bisimulation:

$$S \;=\; \{((W_1 - L, K_1) \triangleright P_1, (W_2 - L, K_2) \triangleright P_2) \mid (W_1, K_1) \triangleright P_1 \smile (W_2, K_2) \triangleright P_2\}$$

Let $((W_1 - L, K_1) \triangleright P_1, (W_2 - L, K_2) \triangleright P_2) \in S$ and assume that $(W_1 - L, K_1) \triangleright P_1 \xrightarrow{\alpha} (W_1 - L, K_1') \triangleright P_1'$. Then we know that $W_1 - L$ *allows* α and thereby that W_1 *allows* α. We also know that $K_1 \triangleright P_1 \xrightarrow{\alpha} K_1' \triangleright P_1'$. These two facts means that $(W_1, K_1) \triangleright P_1 \xrightarrow{\alpha} (W_1, K_1') \triangleright P_1'$. Since $(W_1, K_1) \triangleright P_1 \smile (W_2, K_2) \triangleright P_2$ we conclude that $(W_2, K_2) \triangleright P_2 \xrightarrow{\alpha} (W_2, K_2') \triangleright P_2'$ where $(W_1, K_1') \triangleright P_1' \smile (W_2, K_2') \triangleright P_2'$. This also means that W_2 *allows* α and thereby that $W_2 - L$ *allows* α since $\alpha \notin L$. Also, of course, $K_2 \triangleright P_2 \xrightarrow{\alpha} K_2' \triangleright P_2'$. Consequently $(W_2 - L, K_2) \triangleright P_2 \xrightarrow{\alpha} (W_2 - L, K_2') \triangleright P_2'$ and $((W_1 - L, K_1') \triangleright P_1', (W_2 - L, K_2') \triangleright P_2') \in S$.

A symmetric argument starting with $(W_2 - L, K_2) \triangleright P_2$ ends the proof.

■

The last lemma we shall state before showing $\equiv$ to be a congruence is that $\smile$ in certain cases is a congruence wrt. substitution. Let us first define what we mean by substitution on configurations. Assume a substitution σ : *Chan* $\rightarrow$ *Chan*. Then substitution on a configuration is defined as follows: $((W, K) \triangleright P)[\sigma] \stackrel{def}{=} (W[\sigma], K[\sigma]) \triangleright P[\sigma]$. A substitution applied to a set of elements is effectuated by applying the substitution to each of the elements in the set. Our substitution lemma is then as follows.

Lemma 5.12 (Configuration substitution) *For all configurations C_1, C_2 and for any substitution $\sigma : Chan \rightarrow Chan$ where $\sigma(a) \neq a \Rightarrow \sigma(a) \notin CV[C_1] \cup CV[C_2]$,*

$$\begin{array}{ll} \textit{Whenever} & C_1 \smile C_2 \\ \textit{then} & C_1[\sigma] \smile C_2[\sigma] \end{array}$$

Proof

Left for the reader.

∎

Finally we can show that $\equiv$ is a congruence, i.e. $\equiv$ is preserved by the operators of FC_r.

Theorem 5.13 (Congruence Property) *Assume for two processes p_1, p_2 that $p_1 \equiv p_2$, and that for the processes p_i, p_i' ($i \in I$ for some index set I) that $p_i \equiv p_i'$. Then for any process q:*

$$\sum_i \alpha_i; p_i \equiv \sum_i \alpha_i; p_i' \quad (5.1)$$
$$q; p_1 \equiv q; p_2 \quad (5.2)$$
$$p_1; q \equiv p_2; q \quad (5.3)$$
$$\boldsymbol{fork}(p_1) \equiv \boldsymbol{fork}(p_2) \quad (5.4)$$
$$\boldsymbol{forked}(p_1) \equiv \boldsymbol{forked}(p_2) \quad (5.5)$$
$$(a)p_1 \equiv (a)p_2 \quad (5.6)$$
$$\{a\}p_1 \equiv \{a\}p_2 \quad (5.7)$$

Proof

For each equation $t_1 \equiv t_2$ we shall prove $\mathit{Config}[t_1] \smile \mathit{Config}[t_2]$. Note that our assumption is that $p_1 \equiv p_2$ and therefore that $\mathit{Config}[p_1] \smile \mathit{Config}[p_2]$. We also assume that $p_i \equiv p_i'$ for all i in some index set I.

In the proof we shall use an unlabelled transition relation $\Longrightarrow \subseteq Con \times Con$ which is the least relation closed under the following rules:

$$(1) \qquad \frac{p \overset{\Phi(r)}{\hookrightarrow} p'}{\rho \triangleright P \cup \{\!|p|\!\} \Longrightarrow \rho \triangleright P \cup \{\!|p', r|\!\}}$$

$$(2) \qquad \frac{p \overset{\Lambda(k)}{\hookrightarrow} p'}{(W, K) \triangleright P \cup \{\!|p|\!\} \Longrightarrow (W, K \cup \{k\}) \triangleright P \cup \{\!|p'|\!\}} \; k \notin K$$

$$(3) \qquad \frac{}{\rho \triangleright P \Longrightarrow \rho \triangleright P}$$

$$(4) \qquad \frac{\rho \triangleright P \Longrightarrow \rho' \triangleright P'\ ,\ \rho' \triangleright P' \Longrightarrow \rho'' \triangleright P''}{\rho \triangleright P \Longrightarrow \rho'' \triangleright P''}$$

As an example, consider the following evaluation:

$$\begin{array}{lcl} (W, K) \triangleright \{\!|\{a\}\boldsymbol{forked}(a?); a!|\!\} & \Longrightarrow & (W, K \cup \{k\}) \triangleright \{\!|\boldsymbol{forked}(k?); k!|\!\} \quad \Longrightarrow \\ (W, K \cup \{k\}) \triangleright \{\!|\boldsymbol{nil}; k!, k?|\!\} & \xrightarrow{\tau} & (W, K \cup \{k\}) \triangleright \{\!|\boldsymbol{nil}, \boldsymbol{nil}|\!\} \end{array}$$

(5.1) $\sum_i \alpha_i; p_i \equiv \sum_i \alpha_i; p'_i$

We show that $Config[\sum_i \alpha_i; p_i] \smile Config[\sum_i \alpha_i; p'_i]$.

Let $\rho \stackrel{def}{=} Env[\sum_i \alpha_i; p_i]$ and let $\rho' \stackrel{def}{=} Env[\sum_i \alpha_i; p'_i]$. Assume that $Config[\sum_i \alpha_i; p_i] = \rho \triangleright \{\!|\sum_i \alpha_i; p_i; \pi|\!\} \xrightarrow{\alpha_i} \rho \triangleright \{\!|p_i; \pi|\!\}$. We also have that $Config[\sum_i \alpha_i; p'_i] = \rho' \triangleright \{\!|\sum_i \alpha_i; p'_i; \pi|\!\} \xrightarrow{\alpha_i} \rho' \triangleright \{\!|p'_i; \pi|\!\}$. Due to the extension lemma 5.10 we have that $\rho \triangleright \{\!|p_i; \pi|\!\} \smile Env[p_i] \triangleright \{\!|p_i; \pi|\!\} = Config[p_i]$ as well as $\rho' \triangleright \{\!|p_i; \pi|\!\} \smile Env[p'_i] \triangleright \{\!|p'_i; \pi|\!\} = Config[p'_i]$. Since further $p_i \equiv p'_i$ means that $Config[p_i] \smile Config[p'_i]$ we have that $\rho \triangleright \{\!|p_i; \pi|\!\} \smile \rho' \triangleright \{\!|p'_i; \pi|\!\}$.

The case where we start with $Config[\sum_i \alpha_i; p'_i]$ is similar.

(5.2) $q; p_1 \equiv q; p_2$

We show that $Config[q; p_1] \smile Config[q; p_2]$. We will show that the following relation S is a bisimulation:

$$\begin{array}{rcl} S & = & \{(\rho \triangleright \{\!|q; p_1; \pi|\!\} \cup R, \rho \triangleright \{\!|q; p_2; \pi|\!\} \cup R) \mid \\ & & \qquad p_1 \equiv p_2 \ \wedge \\ & & \qquad CV[p_i] \subseteq W_\rho \ \wedge \\ & & \qquad (\rho \triangleright \{\!|q; p_i; \pi|\!\} \cup R) \in Con,\ i \in \{1, 2\}\} \\ & & \cup \smile \end{array}$$

Before we go on to show that S is a bisimulation, we prove (5.2) under that assumption.

Assume that $p_1 \equiv p_2$. Now let $\rho \stackrel{def}{=} Env[q; p_1] \cup Env[q; p_2]$. Then surely $(\rho \triangleright \{\!|q; p_1; \pi|\!\}, \rho \triangleright \{\!|q; p_2; \pi|\!\}) \in S$, and consequently $\rho \triangleright \{\!|q; p_1; \pi|\!\} \smile \rho \triangleright \{\!|q; p_2; \pi|\!\}$ since we assume that S is a bisimulation. Due to the extension lemma (5.10) we have that $Config[q; p_1] \smile \rho \triangleright \{\!|q; p_1; \pi|\!\}$ and $Config[q; p_2] \smile \rho \triangleright \{\!|q; p_2; \pi|\!\}$. This implies that $Config[q; p_1] \smile Config[q; p_2]$, and therefore $q; p_1 \equiv q; p_2$.

We now show that S is a bisimulation. We don't consider the pairs in $\smile$ since $\smile$ is known to be a bisimulation. Let $(\rho \triangleright \{\!|q; p_1; \pi|\!\} \cup R, \rho \triangleright \{\!|q; p_2; \pi|\!\} \cup R) \in S$ and assume that $\rho \triangleright \{\!|q; p_1; \pi|\!\} \cup R \xrightarrow{\alpha} \rho' \triangleright P_1$. Now there are two cases to analyse, depending on whether q during the evaluation reaches a state q' where $Stop(q')$ (q gets exhausted) or not.

Case 1: q gets exhausted Since q gets exhausted it must hold that $\rho \triangleright \{\!|q; p_1; \pi|\!\} \cup R \Longrightarrow \rho^\star \triangleright \{\!|q'; p_1; \pi|\!\} \cup R' \xrightarrow{\alpha} \rho' \triangleright P_1$ where $Stop(q')$. But then $\rho \triangleright \{\!|q; p_2; \pi|\!\} \cup R \Longrightarrow \rho^\star \triangleright \{\!|q'; p_2; \pi|\!\} \cup R'$ and since $Stop(q')$ it is easy verifiable that $\rho^\star \triangleright \{\!|q'; p_1; \pi|\!\} \smile \rho^\star \triangleright \{\!|q'; p_2; \pi|\!\}$. We can verify this by first observing that $Config[p_1] \smile Config[p_2]$, and then by applying the extension lemma (5.10) to obtain that for $i \in \{1, 2\}$: $Config[p_i] \smile \rho^\star \triangleright \{\!|p_i; \pi|\!\}$. Note that we here use the fact that $CV[p_i] \subseteq W_\rho$ (a condition for being a member of S). This namely ensures the validity of applying lemma (5.10): p_i has no channels local wrt. ρ that become visible wrt. $\rho^\star$.

We can now apply the congruence lemma (5.9) to obtain that $\rho^\star \triangleright \{\!|q'; p_1; \pi|\!\} \cup R' \smile \rho^\star \triangleright \{\!|q'; p_2; \pi|\!\} \cup R'$. Note that also here use the fact that $CV[p_i] \subseteq W_\rho$ (condition for being a member of S). This namely implies that $CV[p_i] \subseteq W_{\rho^\star}$ (since the W component is unchanged) and from this we obtain the condition for applying the congruence lemma (5.9): $\rho^\star : \{\!|q'; p_i; \pi|\!\} \sharp R'$. Now since $\rho^\star \triangleright \{\!|q'; p_1; \pi|\!\} \cup R' \xrightarrow{\alpha} \rho' \triangleright P_1$ we therefore get that $\rho^\star \triangleright \{\!|q'; p_2; \pi|\!\} \cup R' \xrightarrow{\alpha} \rho'' \triangleright P_2$ where $\rho' \triangleright P_1 \smile \rho'' \triangleright P_2$. Finally $\rho \triangleright \{\!|q; p_2; \pi|\!\} \cup R \xrightarrow{\alpha} \rho'' \triangleright P_2$ and $(\rho' \triangleright P_1, \rho'' \triangleright P_2) \in S$.

Case 2: q does not get exhausted Assume that $\rho \triangleright \{\!|q; p_1; \pi|\!\} \cup R \xrightarrow{\alpha} \rho' \triangleright \{\!|q'; p_1; \pi|\!\} \cup R'$. Then obviously $\rho \triangleright \{\!|q; p_2; \pi|\!\} \cup R \xrightarrow{\alpha} \rho' \triangleright \{\!|q'; p_2; \pi|\!\} \cup R'$, and clearly $(\rho' \triangleright \{\!|q'; p_1; \pi|\!\} \cup R', \rho' \triangleright \{\!|q'; p_2; \pi|\!\} \cup R') \in S$. Note that one of the conditions restricting the pairs of S is that $CV[p_i] \subseteq W_{\rho'}$. But this holds since $CV[p_i] \subseteq W_\rho$ and the W component is unchanged during evaluation.

A symmetric argument starting with $\rho \triangleright \{\!|q; p_2; \pi|\!\} \cup R$ ends the proof.

(5.3) $p_1; q \equiv p_2; q$

We show that $\mathit{Config}[p_1; q] \smile \mathit{Config}[p_2; q]$. We will show that the following relation S is a bisimulation:

$$\begin{array}{rcl} S & = & \{(\rho_1 \triangleright \{\!|p_1; q; \pi|\!\} \cup R_1, \rho_2 \triangleright \{\!|p_2; q; \pi|\!\} \cup R_2) \mid \\ & & \quad \rho_1 \triangleright \{\!|p_1; \pi|\!\} \cup R_1 \smile \rho_2 \triangleright \{\!|p_2; \pi|\!\} \cup R_2 \ \wedge \\ & & \quad CV[q] \subseteq W_{\rho_i} \ \wedge \\ & & \quad (\rho_i \triangleright \{\!|p_i; q; \pi|\!\} \cup R_i) \in \mathit{Con},\ i \in \{1, 2\}\} \\ & & \cup \smile \end{array}$$

Before we go on to show that S is a bisimulation, we prove (5.3) under that assumption.

Assume that $p_1 \equiv p_2$. Now, let $\rho \stackrel{def}{=} \mathit{Env}[p_1; q] \cup \mathit{Env}[p_2; q]$. Then due to the extension lemma (5.10), we have that $\rho \triangleright \{\!|p_i; \pi|\!\} \smile \mathit{Config}[p_i]$ for $i \in \{1, 2\}$. But since $\mathit{Config}[p_1] \smile \mathit{Config}[p_2]$ this implies that $\rho \triangleright \{\!|p_1; \pi|\!\} \smile \rho \triangleright \{\!|p_2; \pi|\!\}$. Then we conclude that $(\rho \triangleright \{\!|p_1; q; \pi|\!\}, \rho \triangleright \{\!|p_2; q; \pi|\!\}) \in S$, and assuming that S is a bisimulation, we conclude further that $\rho \triangleright \{\!|p_1; q; \pi|\!\} \smile \rho \triangleright \{\!|p_2; q; \pi|\!\}$. Again due to the extension lemma (5.10), we have that for $i \in \{1, 2\}$: $\mathit{Config}[p_i; q] \smile \rho \triangleright \{\!|p_i; q; \pi|\!\}$. It follows that $\mathit{Config}[p_1; q] \smile \mathit{Config}[p_2; q]$ and thereby that $p_1; q \equiv p_2; q$.

We now show that S is a bisimulation. We don't consider the pairs in $\smile$ since $\smile$ is known to be a bisimulation. Let $(\rho_1 \triangleright \{\!|p_1; q; \pi|\!\} \cup R_1, \rho_2 \triangleright \{\!|p_2; q; \pi|\!\} \cup R_2) \in S$ and assume that $\rho_1 \triangleright \{\!|p_1; q; \pi|\!\} \cup R_1 \xrightarrow{\alpha} \rho_1' \triangleright P_1$. Now there are two cases to analyse, depending on whether p_1 during the evaluation reaches a state p_1' where $\mathit{Stop}(p_1')$ (p_1 gets exhausted) or not.

Case 1: p_1 gets exhausted Since p_1 gets exhausted it must hold that: $\rho_1 \triangleright \{\!|p_1; q; \pi|\!\} \cup R_1 \Longrightarrow \rho_1^\star \triangleright \{\!|p_1'; q; \pi|\!\} \cup R_1' \xrightarrow{\alpha} \rho_1' \triangleright P_1$ where $\mathit{Stop}(p_1')$. But then $\rho_1 \triangleright \{\!|p_1; \pi|\!\} \cup R_1 \Longrightarrow \rho_1^\star \triangleright \{\!|p_1'; \pi|\!\} \cup R_1' \xrightarrow{\pi} \rho_1^\star \triangleright \{\!|\boldsymbol{nil}|\!\} \cup R_1'$. Since $\rho_1 \triangleright \{\!|p_1; \pi|\!\} \cup R_1 \smile \rho_2 \triangleright \{\!|p_2; \pi|\!\} \cup R_2$ it must then hold that $\rho_2 \triangleright \{\!|p_2; \pi|\!\} \cup R_2 \Longrightarrow \rho_2^\star \triangleright \{\!|p_2'; \pi|\!\} \cup R_2' \xrightarrow{\pi} \rho_2^\star \triangleright \{\!|\boldsymbol{nil}|\!\} \cup R_2'$ where $\mathit{Stop}(p_2')$ and $\rho_1^\star \triangleright \{\!|\boldsymbol{nil}|\!\} \cup R_1' \smile \rho_2^\star \triangleright \{\!|\boldsymbol{nil}|\!\} \cup R_2'$. This means of course that $\rho_1^\star \triangleright R_1' \smile \rho_2^\star \triangleright R_2'$. We can further conclude that $\rho_2 \triangleright \{\!|p_2; q; \pi|\!\} \cup R_2 \Longrightarrow \rho_2^\star \triangleright \{\!|p_2'; q; \pi|\!\} \cup R_2'$. The congruence lemma (5.9) now yields that $\rho_1^\star \triangleright \{\!|p_1'; q; \pi|\!\} \cup R_1' \smile \rho_2^\star \triangleright \{\!|p_2'; q; \pi|\!\} \cup R_2'$. This can be seen by observing the following facts:

1. $\rho_1^\star \triangleright R_1' \smile \rho_2^\star \triangleright R_2'$
2. $\rho_1^\star \triangleright \{\!|p_1'; q; \pi|\!\} \smile \rho_2^\star \triangleright \{\!|p_2'; q; \pi|\!\}$. This follows from observing that $\mathit{Stop}(p_1')$ and $\mathit{Stop}(p_2')$ and from the fact that $\mathit{Config}[q] \smile \mathit{Config}[q]$: apply the extension lemma (5.10) to show that $\rho_i^\star \triangleright \{\!|q; \pi|\!\} \smile \mathit{Config}[q]$ — note that $CV[q] \subseteq W_{\rho_i}$ (from condition in S), so q has no local channels neither wrt. ρ_1 or ρ_2 nor wrt. $\rho_1^\star$ or $\rho_2^\star$.

3. $\rho_1^\star : \{|p_1'; q; \pi|\} \sharp R_1'$. This is a consequence of $CV[q] \subseteq W_{\rho_1}$ (from condition in S) which implies that $CV[q] \subseteq W_{\rho_1^\star}$ since $W_{\rho_1} = W_{\rho_1^\star}$.
4. $\rho_2^\star : \{|p_2'; q; \pi|\} \sharp R_2'$.

Now,since $\rho_1^\star \triangleright \{|p_1'; q; \pi|\} \cup R_1' \xrightarrow{\alpha} \rho_1' \triangleright P_1$ also $\rho_2^\star \triangleright \{|p_2'; q; \pi|\} \cup R_2' \xrightarrow{\alpha} \rho_2' \triangleright P_2$ where $\rho_1' \triangleright P_1 \smile \rho_2' \triangleright P_2$. But this means that $\rho_2 \triangleright \{|p_2; q; \pi|\} \cup R_2 \xrightarrow{\alpha} \rho_2' \triangleright P_2$ and $(\rho_1' \triangleright P_1, \rho_2' \triangleright P_2) \in S$.

Case 2: p_1 does not get exhausted Assume that $\rho_1 \triangleright \{|p_1; q; \pi|\} \cup R_1 \xrightarrow{\alpha} \rho_1' \triangleright \{|p_1'; q; \pi|\} \cup R_1'$. But then $\rho_1 \triangleright \{|p_1; \pi|\} \cup R_1 \xrightarrow{\alpha} \rho_1' \triangleright \{|p_1'; \pi|\} \cup R_1'$. Since $\rho_1 \triangleright \{|p_1; \pi|\} \cup R_1 \smile \rho_2 \triangleright \{|p_2; \pi|\} \cup R_2$ we also have that $\rho_2 \triangleright \{|p_2; \pi|\} \cup R_2 \xrightarrow{\alpha} \rho_2' \triangleright \{|p_2'; \pi|\} \cup R_2'$ where $\rho_1' \triangleright \{|p_1'; \pi|\} \cup R_1' \smile \rho_2' \triangleright \{|p_2'; \pi|\} \cup R_2'$. But then $\rho_2 \triangleright \{|p_2; q; \pi|\} \cup R_2 \xrightarrow{\alpha} \rho_2' \triangleright \{|p_2'; q; \pi|\} \cup R_2'$ and finally $(\rho_1' \triangleright \{|p_1'; q; \pi|\} \cup R_1', \rho_2' \triangleright \{|p_2'; q; \pi|\} \cup R_2') \in S$.

A symmetric argument starting with $\rho_2 \triangleright \{|p_2; q; \pi|\} \cup R_2$ ends the proof.

(5.4) ***fork***$(p_1) \equiv$ ***fork***(p_2)

The proof is similar to that of theorem 3.11 (3.1). The proof, however, uses the new inclusion lemma (5.8) and the new congruence lemma (5.9).

(5.5) ***forked***$(p_1) \equiv$ ***forked***(p_2)

The proof is similar to that of theorem 4.1 (4.4). The proof uses the inclusion lemma (5.8) and the congruence lemma (5.9).

(5.6) $(a)p_1 \equiv (a)p_2$

We show that $\mathit{Config}[(a)p_1] \smile \mathit{Config}[(a)p_2]$. Let $(W, K) \stackrel{def}{=} \mathit{Env}[(a)p_1] \cup \mathit{Env}[(a)p_2]$. Due to the extension lemma (5.10) we have that $\mathit{Config}[(a)p_1] \smile (W, K) \triangleright \{|(a)p_1; \pi|\}$ and $\mathit{Config}[(a)p_2] \smile (W, K) \triangleright \{|(a)p_2; \pi|\}$. So we want to show that $(W, K) \triangleright \{|(a)p_1; \pi|\} \smile (W, K) \triangleright \{|(a)p_2; \pi|\}$. Let S^x stand for $S \cup \{x\}$, for an arbitrary set S and element x. Assume that $(W,K) \triangleright \{\!|(a)p_1;\pi|\!\} \xrightarrow{\tau} (W,K^k) \triangleright \{\!|p_1[k/a];\pi|\!\}$. We also have that $(W,K) \triangleright \{\!|(a)p_2;\pi|\!\} \xrightarrow{\tau} (W,K^k) \triangleright \{\!|p_2[k/a];\pi|\!\}$. So we are now left to show that $(W,K^k) \triangleright \{\!|p_1[k/a];\pi|\!\} \smile (W,K^k) \triangleright \{\!|p_2[k/a];\pi|\!\}$. Since $p_1 \equiv p_2$ we have that $Config[p_1] \smile Config[p_2]$. Further, due to the extension lemma (5.10), we have that $(W^a,K^a) \triangleright \{\!|p_1;\pi|\!\} \smile Config[p_1]$ and $(W^a,K^a) \triangleright \{\!|p_2;\pi|\!\} \smile Config[p_2]$; and thereby that $(W^a,K^a) \triangleright \{\!|p_1;\pi|\!\} \smile (W^a,K^a) \triangleright \{\!|p_2;\pi|\!\}$. The restriction lemma (5.11) then yields that $(W,K^a) \triangleright \{\!|p_1;\pi|\!\} \smile (W,K^a) \triangleright \{\!|p_2;\pi|\!\}$. Finally, the substitution lemma (5.12) yields that $(W,K^k) \triangleright \{\!|p_1[k/a];\pi|\!\} \smile (W,K^k) \triangleright \{\!|p_2[k/a];\pi|\!\}$.

A symmetric argument where $(W,K) \triangleright \{\!|(a)p_2;\pi|\!\}$ moves first ends the proof.

(5.7) $\{a\}p_1 \equiv \{a\}p_2$

We show that $Config[\{a\}p_1] \smile Config[\{a\}p_2]$. Let $(W,K) \stackrel{def}{=} Env[\{a\}p_1] \cup Env[\{a\}p_2]$. Due to the extension lemma (5.10) we have that $Config[\{a\}p_1] \smile (W,K) \triangleright \{\!|\{a\}p_1;\pi|\!\}$ and $Config[\{a\}p_2] \smile (W,K) \triangleright \{\!|\{a\}p_2;\pi|\!\}$. So we want to show that $(W,K) \triangleright \{\!|\{a\}p_1;\pi|\!\} \smile (W,K) \triangleright \{\!|\{a\}p_2;\pi|\!\}$. Assume that $(W,K) \triangleright \{\!|\{a\}p_1;\pi|\!\} \xrightarrow{\alpha} (W,K') \triangleright P_1$. Then we have that $(W,K) \triangleright \{\!|\{a\}p_1;\pi|\!\} \Longrightarrow (W,K^k) \triangleright \{\!|p_1[k/a];\pi|\!\} \xrightarrow{\alpha} (W,K') \triangleright P_1$. We also have that $(W,K) \triangleright \{\!|\{a\}p_2;\pi|\!\} \Longrightarrow (W,K^k) \triangleright \{\!|p_2[k/a];\pi|\!\}$. By a reasoning similar to the previous case (5.6), we can deduce that $(W,K^k) \triangleright \{\!|p_1[k/a];\pi|\!\} \smile (W,K^k) \triangleright \{\!|p_2[k/a];\pi|\!\}$. So since $(W,K^k) \triangleright \{\!|p_1[k/a];\pi|\!\} \xrightarrow{\alpha} (W,K') \triangleright P_1$ also $(W,K^k) \triangleright \{\!|p_2[k/a];\pi|\!\} \xrightarrow{\alpha} (W,K'') \triangleright P_2$ and $(W,K') \triangleright P_1 \smile (W,K'') \triangleright P_2$. Finally it follows that $(W,K) \triangleright \{\!|\{a\}p_2;\pi|\!\} \xrightarrow{\alpha} (W,K'') \triangleright P_2$.

A symmetric argument where $(W,K) \triangleright \{\!|\{a\}p_2;\pi|\!\}$ moves first ends the proof.

■

We do not give a congruence result for the ***fix*** $x \cdot \_$ construct. The reason is that this would involve a more elaborated notion of equivalence between terms with free variables and, in addition, the congruence proof gets even more complicated. When defining the logic in a later section, we don't seem to need this congruence property since other means of proving properties about fixpoints are provided.

Chapter 6

Reasoning about Programs in the New Setting

Analog to what we did for the basic Fork Calculus, this chapter presents two different theories for reasoning about processes: an axiomatisation, section 6.1, and a logic, section 6.2. The logic will though be quite different from the logic defined for FC in section 4.2. The major difference is that, in the new logic, processes will be special cases of formulae, since formulae may include a mixture of programming constructs and logic constructs. Such a logic allows compositional reasoning in the sense that if for example $p_1 \models \psi_1$ and $p_2 \models \psi_2$, then $p_1; p_2 \models \psi_1; \psi_2$.

6.1 Axiomatisation of Strong Equivalence

In this section we present a sound and complete axiomatisation for the process congruence $\equiv$. We begin with a motivation for having two restriction operators. The reason is the same as it is for having two forking operators: to be able to axiomatise strong congruence; strong in the sense that the τ action is observable.

6.1.1 Searching for an Expansion Law

In this section we motivate the existence of two restriction operators $\{\_\}_\_$ and $(\_)_\_$. Suppose we only have $(\_)_\_$. We will need axioms involving the restriction operator $(\_)_\_$. Basically, we need an expansion law, that describes how restriction may be expanded away — similar to the expansion law of CCS that expands away restriction. For example, consider the following instance of the CCS expansion law:

$$(a.p)\backslash b \quad = \quad a.(p\backslash b)$$

Let us try to search for a similar expansion law for the $(\_)\_$ operator. We will thus try to expand the corresponding process $(b)(a;p)$. An initial guess can be the following:

$$(b)(a;p) \quad = \quad a;(b)p \tag{6.1}$$

However, this equation is not sound (it does not hold when replacing $=$ with $\equiv$). The reason is that the left hand side can perform a τ action as the first thing, while the right hand side can perform an a action (note that since we want to axiomatise strong process congruence $\equiv$, τ actions are observable). Trying to repair this problem we obtain:

$$(b)(a;p) \quad = \quad \tau;a;(b)p \tag{6.2}$$

This equation is not sound either: the left hand side can perform one τ action, while the right hand side can perform two (the explicitely mentioned and the one caused by the now inner restriction).

So it seems that we cannot in general expand restriction. The latter two attempts above suggest that we need a version of restriction which is instantaneous without the initial internal transition caused by $(\_)\_$. Put alternatively, we lack the ability to express that a channel *has been* allocated, with the initial τ action already having occurred previously. To gain this expressivity, we have added the instantaneous restriction operator $\{\_\}\_$ to our calculus. There is the following relationship between the two restriction operators:

$$(a)p \quad = \quad \tau;\{a\}p$$

Let us now try to write new versions of the equations (6.1) and (6.2), that contain the $\{\_\}\_$ operator in appropriate positions. These equations can be shown sound on basis of the formal definition we have given of the $\{\_\}\_$ operator. The equations are:

$$\begin{aligned} \{b\}(a;p) \quad &= \quad a;\{b\}p \\ (b)(a;p) \quad &= \quad \tau;a;\{b\}p \end{aligned}$$

6.1.2 Basic Axioms and Inference Rules

The axiom system $\mathcal{AS}$ consists of a set $\mathcal{AX}$ of equational axioms, three expansion laws $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$, using normal forms, and a set $\mathcal{I}$ of inference rules. In this section we shall present the basic equational axioms $\mathcal{AX}$ and the the inference rules $\mathcal{I}$.

Definition 6.1 (The basic axioms $\mathcal{AX}$) *The set $\mathcal{AX}$ contains the following axioms:*

$$\begin{aligned} p; \boldsymbol{nil} &= p && (6.3)\\ \boldsymbol{nil}; p &= p && (6.4)\\ (p;q);r &= p;(q;r) && (6.5)\\ (\sum_i \alpha_i; p_i); q &= \sum_i \alpha_i; (p_i; q) && (6.6) \end{aligned}$$

$$\sum_{i \in I} \alpha_i; p_i = \sum_{i \in I - \{j\}} \alpha_i; p_i \quad \text{– provided } \exists i \in I - \{j\} \cdot (\alpha_i; p_i) = (\alpha_j; p_j) \qquad (6.7)$$

$$\begin{aligned} \boldsymbol{fork}(p) &= \tau; \boldsymbol{forked}(p) && (6.8)\\ \boldsymbol{forked}(\alpha; \boldsymbol{forked}(p) + \sum_{\dots} \dots) &= \boldsymbol{forked}(\alpha; p + \sum_{\dots} \dots) && (6.9)\\ \boldsymbol{forked}(\boldsymbol{nil}) &= \boldsymbol{nil} && (6.10) \end{aligned}$$

$$(a)p = \tau; \{a\}p \qquad (6.11)$$

Proposition 6.2 ($\mathcal{AX}$ is sound) *The axioms $\mathcal{AX}$ are sound with respect to strong process congruence.*

Proof

Left to the reader.

■

Note that axiom (6.7) is the only one that says something about equality between pure choice terms. It explains how one of the entries, $\alpha_j; p_j$, in a choice may be eliminated, if there exists another entry, $\alpha_i; p_i$, such that $\alpha_i; p_i$ is identical to $\alpha_j; p_j$. For example $\alpha; p = \alpha; p + \alpha; p$. From this axiom we can further deduce that choice is commutative. That is, assume a permutation $\sigma : I \rightarrow I$ of the elements in I. Then the following derived property says that we can permute the elements in a choice without changing the meaning (commutativity):

$$\sum_{i \in I} \alpha_i; p_i = \sum_{i \in I} \alpha_{\sigma(i)}; p_{\sigma(i)} \qquad (6.12)$$

Proof

Assume an index set I, assume a choice term $\sum_{i \in I} \alpha_i; p_i$ ranging over I, and assume a permutation $\sigma : I \rightarrow I$. Define now another index set N isomorphic to I via the isomorphism $\eta : N \rightarrow I$ and with no elements in common with I. Due to axiom (6.7), which says that we can always add elements to a choice if they already exist and vice versa (for removing elements), it holds that:

$$\begin{array}{l} \sum_{i \in I} \alpha_i; p_i \\ = \\ \sum_{i \in I} \alpha_i; p_i + \sum_{n \in N} \alpha_{\eta(n)}; p_{\eta(n)} \\ = \\ \sum_{n \in N} \alpha_{\eta(n)}; p_{\eta(n)} \\ = \\ \sum_{n \in N} \alpha_{\eta(n)}; p_{\eta(n)} + \sum_{i \in I} \alpha_{\sigma(i)}; p_{\sigma(i)} \\ = \\ \sum_{i \in I} \alpha_{\sigma(i)}; p_{\sigma(i)} \end{array}$$

∎

Another property which can be proved in the same manner as we have just shown commutativity above is that the choice of index set is of no importance to the meaning.

Definition 6.3 (The inference rules $\mathcal{I}$) *The inference rules $\mathcal{I}$ are obtained by adding the following congruence rules to the original ones for equivalence and congruence wrt. forking and sequential composition:*

$$(\textbf{Congruence}_6) \quad \frac{p_i = p_i'}{\sum_i \alpha_i; p_i = \sum_i \alpha_i; p_i'}$$

$$(\textbf{Congruence}_7) \quad \frac{p_1 = p_2}{(a)p_1 = (a)p_2}$$

$$(\textbf{Congruence}_8) \quad \frac{p_1 = p_2}{\{a\}p_1 = \{a\}p_2}$$

Proposition 6.4 ($\mathcal{I}$ is sound) *The inference rules $\mathcal{I}$ are sound with respect to strong process congruence.*

Proof The rules must hold when replacing $=$ with $\equiv$. For the equivalence rules, this is the case as stated in section 5.2. Concerning the remaining congruence rules, theorem 5.13 states this. ∎

6.1.3 A New Expansion Law

The normal forms are as for FC. That is, restriction can always be expanded away via the expansion law that we shall introduce below.

Recall that the two expansion laws $\mathcal{E}_1$ and $\mathcal{E}_2$ enabelled us to transform any process term into normal form. Since normal forms do not involve restriction, we shall in addition need a third expansion law for transforming restriction away. We therefore introduce the following expansion law.

Definition 6.5 (Expansion law $\mathcal{E}_3$)

$$\begin{array}{l} \{a\}(\sum_i \alpha_i; N_i + \sum_j \beta_j; \textbf{\textit{forked}}(B_j)) \\ = \\ \sum_{\alpha_i \neq a} \alpha_i; \{a\} N_i + \sum_{\beta_j \neq a} \beta_j; \textbf{\textit{forked}}(\{a\} B_j) \end{array}$$

In order to prove the soundness of expansion law $\mathcal{E}_3$, we need two lemmas which we shall state here. The two lemmas basically says that configurations are equivalent up to renaming of local channels (channels which are in the K component but not in the W component). The first lemma says this for simple normal forms while the second extends to (full) normal forms.

Lemma 6.6 (Renaming of simple normal forms) *For any two configurations* $(W, K \cup \{k\}) \triangleright \{\!|A[k/a]|\!\}$ *and* $(W, K \cup \{l\}) \triangleright \{\!|A[l/a]|\!\}$ *where* A *is on simple normal form and where* $k, l \notin W$ *and where* $k, l \notin CV[A]$,

$$(W, K \cup \{k\}) \triangleright \{\!|A[k/a]|\!\} \smile (W, K \cup \{l\}) \triangleright \{\!|A[l/a]|\!\}$$

Proof

We use induction on the depth n of A.

<u>**Base case $n = 0$**</u>: In this case A is ***nil***, and the property obviously holds.

<u>**Induction step $n > 0$**</u>: Then A has the form $\sum_i \alpha_i; A_i$. Assume that $(W, K \cup \{k\}) \triangleright \{\!| \sum_i \alpha_i; A_i[k/a]|\!\} \xrightarrow{\alpha} (W, K \cup \{k\}) \triangleright \{\!|A_i[k/a]|\!\}$. Then obviously $\alpha \neq k$ so $\alpha = \alpha_i$ for some i where α_i contains no a. Consequently $\alpha_i \in W$ and therefore $(W, K \cup \{l\}) \triangleright \{\!| \sum_i \alpha_i; A_i[l/a]|\!\} \xrightarrow{\alpha_i} (W, K \cup \{l\}) \triangleright \{\!|A_i[l/a]|\!\}$. The induction hypothesis yields that $(W, K \cup \{k\}) \triangleright \{\!|A_i[k/a]|\!\} \smile (W, K \cup \{l\}) \triangleright \{\!|A_i[l/a]|\!\}$.

A symmetric argument starting with the right hand side ends the proof.

∎

Lemma 6.7 (Renaming of normal forms) *For any two configurations* $(W, K \cup \{k\}) \triangleright \{\!|N[k/a]; \pi|\!\}$ *and* $(W, K \cup \{l\}) \triangleright \{\!|N[l/a]; \pi|\!\}$ *where* N *is on normal form and where* $k, l \notin W$ *and where* $k, l \notin CV[N]$,

$$(W, K \cup \{k\}) \triangleright \{\!|N[k/a]; \pi|\!\} \smile (W, K \cup \{l\}) \triangleright \{\!|N[l/a]; \pi|\!\}$$

Proof

We use induction on the depth n of N.

Base case $n = 0$: In this case N is ***nil***, and the property obviously holds.

Induction step $n > 0$: Then N has the form $\sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j)$. Assume that the left hand side moves. Then there are two cases to consider.

Case 1 Assume that $(W, K \cup \{k\}) \triangleright \{\!|(\sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j))[k/a]; \pi|\!\} \xrightarrow{\alpha_i} (W, K \cup \{k\}) \triangleright \{\!|N_i[k/a]; \pi|\!\}$. Then obviously $\alpha_i \neq k$ so α_i contains no a. Consequently $\alpha_i \in W$ and therefore $(W, K \cup \{l\}) \triangleright \{\!|(\sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j))[l/a]; \pi|\!\} \xrightarrow{\alpha_i} (W, K \cup \{l\}) \triangleright \{\!|N_i[l/a]; \pi|\!\}$. The induction hypothesis yields that $(W, K \cup \{k\}) \triangleright \{\!|N_i[k/a]; \pi|\!\} \smile (W, K \cup \{l\}) \triangleright \{\!|N_i[l/a]; \pi|\!\}$.

Case 2 Assume that $(W, K \cup \{k\}) \triangleright \{\!|(\sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j))[k/a]; \pi|\!\} \xrightarrow{\beta_j} (W, K \cup \{k\}) \triangleright \{\!|\boldsymbol{forked}(B_j[k/a]); \pi|\!\}$. Then obviously $\beta_j \neq k$ so β_j contains no a. Consequently $\beta_j \in W$ and therefore $(W, K \cup \{l\}) \triangleright \{\!|(\sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j))[l/a]; \pi|\!\} \xrightarrow{\beta_j} (W, K \cup \{l\}) \triangleright \{\!|\boldsymbol{forked}(B_j[l/a]); \pi|\!\}$. Finally, $(W, K \cup \{k\}) \triangleright \{\!|\boldsymbol{forked}(B_j[k/a]); \pi|\!\} \smile (W, K \cup \{l\}) \triangleright \{\!|\boldsymbol{forked}(B_j[l/a]); \pi|\!\}$. This follows from the fact that generally $\rho \triangleright \{\!|\boldsymbol{forked}(p); q|\!\} \smile \rho \triangleright \{\!|p, q|\!\}$, and then by applying the congruence lemma (5.9) to the fact that due to the previous lemma (6.6): $(W, K \cup \{k\}) \triangleright \{\!|B_j[k/a]|\!\} \smile (W, K \cup \{l\}) \triangleright \{\!|B_j[l/a]|\!\}$.

A symmetric argument starting with the right hand side ends the proof.

■

We can now prove the soundness of expansion law $\mathcal{E}_3$.

Proposition 6.8 ($\mathcal{E}_3$ is sound) *The expansion law $\mathcal{E}_3$ is sound with respect to strong process congruence.*

Proof

Let $\rho \stackrel{def}{=} Env[\{a\}(\sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j))] \cup Env[\sum_{\alpha_i \neq a} \alpha_i; \{a\}N_i + \sum_{\beta_j \neq a} \beta_j; \boldsymbol{forked}(\{a\}B_j)]$. We want to show that $\rho \triangleright \{\!|\{a\}(\sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j)); \pi|\!\} \smile \rho \triangleright \{\!|(\sum_{\alpha_i \neq a} \alpha_i; \{a\}N_i + \sum_{\beta_j \neq a} \beta_j; \boldsymbol{forked}(\{a\}B_j)); \pi|\!\}$. Assume that the left hand side moves. The first to happen is the allocation of a new channel k (we use the notation ρ^k for the environment $(W, K \cup \{k\})$ when $\rho = (W, K)$): $\rho \triangleright \{\!|\{a\}(\sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j)); \pi|\!\} \Longrightarrow \rho^k \triangleright \{\!|(\sum_i \alpha_i; N_i[k/a] + \sum_j \beta_j; \boldsymbol{forked}(B_j)[k/a]); \pi|\!\}$. Now there are two cases to consider.

Case 1 $\rho^k \triangleright \{\!|(\sum_i \alpha_i; N_i[k/a] + \sum_j \beta_j; \boldsymbol{forked}(B_j)[k/a]); \pi|\!\} \xrightarrow{\alpha_i} \rho^k \triangleright \{\!|N_i[k/a]; \pi|\!\}$ where $\alpha_i \neq k$. But we also have that $\rho \triangleright \{\!|(\sum_{\alpha_i \neq a} \alpha_i; \{a\}N_i + \sum_{\beta_j \neq a} \beta_j; \boldsymbol{forked}(\{a\}B_j)); \pi|\!\} \xrightarrow{\alpha_i} \rho \triangleright \{\!|\{a\}N_i; \pi|\!\}$. So we finish this case by showing that $\rho^k \triangleright \{\!|N_i[k/a]; \pi|\!\} \smile \rho \triangleright \{\!|\{a\}N_i; \pi|\!\}$. We do this by examining the two cases where respectively the left hand side and the right hand side makes a move.

Case 1.1 $\rho^k \triangleright \{\!|N_i[k/a]; \pi|\!\} \xrightarrow{\alpha} \rho^k \triangleright P$. But we also have that $\rho \triangleright \{\!|\{a\}N_i; \pi|\!\} \Longrightarrow \rho^k \triangleright \{\!|N_i[k/a]; \pi|\!\} \xrightarrow{\alpha} \rho^k \triangleright P$.

Case 1.2 $\rho \triangleright \{\!|\{a\}N_i; \pi|\!\} \Longrightarrow \rho^l \triangleright \{\!|N_i[l/a]; \pi|\!\} \xrightarrow{\alpha} C_2$. But due to lemma (6.7) we have that $\rho^k \triangleright \{\!|N_i[k/a]; \pi|\!\} \smile \rho^l \triangleright \{\!|N_i[l/a]; \pi|\!\}$, so consequently $\rho^k \triangleright \{\!|N_i[k/a]; \pi|\!\} \xrightarrow{\alpha} C_1$ where $C_1 \smile C_2$.

Case 2 $\rho^k \triangleright \{\!|(\sum_i \alpha_i; N_i[k/a] + \sum_j \beta_j; \boldsymbol{forked}(B_j)[k/a]); \pi|\!\} \xrightarrow{\beta_j} \rho^k \triangleright \{\!|\boldsymbol{forked}(B_j[k/a]); \pi|\!\}$ where $\beta_j \neq k$. But we also have that $\rho \triangleright \{\!|(\sum_{\alpha_i \neq a} \alpha_i; \{a\}N_i + \sum_{\beta_j \neq a} \beta_j; \boldsymbol{forked}(\{a\}B_j)); \pi|\!\} \xrightarrow{\beta_j} \rho \triangleright \{\!|\boldsymbol{forked}(\{a\}B_j); \pi|\!\}$. So we finish this case by showing that $\rho^k \triangleright \{\!|\boldsymbol{forked}(B_j[k/a]); \pi|\!\} \smile \rho \triangleright \{\!|\boldsymbol{forked}(\{a\}B_j); \pi|\!\}$. This follows from the fact that generally $\rho \triangleright \{\!|\boldsymbol{forked}(p); q|\!\} \smile \rho \triangleright \{\!|p, q|\!\}$, and then by applying the congruence lemma (5.9) to the fact that $\rho^k \triangleright \{\!|B_j[k/a]|\!\} \smile \rho \triangleright \{\!|\{a\}B_j|\!\}$. We prove the latter by examining the two cases where respectively the left hand side and the right hand side makes a move.

Case 1.1 $\rho^k \triangleright \{\!|B_j[k/a]|\!\} \xrightarrow{\alpha} \rho^k \triangleright P$. But we also have that $\rho \triangleright \{\!|\{a\}B_j|\!\} \Longrightarrow \rho^k \triangleright \{\!|B_j[k/a]|\!\} \xrightarrow{\alpha} \rho^k \triangleright P$.

Case 1.2 $\rho \triangleright \{\!|\{a\}B_j|\!\} \Longrightarrow \rho^l \triangleright \{\!|B_j[l/a]|\!\} \xrightarrow{\alpha} C_2$. But due to lemma (6.6) we have that $\rho^k \triangleright \{\!|B_j[k/a]|\!\} \smile \rho^l \triangleright \{\!|B_j[l/a]|\!\}$, so consequently $\rho^k \triangleright \{\!|B_j[k/a]|\!\} \xrightarrow{\alpha} C_1$ where $C_1 \smile C_2$.

A symmetric argument starting with $\rho \triangleright \{\!|(\sum_{\alpha_i \neq a} \alpha_i; \{a\}N_i + \sum_{\beta_j \neq a} \beta_j; \boldsymbol{forked}(\{a\}B_j)); \pi|\!\}$ ends the proof.

■

We also need to verify that the original expansion laws $\mathcal{E}_1$ and $\mathcal{E}_2$ are sound in the new setting:

Proposition 6.9 ($\mathcal{E}_1$ and $\mathcal{E}_2$ are sound) *The expansion laws $\mathcal{E}_1$ and $\mathcal{E}_2$ are sound with respect to strong process congruence.*

Proof

The two expansion laws do not involve applications of the channel allocating operators $(\_)_\_$ and $\{\_\}_\_$. Basically only normal forms (and manipulations on these) are involved. The original proofs of soundness – propositions (4.8) and (4.10) – therefore carry over unchanged, except for the addition of an environment that, however, never changes during evaluation.

■

The basic axioms $\mathcal{A}X$, the three expansion laws $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$ and the inference rules $\mathcal{I}$ constitute our proof system $\mathcal{A}S$, which we shall show to be (limited) complete. We write $\vdash p = q$ if the equivalence of p and q can be entailed within the proof system $\mathcal{A}S$.

6.1.4 Soundness and Limited Completeness

$\mathcal{A}S$ is Sound

Proposition 6.10 ($\mathcal{A}S$ is sound) *For all p, q in* FC_r*,*

$$\textit{Whenever} \vdash p = q \textit{ then } p \equiv q.$$

Proof

Soundness of $\mathcal{A}S$ reduces to soundness of the basic axioms $\mathcal{A}X$, soundness of the expansion laws $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$ and soundness of the inference rules $\mathcal{I}$. This was proved in propositions 6.2, 6.8, 6.9 and 6.4.

■

Existence of Normal Forms

The lemmas 4.12, 4.13, 4.14 and 4.15 all hold in the new setting. The proofs are exactly as before. Recall that the two first lemmas state how occurrences of the ***forked*** operator can be eliminated and the next two lemmas state that there exist normal forms for subcategories of process terms. In addition we need two more lemmas concerning restriction. These six lemmas lead to the final proposition stating that there exists a normal form for every process term in FC_r. The two additional lemmas for restriction are as follows.

Lemma 6.11 (Restriction on simple normal forms) *For any simple normal form A and for any channel name a, there exists a simple normal form B such that:*

$$\vdash \{a\}A = B$$

Proof

We use induction on the depth n of A.

Base case $n = 0$: In this case A is ***nil***, and the expansion law $\mathcal{E}_3$ yields that $\vdash \{a\}\boldsymbol{nil} = \boldsymbol{nil}$.

Induction step $n > 0$: Then A has the form $\sum_i \alpha_i; A_i$, and we can deduce the following:

$\{a\} \sum_i \alpha_i; A_i$
$=$
$\sum_{\alpha_i \neq a} \alpha_i; \{a\}A_i$ – *expansion law $\mathcal{E}_3$*
$=$
$\sum_{\alpha_i \neq a} \alpha_i; B_i$ – *induction hypothesis: $\{a\}A_i$ equates simple normal form B_i*

■

Lemma 6.12 (Restriction on normal forms) *For any normal form M and for any channel name a, there exists a normal form N such that:*

$$\vdash \{a\}M = N$$

Proof

We use induction on the depth n of M.

Base case $n = 0$: In this case M is ***nil***, and the expansion law $\mathcal{E}_3$ yields that $\vdash \{a\}\,\boldsymbol{nil} = \boldsymbol{nil}$.

Induction step $n > 0$: Then M has the form $\sum_i \alpha_i; M_i + \sum_j \beta_j; \boldsymbol{forked}(B_j)$, and we can deduce the following:

$$\{a\}(\textstyle\sum_i \alpha_i; M_i + \sum_j \beta_j; \boldsymbol{forked}(B_j))$$
$$=$$
$$\textstyle\sum_{\alpha_i \neq a} \alpha_i; \{a\}M_i + \sum_{\beta_j \neq a} \beta_j; \boldsymbol{forked}(\{a\}B_j) \quad \text{– \textit{expansion law} } \mathcal{E}_3$$
$$=$$
$$\textstyle\sum_{\alpha_i \neq a} \alpha_i; N_i + \sum_{\beta_j \neq a} \beta_j; \boldsymbol{forked}(C_j)$$

– *induction hypothesis applied to $\{a\}M_i$ yielding normal form N_i,*
– *and lemma (6.11) applied to $\{a\}B_j$ yielding simple normal form C_j*

■

We can finally state the following proposition. Let $\mathcal{L}^-$ represent the subset of $\mathcal{L}$ which contains no occurrences of ***forked***.

Proposition 6.13 (Normal forms for FC$_r$) *For any p in $\mathcal{L}^-$, there exists a normal form N such that:*

$$\vdash p = N$$

Proof

The proof is as the proof of proposition 4.16 using structural induction. Assume that any subterm has a normal form. That is, if p and p_i (for any index i) are subterms, then $\vdash p = M$ and $\vdash p_i = M_i$. Then we proceed by case analysis. The cases $p; q$ and $\boldsymbol{fork}(p)$ are as in the proof of proposition 4.16. The remaining cases are treated below.

Case $\sum_i \alpha_i; p_i$:

$$\textstyle\sum_i \alpha_i; p_i$$
$$=$$
$$\textstyle\sum_i \alpha_i; M_i$$

– *induction hypothesis*

Case $(a)p$:

$$
\begin{array}{l}
(a)p \\
= \\
(a)M \\
\text{– } \textit{induction hypothesis} \\
= \\
\tau; \{a\}M \\
\text{– } \textit{axiom (6.11)} \\
= \\
\tau; N \\
\text{– } \textit{lemma 6.12}
\end{array}
$$

Case $\{a\}p$:

$$
\begin{array}{l}
\{a\}p \\
= \\
\{a\}M \\
\text{– } \textit{induction hypothesis} \\
= \\
N \\
\text{– } \textit{lemma 6.12}
\end{array}
$$

■

Limited Completeness

We are now able to formally present our completeness result. Lemma (4.17) and proposition (4.18) stating respectively completeness for a subcategory of processes and completeness for normal forms also hold in the new setting. The lemma and proposition do not involve applications of the channel allocating operators $(\_)\_$ and $\{\_\}\_$. Basically only normal forms (and manipulations on these) are involved. The original proofs of these properties therefore carry over nearly unchanged, except for the addition of an environment that, however, never changes during evaluation.

The completeness result we arrive at is limited in the sense that it only holds for process terms of $\mathcal{L}^-$ (and not $\mathcal{L}$).

Theorem 6.14 ($\mathcal{AS}$ is limited Complete) *For all p, q in $\mathcal{L}^-$,*

$$\textit{Whenever } p \equiv q \textit{ then } \vdash p = q$$

Proof

As the proof of theorem 4.19.

■

6.2 A Refinement Logic

One goal for work within program specification is to provide a theory for the formal refinement of specifications into programs via sequences of verified-correct development steps. In this section we shall pursue this goal by focusing on specification and stepwise refinement into programs in the Fork Calculus. That is, we define a refinement logic for FC_r which includes programming constructs as well as specification constructs, so programs are special cases of formulae. The programming constructs are those of FC_r, while the specification constructs are those of Hennessy-Milner logic [HM85] with recursion [Koz82, Lar90, GLZ, Lar87]. We shall for example allow a formula like $\boldsymbol{fork}(<a!>\, \boldsymbol{tt}); \boldsymbol{fork}(<a?>\, \boldsymbol{tt})$.

In order to motivate this work, we shall make some considerations on problem solving in general, without necessarily referring to the special kind of problem solving dealing with refinement of specifications into programs.

Suppose we are given a formulation P of some *problem*, and suppose we are to find a *solution* to this problem. We assume that there is a way of proving whether a candidate solution is a solution. We shall refer to our task as *problem solving*, and there seems to be two ways of approaching this:

Solution matching One way is just to *search* for some candidate solution p and then when one is found: to prove that it really is a solution.

Problem decomposition Another way is to *decompose* the problem into a number of *subproblems* $P_1, \ldots, P_n$ together with a *composition*-function f, such that given solutions $x_1, \ldots, x_n$ of these subproblems (x_i is a solution to subproblem P_i for $i \in \{1, \ldots, n\}$), then $f(x_1, \ldots, x_n)$ is a solution to P. The decomposition leaves two activities to be done:

composition proof: where it is proved that whichever solutions $x_1, \ldots, x_n$ are found for the subproblems $P_1, \ldots, P_n$, then $f(x_1, \ldots, x_n)$ is a solution to P.

subproblem solving where the subproblems $P_1, \ldots, P_n$ are solved yielding sub solutions $p_1, \ldots, p_n$. Problem solving is recursive in nature.

Let us return to the grounds of specifications and programs. A problem is in this context a specification in some logic, and a solution is a program in some programming language. There must be a formal definition of a satisfaction relation $\models$ between programs and specifications, and a program p is a solution to the problem P if $p \models P$. So what about problem solving? If we are satisfied with an approach based on *solution matching* only, then what we need (besides the programming language) is "just" a logic and a satisfaction relation. Examples of this is Hennessy-Milner logic for CCS, and our logic for FC as presented in section 4.2.

If we on the other hand prefer to approach problem solving by means of problem decomposition, then we need a way of formulating that a specification P is decomposed into subspecifications $P_1, \ldots, P_n$ together with a composition function f. One nice formulation is as an implication: $f(P_1, \ldots, P_n) \Rightarrow P$, observing that f will naturally be a composition of operators in the programming language. This leads us to conclude that $f(P_1, \ldots, P_n)$ can be a formula in the logic and that the operators of the programming language therefore must be among the operators of the logic. A *refinement logic* for FC_r is consequently an extension of FC_r with Hennessy-Milner Logic operators. As a further consequence, FC_r programs are special cases of formulae. Note that in this setting, the verification of the assertion $p \models P$ is converted to a proof of the validity of the formula $p \Rightarrow P$.

Hence, the refinement logic provides a tool of decomposing problems into smaller problems; an approach which sounds, generally speaking, to be an advantage. We shall, however, not judge whether such working method is useful in practice. There is always an overhead in writing abstract specifications before doing programming, and with the refinement approach, one has to do this repeatedly.

The logic has though some advantages, even if we don't believe in refinement via problem decomposition. First of all, a prerequisite for the refinement via decomposition to work is that the logic is compositional. That is, suppose that $p_1 \models \psi_1, \ldots, p_n \models \psi_n$, then $Op(p_1, \ldots, p_n) \models Op(\psi_1, \ldots, \psi_n)$. When decomposing a problem we go from right (of $\models$) to left. This compositionality property can, however, also be applied in the opposite direction (left to right): to infer a property about a program from properties about its subprograms. That is, suppose we have proved that $p_1 \models \psi_1, \ldots, p_n \models \psi_n$. A natural question is: what is the strongest property one can deduce about $Op(p_1, \ldots, p_n)$, which depends solely on $\psi_1, \ldots, \psi_n$ and Op? With the logic presented it is $Op(\psi_1, \ldots, \psi_n)$. Such a compositional proof style may be useful, in the cases where proofs have to be carried out partly "by hand" because automatisation is impossible. Second, the logic allows to write very concise specifications, compared to what is possible in pure modal logic. That is, often we know quite precisely how a system component has to work, and it seems easy just to program it. Try for example to formulate $\textit{\textbf{fix}}\, x \cdot accept?; deliver!; x$ in your favorite modal or temporal logic.

The mix of programming and formula syntax to obtain compositional proof systems has been applied to CCS-like calculi in [Win86, GS86, GS87, Hol89]. Other attempts to obtain compositional proof systems, using different techniques, have been performed for variants of CCS. The approach in [Sti85c, Sti85b, Sti85a] uses relativised satisfaction relations. For example $p \models_B A$ means that for any program q, if $q \models B$ then $p|q \models A$. So here the environment is represented by the formula B. In [Win84] this idea is developed into a proof system based on assumptions about the environment. The statement $p \models_B A$ from before is now expressed as $x : B \models p|x : A$. Winskel announces in [Win84] the forthcoming paper [Win86] cited above, where the approach of mixing programming and formulae is presented. Note that in the approach we use where syntax of programs and formulae is mixed, this statement (using CCS notation) becomes: $p|B \Longrightarrow A$, meaning that $p|B$ refines/implements A. We shall detail this below. The specification-program mixture is sometimes referred to under the heading "programs are predicates" ([Hoa85b, Abr85]). An early attempt to obtain a compositional proof system for temporal logic is given in [BKP84]. This approach is based on distinguishing between actions performed by the process under observation and actions performed by the environment.

As we have mentioned, FC was originally designed as a projection of CML. A refinement logic can likewise be regarded as contributing to a refinement logic for CML, in the style of EML (Extended ML) [ST90].

Section 6.2.1 defines the syntax and informal semantics of the logic. Section 6.2.2 defines its formal semantics. Section 6.2.3 proves a set of properties of the logic. Section 6.2.4 defines derived forms of formulae that are useful when writing specifications in praxis. Section 6.2.5 sets up a collection of proof rules, which allow us to deduce statements within the logic. These proof rules are applied to a non-trivial example in section 6.2.6, where a protocol specification is refined into a program. Section 6.2.7 concludes with a further discussion of related work.

6.2.1 Syntax and Informal Semantics

The syntax of the logic is as follows:

$$
\begin{aligned}
\psi &::= \sum_i \alpha_i;\psi_i \mid \psi_1;\psi_2 \mid \textit{\textbf{fork}}(\psi) \mid \textit{\textbf{forked}}(\psi) \mid (a)\psi \mid \{a\}\psi \mid \\
&\qquad \textit{\textbf{tt}} \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid <\!\alpha\!>\psi \mid \nu\, x \cdot \psi \mid x \\
\alpha &::= \tau \mid a? \mid a!
\end{aligned}
$$

For recursive formulae $\nu\ x \cdot \psi$ we shall assume that any free occurrence of x within ψ is under the scope of an even number of negations (this will ensure monotonicity of ψ). We let Ψ_o stand for all the formulae generated by the above syntax, including formulae with free variables. The set of closed formulae is denoted by Ψ.

Given a formula ψ in this logic and given a FC_r program p, we shall formally define what it means for the program to satisfy the formula, which we write as $p \models \psi$. Let us first give a mostly informal description.

The first six alternatives defining ψ are just the (non-recursive) operators of FC_r. Suppose Op is one of these operators (actions α are part of the choice operator) and suppose $\psi_1, \ldots, \psi_n$ are formulae, then $p \models Op(\psi_1, \ldots, \psi_n)$ if $p \equiv Op(q_1, \ldots, q_n)$ for some processes $q_1, \ldots, q_n$ such that $q_i \models \psi_i$ for $i \in \{1, \ldots, n\}$.

The remaining alternatives are the logic constructs of Hennessy-Milner-Logic with recursion. These include the truth $\boldsymbol{tt}$ which any process satisfies, and conjunction and negation with the obvious meanings. The formula $<\alpha>\psi$ is satisfied by any process that can perform an α-action (as well as possibly other actions) and then become a process that satisfies ψ. Finally, the maximal fixpoint $\nu\ x \cdot \psi$ provides the basic mechanism for recursion. An understanding of this operator can be obtained by noting that a maximal fixpoint in restricted cases has the same meaning as an infinite conjunction. Let $\psi[A]$ mean $\psi[A/x]$ and let $\psi^n[A]$ mean n applications $\psi[\psi[\ldots\psi[A]]]$, with $\psi^0[A] = A$. Then $\nu\ x \cdot \psi$ means: $\bigwedge_{n \in \mathbf{N}} \psi^n[\boldsymbol{tt}]$.

As an example, consider the formula $\nu\ X \cdot <\tau> X$. A process satisfies this formula if it can perform an infinite sequence of τ-actions. To see this, note that this maximal fixpoint is equivalent to the following infinite conjunction: $\boldsymbol{tt} \wedge <\tau>\boldsymbol{tt} \wedge <\tau><\tau>\boldsymbol{tt} \wedge \ldots$. For example, the process $\boldsymbol{fix}\, x \cdot \tau; x + a!; \boldsymbol{nil}$ satisfies the formula.

Note that there is no need for a special $\boldsymbol{fix}\, x \cdot \psi$ construct for writing recursive programs. The $\nu\ x \cdot \psi$ serves this purpose as well. To see this, observe that $\nu\ x \cdot a!; x$ stands for the infinite conjunction: $\boldsymbol{tt} \wedge a!; \boldsymbol{tt} \wedge a!; a!; \boldsymbol{tt} \wedge \ldots$. The general relationship between constructs in FC_r and the corresponding constructs in the logic is formalised later in theorem 6.23.

6.2.2 Formal Semantics

Technically speaking, the semantics maps a formula into a set of processes, where each process in this set is said to satisfy the formula. Since the maximal fixpoint construct $\nu\ x \cdot \psi$ introduces a bound name x, this semantic mapping will further depend on an environment. An environment is a function $\rho : Env = X \rightarrow \mathcal{P}(\mathcal{L})$ from variables to sets of processes. For an environment ρ, variable x and a process set $S \subseteq \mathcal{L}$ we use $\rho[S/x]$ to mean ρ updated to have the value S at x i.e.

$$\rho[S/x](y) = \begin{cases} \rho(y) & \text{if } y \neq x \\ S & \text{if } y = x \end{cases}$$

The denotation of a formula ψ is a function $[\![\psi]\!] : Env \rightarrow \mathcal{P}(\mathcal{L})$. It is defined as follows.

Definition 6.15 (The denotation $[\![.]\!]$ of formulae) *The semantics $[\![.]\!] : \Psi_o \rightarrow (Env \rightarrow \mathcal{P}(\mathcal{L}))$ of formulae is defined as follows. Let operators Op range over the set: $\{\sum_i \alpha_i;\_\,,\ \_;\_\,,\ \boldsymbol{fork}(\_)\,,\ \boldsymbol{forked}(\_)\,,\ (a)\_\,,\ \{a\}\_\}$ with arity n.*

$$\begin{array}{rcl}
[\![\boldsymbol{tt}]\!]\rho & = & \mathcal{L} \\
[\![\psi_1 \wedge \psi_2]\!]\rho & = & [\![\psi_1]\!]\rho \cap [\![\psi_2]\!]\rho \\
[\![\neg\psi]\!]\rho & = & \mathcal{L} - [\![\psi]\!]\rho \\
[\![<\!\alpha\!>\psi]\!]\rho & = & \{p \in \mathcal{L} \mid \exists C \in Con \cdot Config[p] \xrightarrow{\alpha} C \wedge C \models_\rho \psi\} \\
[\![\nu\, x \cdot \psi]\!]\rho & = & \bigcup\{S \subseteq \mathcal{L} \mid S \subseteq [\![\psi]\!]\rho[S/x]\} \\
[\![x]\!]\rho & = & \rho(x) \\
[\![Op(\psi_1, \ldots, \psi_n)]\!]\rho & = & \{p \in \mathcal{L} \mid \exists q_1, \ldots, q_n \in \mathcal{L} \cdot \\
 & & \quad q_i \in [\![\psi_i]\!]\rho \wedge p \equiv Op(q_1, \ldots, q_n)\}
\end{array}$$

where for any configuration C in Con and for any formula ψ in Ψ_o:

$$C \models_\rho \psi \quad \stackrel{def}{=} \quad \exists q \in \mathcal{L} \cdot q \in [\![\psi]\!]\rho \wedge C \smile Config[q]$$

A special case of $[\![Op(\psi_1, \ldots, \psi_n)]\!]\rho$ is when Op is the empty choice $\boldsymbol{nil}$ (so $n = 0$) in which case the denotation is the set of processes p where $p \equiv \boldsymbol{nil}$.

As a convention, when ψ is closed, we shall let $p \models \psi$ mean that $p \in [\![\psi]\!]\rho$ for any ρ. Likewise, when ψ is closed, we shall let $C \models \psi$ mean that $C \models_\rho \psi$ for any ρ.

A comment on the semantics of the maximal fixpoint construct is necessary. We recapture the following theorem from [Tar55]:

Theorem 6.16 (Tarski's theorem) *Let E be a set. Let $\varphi : \mathcal{P}(E) \rightarrow \mathcal{P}(E)$ be a monotonic function i.e. for all $S_1, S_2 \in \mathcal{P}(E)$,*

$$S_1 \subseteq S_2 \Rightarrow \varphi(S_1) \subseteq \varphi(S_2)$$

Then φ has a maximal fixpoint $\nu\, S \cdot \varphi(S)$ and a minimal fixpoint $\mu\, S \cdot \varphi(S)$ given by:

$$\begin{array}{rcl}
\nu\, S \cdot \varphi(S) & \stackrel{def}{=} & \bigcup\{S' \subseteq E \mid S' \subseteq \varphi(S')\} \\
\mu\, S \cdot \varphi(S) & \stackrel{def}{=} & \bigcap\{S' \subseteq E \mid \varphi(S') \subseteq S'\}
\end{array}$$

The denotation $[\![\nu\, x \cdot \psi]\!]\rho$ is consequently the maximal fixpoint of the function:

$$F = \lambda S \cdot [\![\psi]\!]\rho[S/x]$$

provided that $[\![\psi]\!]\rho[S/x]$ is monotone in S, which is easily proven. As we shall see below, the minimal fixpoint can be obtained as a derived form from negation and the maximal fixpoint construct.

It is natural to define a refinement (implementation) relation $\Longrightarrow \subseteq \Psi \times \Psi$ between formulae in the logic. We write $\psi_1 \Longrightarrow \psi_2$ instead of $(\psi_1, \psi_2) \in \Longrightarrow$ and we read this as "the specification ψ_1 refines (implements) ψ_2. The exact meaning of this is formulated as follows. Let $\mathcal{L}(\psi)$ stand for the programs satisfying ψ. That is: $\mathcal{L}(\psi) \stackrel{def}{=} \{p \in \mathcal{L} \mid p \models \psi\}$. Then $\psi_1 \Longrightarrow \psi_2$ iff. $\mathcal{L}(\psi_1) \subseteq \mathcal{L}(\psi_2)$. We shall below provide a proof system for proving statements involving $\Longrightarrow$.

We give two examples where it is proved that some program satisfies some formula.

Example 6.17 *We shall prove that:*

$$\boldsymbol{fork}(\alpha); \beta \models \tau; (<\beta> \boldsymbol{forked}(\alpha))$$

This holds if we can find processes p_1 and p_2 such that:

$$\begin{array}{ll} (1) & p_1 \models \tau \\ (2) & p_2 \models <\beta> \boldsymbol{forked}(\alpha) \\ (3) & \boldsymbol{fork}(\alpha); \beta \equiv p_1; p_2 \end{array}$$

If $p_1 = \tau$ and $p_2 = \boldsymbol{forked}(\alpha); \beta$, conditions (1), (2) and (3) are all satisfied as proved in the following. Thus, the program satisfies the specification.

(1) $\tau \models \tau$: *This is straightforward.*

(2) $\boldsymbol{forked}(\alpha); \beta \models <\beta> \boldsymbol{forked}(\alpha)$: *This holds if $\{\!|(\boldsymbol{forked}(\alpha); \beta); \pi|\!\} \xrightarrow{\beta} P'$ for some program P' such that $P' \models \boldsymbol{forked}(\alpha)$. We ignore the environment (W, K) since no channel allocation takes place.*

Now, from the semantics we can obtain P' to be $\{\!|\boldsymbol{nil}; \pi, \alpha|\!\}$. It remains to show that $P' \models \boldsymbol{forked}(\alpha)$, that is:

$$\{\!|\boldsymbol{nil}; \pi, \alpha|\!\} \models \boldsymbol{forked}(\alpha)$$

This holds if:

$$\exists q \in \mathcal{L} \cdot q \models \boldsymbol{forked}(\alpha) \ \wedge \ \{\!|\boldsymbol{nil}; \pi, \alpha|\!\} \smile \{\!|q; \pi|\!\}$$

This holds since choosing $q = \boldsymbol{forked}(\alpha)$ *makes the following hold (it is easy to show that* $\boldsymbol{forked}(\alpha) \models \boldsymbol{forked}(\alpha)$*):*

$$\{\!|\boldsymbol{nil}; \pi, \alpha|\!\} \smile \{\!|\boldsymbol{forked}(\alpha); \pi|\!\}$$

To see that this holds we can find a bisimulation containing the pair $(\{\!|\boldsymbol{nil}; \pi, \alpha|\!\}, \{\!|\boldsymbol{forked}(\alpha); \pi|\!\})$.

(3) $\boldsymbol{fork}(\alpha); \beta \equiv \tau; (\boldsymbol{forked}(\alpha); \beta)$**:** *This can be seen easily by applying the axioms for strong congruence.*

Example 6.18 *Let* $p \stackrel{def}{=} \boldsymbol{fix}\, x \cdot \alpha; x$ *and let* $\psi \stackrel{def}{=} \nu X \cdot \langle\alpha\rangle\, \boldsymbol{tt} \wedge [\alpha]X$. *We shall prove that:*

$$\boldsymbol{fork}(p) \models \tau; \boldsymbol{forked}(\psi)$$

Note first, that due to the axiomatisation we can deduce that $\boldsymbol{fork}(p) \equiv \tau; \boldsymbol{forked}(p)$. *It is hereafter straight forward to see that the program satisfies the specification if* $p \models \psi$, *which is equivalent to* $p \in [\![\psi]\!]\rho_{\{\}}$ *where* $\rho_{\{\}}$ *is the empty environment. To prove this, note that*

$$[\![\psi]\!]\rho_{\{\}} = \bigcup\{S \subseteq \mathcal{L} \mid S \subseteq [\![\langle\alpha\rangle\, \boldsymbol{tt} \wedge [\alpha]X]\!]\rho_{\{\}}[S/X]\}$$

Let $\rho_s \stackrel{def}{=} \rho_{\{\}}[S/X]$. *It is then sufficient to find some subset* $S \subseteq \mathcal{L}$ *such that* $p \in S$ *and*

$$S \subseteq [\![\langle\alpha\rangle\, \boldsymbol{tt} \wedge [\alpha]X]\!]\rho_s$$

Let S *be the set* $\{p, \boldsymbol{nil}; p\}$. *Due to the semantics of* $\wedge$ *we shall show that*

$$\begin{array}{l} S \subseteq [\![\langle\alpha\rangle\, \boldsymbol{tt}]\!]\rho_s \quad \textit{and} \\ S \subseteq [\![[\alpha]X]\!]\rho_s \end{array}$$

The set $[\![\langle\alpha\rangle\, \boldsymbol{tt}]\!]\rho_s$ *is the set of processes that can perform an* α *action and clearly* S *is a subset. The set* $[\![[\alpha]X]\!]\rho_s$ *is the set of processes, that if they perform an* α *action, then the resulting process will be in* S. *It is easy to check that this holds for both processes in* S*: both* p *and* $\boldsymbol{nil}; p$ *go into* $\boldsymbol{nil}; p$, *when performing an* α *action.*

6.2.3 Properties of the Logic

In this section we state some properties of the satisfaction relation $\models$ which may increase our trust in its definition. Note first, that our logic Ψ_o (regarded as a set of formulae) contains the programming language $\mathcal{L}_o$ as a subset. That is, we define a mapping $\varphi[\_] : \mathcal{L}_o \rightarrow \Psi_o$ as follows:

$$\begin{aligned} \varphi[\boldsymbol{fix}\, x \cdot p] &= \nu\, x \cdot \varphi[p] \\ \varphi[x] &= x \\ \varphi[Op(p_1, \ldots, p_n)] &= Op(\varphi[p_1], \ldots, \varphi[p_n]) \end{aligned}$$

The first theorem we state says that for any process q, the denotation of the corresponding formula, $\varphi[q]$, is the set of processes that are equivalent to q. That is: for any two processes p and q:

$$p \models \varphi[q] \Leftrightarrow p \equiv q$$

The importance of the lemma is due to the relationship it expresses between maximal fixpoints in the logic and programming fixpoints. In order to prove this, a number of definitions and lemmas are needed. They follow below and in appendix C.

We shall define a relation $\overset{\circ}{\smile}$ between open configurations, that is: configurations containing free variables. Assume that C_1 and C_2 are open configurations, then $C_1 \overset{\circ}{\smile} C_2$ iff. for all substitutions $\sigma : X \rightarrow \mathcal{L}$, it holds that $C_1[\sigma] \smile C_2[\sigma]$. The following definition is essential.

Definition 6.19
For any environment $\rho \in Env$, and for any two configurations C_1, C_2, where C_1 is closed and where C_2 is open,
$C_1 \in_\rho C_2$ iff. there exist processes $p \in \mathcal{L}$ and $q \in \mathcal{L}_o$ such that $p \in [\![\varphi[q]]\!]\rho$ and such that:

$$\begin{array}{lcl} C_1 \smile \mathit{config}[p] & \mathit{and} & \mathit{config}[q] \overset{\circ}{\smile} C_2 \\ & \mathit{or} & \\ C_1 \smile \mathit{config}[p;\pi] & \mathit{and} & \mathit{config}[q;\pi] \overset{\circ}{\smile} C_2 \end{array}$$

Whenever C_2 is closed, we shall write $C_1 \in C_2$ instead of $C_1 \in_\rho C_2$, no matter what ρ is, since it is of no importance.

We shall need the following lemma.

Lemma 6.20 *For any two closed configurations $C_1, C_2 \in Con$,*

$$\begin{array}{llllll} \mathit{1.)} & \mathit{Whenever} & C_1 \smile C_2 & \mathit{and} & C_2 \in C_3 & \mathit{then} \quad C_1 \in C_3 \\ \mathit{2.)} & \mathit{Whenever} & C_1 \in C_2 & \mathit{and} & C_2 \smile C_3 & \mathit{then} \quad C_1 \in C_3 \end{array}$$

Proof

To prove 1.) assume that $C_1 \smile C_2$ and $C_2 \in C_3$. Then $C_2 \smile \mathit{config}[p]$ and $C_3 \smile \mathit{config}[q]$ (or $C_2 \smile \mathit{config}[p;\pi]$ and $C_3 \smile \mathit{config}[q;\pi]$) for some $p, q \in \mathcal{L}$ such that $p \in [\![\varphi[q]]\!]$. But then $C_1 \smile \mathit{config}[p]$ (or $C_1 \smile \mathit{config}[p;\pi]$) and so $C_1 \in C_3$.

The proof of 2.) is similar.

■

The following lemma is the core lemma, that will be used to prove our main theorem.

Lemma 6.21 *For any processes $p \in \mathcal{L}$ and $q \in \mathcal{L}_g$, and for any environment $\rho \in Env$, such that $p \in [\![\varphi[q]]\!]\rho$ it holds that for all $\alpha \in Act$,*

1. *Whenever $\mathit{Config}[p] \xrightarrow{\alpha} C_1$ for some C_1 then $\mathit{Config}[q] \xrightarrow{\alpha} C_2$ for some C_2 and $C_1 \in_\rho C_2$*
2. *Whenever $\mathit{Config}[q] \xrightarrow{\alpha} C_2$ for some C_2 then $\mathit{Config}[p] \xrightarrow{\alpha} C_1$ for some C_1 and $C_1 \in_\rho C_2$*

Proof

See appendix C lemma C.7.

■

Lemma 6.22 *For any two (closed) configurations $C_1, C_2 \in Con$,*
if $C_1 \in C_2$ then,

1. *Whenever $C_1 \xrightarrow{\alpha} C_1'$ for some C_1' then $C_2 \xrightarrow{\alpha} C_2'$ for some C_2' and $C_1' \in C_2'$*
2. *Whenever $C_2 \xrightarrow{\alpha} C_2'$ for some C_2' then $C_1 \xrightarrow{\alpha} C_1'$ for some C_1' and $C_1' \in C_2'$*

Proof

We consider case (1), case (2) is similar. Assume that $C_1 \in C_2$. Now there are two cases depending on whether C_1 and C_2 both contain π or not.

Both C_1 and C_2 contain π. In this case $C_1 \smile \mathit{config}[p;\pi] = \mathit{Config}[p]$ and $C_2 \smile \mathit{config}[q;\pi] = \mathit{Config}[q]$ for some processes $p, q \in \mathcal{L}$ such that $p \in [\![\varphi[q]]\!]$. Assume that $C_1 \xrightarrow{\alpha} C_1'$. Then due to the above also $\mathit{Config}[p] \xrightarrow{\alpha}$

C_1'' where $C_1' \smile C_1''$. But then since $p \in [\![\varphi[q]]\!]$ lemma (6.21) yields that also $\mathit{Config}[q] \xrightarrow{\alpha} C_2''$ where $C_1'' \in C_2''$. Now since $C_2 \smile \mathit{Config}[q]$ it follows that $C_2 \xrightarrow{\alpha} C_2'$ where $C_2' \smile C_2''$. Since we have shown that $C_1' \smile C_1''$ and $C_1'' \in C_2''$ and $C_2'' \smile C_2'$, we conclude by lemma (6.20) that $C_1' \in C_2'$

Nor C_1 nor C_2 contain π. Let $\rho_p \stackrel{def}{=} \mathit{Env}[p]$ and $\rho_q \stackrel{def}{=} \mathit{Env}[q]$. In this case $C_1 \smile \mathit{config}[p] = \rho_p \triangleright \{|p|\}$ and $C_2 \smile \mathit{config}[q] = \rho_q \triangleright \{|q|\}$ for some processes $p, q \in \mathcal{L}$ such that $p \in [\![\varphi[q]]\!]$. Assume now that $C_1 \xrightarrow{\alpha} C_1'$. Then due to the above also $\rho_p \triangleright \{|p|\} \xrightarrow{\alpha} \rho_p' \triangleright \{|p'|\} \cup P$ where $C_1' \smile \rho_p' \triangleright \{|p'|\} \cup P$. From this latter α-transition we can easily conclude that $\mathit{Config}[p] = \rho_p \triangleright \{|p; \pi|\} \xrightarrow{\alpha} \rho_p' \triangleright \{|p'; \pi|\} \cup P$. But then since $p \in [\![\varphi[q]]\!]$ lemma (6.21) yields that also $\mathit{Config}[q] = \rho_q \triangleright \{|q; \pi|\} \xrightarrow{\alpha} \rho_q' \triangleright \{|q'; \pi|\} \cup Q$ where $\rho_p' \triangleright \{|p'; \pi|\} \cup P \in \rho_q' \triangleright \{|q'; \pi|\} \cup Q$. Now, from the fact that $\rho_q \triangleright \{|q; \pi|\} \xrightarrow{\alpha} \rho_q' \triangleright \{|q'; \pi|\} \cup Q$ we can easily deduce that $\mathit{config}[q] = \rho_q \triangleright \{|q|\} \xrightarrow{\alpha} \rho_q' \triangleright \{|q'|\} \cup Q$. Since further $C_2 \smile \mathit{config}[q]$ we get that $C_2 \xrightarrow{\alpha} C_2'$ where $C_2' \smile \rho_q' \triangleright \{|q'|\} \cup Q$. Now recall that $\rho_p' \triangleright \{|p'; \pi|\} \cup P \in \rho_q' \triangleright \{|q'; \pi|\} \cup Q$. That is: $\rho_p' \triangleright \{|p'; \pi|\} \cup P \smile \mathit{config}[p''; \pi]$ and $\rho_q' \triangleright \{|q'; \pi|\} \cup Q \smile \mathit{config}[q''; \pi]$ such that $p'' \in [\![\varphi[q'']]\!]$. But then we can easily deduce that $\rho_p' \triangleright \{|p'|\} \cup P \smile \mathit{config}[p'']$ and that $\rho_q' \triangleright \{|q'|\} \cup Q \smile \mathit{config}[q'']$. So we have that $C_1' \smile \mathit{config}[p'']$ and $C_2' \smile \mathit{config}[q'']$ where (as just stated) $p'' \in [\![\varphi[q'']]\!]$.

■

We are now able to prove the first main property.

Theorem 6.23 (Equivalence and satisfaction) *For any processes $p, q \in \mathcal{L}$,*

$$p \models \varphi[q] \quad \textit{iff} \quad p \equiv q$$

Proof

Our proof is divided into two cases corresponding to the two directions. Let Op range over $\{\sum_i \alpha_i;\_ ,\ \_;\_ ,\ \textbf{\textit{fork}}(\_)\ ,\ \textbf{\textit{forked}}(\_)\ ,\ (\_)\_ ,\ \{\_\}\_\}$.

A) $p \equiv q \Rightarrow p \models \varphi[q]$. Another way of formulating this property is as follows. Let $|q|$ denote the equivalence class of q. That is $|q| \stackrel{def}{=} \{p \in \mathcal{L} \mid p \equiv q\}$. Then this case can be stated as $|q| \subseteq [\![\varphi[q]]\!]$. In order to prove this we prove the following more general lemma. Assume that the process $q \in \mathcal{L}_w$ contains at most the free variables $\{x_1, \ldots, x_n\}$. Assume further that the processes $r_1, \ldots, r_n \in \mathcal{L}$ are closed (no free variables). Then:

Lemma 6.24

$$|q[r_1/x_1,\ldots,r_n/x_n]| \subseteq [\![\varphi[q]]\!]_{\{|r_1|/x_1,\ldots,|r_n|/x_n\}}$$

Obviously a special case of this lemma is that $|q| \subseteq [\![\varphi[q]]\!]$ for any closed process q. To prove the general lemma we use induction on the structure of q. We shall use the notation $\overline{r}/\overline{x}$ for the substitution $r_1/x_1,\ldots,r_n/x_n$, and we use $|\overline{r}|\,/\overline{x}$ for $|r_1|\,/x_1,\ldots,|r_n|\,/x_n$.

Case 1: $q = x$. Since q contains at most the free variables $\{x_1,\ldots,x_n\}$, $x = x_i$ for some i, so we have that: $|q[\overline{r}/\overline{x}]| = |x_i[\overline{r}/\overline{x}]| = |r_i| = [\![x_i]\!]_{\{|\overline{r}|/\overline{x}\}} = [\![\varphi[q]]\!]_{\{|\overline{r}|/\overline{x}\}}$.

Case 2: $q = Op(q_1,\ldots,q_m)$. We can perform the following deductions.

$$\begin{array}{l}
|Op(q_1,\ldots,q_m)[\overline{r}/\overline{x}]| = \\
|Op(q_1[\overline{r}/\overline{x}],\ldots,q_m[\overline{r}/\overline{x}])| = \\
\{t \in \mathcal{L} \mid t \equiv Op(q_1[\overline{r}/\overline{x}],\ldots,q_m[\overline{r}/\overline{x}])\} = \\
\text{– \textit{any process belongs to its equivalence class}} \\
\{t \in \mathcal{L} \mid t \equiv Op(q_1[\overline{r}/\overline{x}],\ldots,q_m[\overline{r}/\overline{x}]) \wedge q_i[\overline{r}/\overline{x}] \in |q_i[\overline{r}/\overline{x}]|\} \subseteq \\
\text{– \textit{induction hypothesis and classical law about sets}} \\
\{t \in \mathcal{L} \mid t \equiv Op(q_1[\overline{r}/\overline{x}],\ldots,q_m[\overline{r}/\overline{x}]) \wedge q_i[\overline{r}/\overline{x}] \in [\![\varphi[q_i]]\!]_{\{|\overline{r}|/\overline{x}\}}\} \subseteq \\
\{t \in \mathcal{L} \mid \exists t_1,\ldots,t_m \in \mathcal{L} \cdot t \equiv Op(t_1,\ldots,t_m) \wedge t_i \in [\![\varphi[q_i]]\!]_{\{|\overline{r}|/\overline{x}\}}\} = \\
[\![Op(\varphi[q_1],\ldots,\varphi[q_m])]\!]_{\{|\overline{r}|/\overline{x}\}} = \\
[\![\varphi[Op(q_1,\ldots,q_m)]]\!]_{\{|\overline{r}|/\overline{x}\}}
\end{array}$$

Case 3: $q = \boldsymbol{fix}\, y \cdot s$. We can perform the following deduction:

$$\begin{array}{l}
|(\boldsymbol{fix}\, y \cdot s)[\overline{r}/\overline{x}]| = \\
|\boldsymbol{fix}\, y \cdot s[\overline{r}/\overline{x}]| = \\
\text{– \textit{an unfolding is equivalent}} \\
|s[\overline{r}/\overline{x}][(\boldsymbol{fix}\, y \cdot s[\overline{r}/\overline{x}])/y]| = \\
\text{– \textit{since y does not occur free in $\overline{r}$}} \\
|s[\overline{r}/\overline{x};(\boldsymbol{fix}\, y \cdot s[\overline{r}/\overline{x}])/y]| \subseteq \\
\text{– \textit{induction hypothesis}} \\
[\![\varphi[s]]\!]_{\{|\overline{r}|/\overline{x};|(\boldsymbol{fix}\, y \cdot s)[\overline{r}/\overline{x}]|/y\}}
\end{array}$$

This means that $|(\boldsymbol{fix} y \cdot s)[\overline{r}/\overline{x}]|$ is a postfix point of $\lambda S.[\![\varphi[s]]\!]_{\{|\overline{r}|/\overline{x};S/y\}}$. Due to the semantics of maximal fixpoints, this means that $|(\boldsymbol{fix}\, y \cdot s)[\overline{r}/\overline{x}]| \subseteq [\![\nu\, y \cdot \varphi[s]]\!]_{\{|\overline{r}|/\overline{x}\}} = [\![\varphi[\boldsymbol{fix} y \cdot s]]\!]_{\{|\overline{r}|/\overline{x}\}}$.

B) $p \models \varphi[q] \Rightarrow p \equiv q$. Let $S \stackrel{def}{=} \{(C_1, C_2) \in Con \times Con \mid C_1 \in C_2\}$. We will show that S is a bisimulation. Suppose namely that it is. Then we can prove this case as follows.

Assume that $p \models \varphi[q]$. That is, $p \in [\![\varphi[q]]\!]$. Since $Config[p] = config[p;\pi]$ and $Config[q] = config[q;\pi]$ we obviously have that $Config[p] \in Config[q]$. Due to the definition of S we thus have that $(Config[p], Config[q]) \in S$. Since S is assumed to be a bisimulation, $Config[p] \smile Config[q]$ and therefore $p \equiv q$.

It is easy to see that S is a bisimulation: assume that $(C_1, C_2) \in S$, thus $C_1 \in C_2$. Assume that $C_1 \xrightarrow{\alpha} C_1'$. Then due to lemma (6.22), $C_2 \xrightarrow{\alpha} C_2'$ and $C_1' \in C_2'$. Therefore $(C_1', C_2') \in S$. The case where C_2 moves first is similar.

■

The proof of this theorem is surprisingly involved. A similar theorem has been given in [GS86], but their result is restricted in the sense that q cannot contain more than one ***fix***_·_-construct. Their proof is correspondingly less involved. The proof that we give is similar to the one given in [LT88] for the related theorems 3.6 and 3.10, but for the case $p \models \varphi[q] \Rightarrow p \equiv q$, our proof gets rather more involved. One reason that our proof is longer is due to the fact that we treat parallel composition, which is not the case in [LT88]. But the ***forked*** operator and the associated multiset semantics seems to be the major reason.

The following theorem states that the refinement logic is adequate with respect to process congruence $\equiv$. That is, two processes are equivalent, if and only if they satisfy the same formulae. Let for any process $p \in \mathcal{L}$, $\Psi(p)$ be defined as the set of properties that p satisfies:

$$\Psi(p) = \{\psi \in \Psi \mid p \models \psi\}$$

Then we can state adequateness as follows:

Theorem 6.25 (Adequateness wrt. $\equiv$) *For any processes $p, q \in \mathcal{L}$,*

$$\Psi(p) = \Psi(q) \quad \textit{iff} \quad p \equiv q$$

Proof

A) $\Psi(p) = \Psi(q) \Rightarrow p \equiv q$. Since $\equiv$ is reflexive we know that $p \equiv p$. Due to the previous theorem 6.23, this means that $p \models \varphi[p]$. But then since $\Psi(p) = \Psi(q)$ we have that $q \models \varphi[p]$. We can finally apply theorem 6.23 again on this to obtain $q \equiv p$.

B) $p \equiv q \Rightarrow \Psi(p) = \Psi(q)$. We shall show that if $p \equiv q$ and $p \in [\![\psi]\!]$ for some ψ then $q \in [\![\psi]\!]$. We shall prove a more general property. Let for any set P of processes $\overline{P}$ denote the closure under equivalence: $\{p \mid \exists q \in P \cdot p \equiv q\}$. We shall further say that an environment ρ respects $\equiv$ if for any x it holds that $\rho(x)$ is closed with respect to $\equiv$; that is: $\rho(x) = \overline{\rho(x)}$. The general property that we shall show is the following. Let ρ respect $\equiv$, then:

$$[\![\psi]\!]\rho = \overline{[\![\psi]\!]\rho}$$

From this follows in particular that $[\![\psi]\!] = \overline{[\![\psi]\!]}$ for closed formulae ψ, and then case B of the adequateness proof is done with.

It is easy to see that $[\![\psi]\!]\rho \subseteq \overline{[\![\psi]\!]\rho}$, so we shall concentrate on the other direction: $\overline{[\![\psi]\!]\rho} \subseteq [\![\psi]\!]\rho$. We use induction over the structure of ψ. Only the recursion-case is proved here, the rest of the cases are left to the reader.

$$\begin{array}{ll}
\overline{[\![\nu\, x \cdot \psi]\!]\rho} & \\
= & \\
\overline{\bigcup\{S \subseteq \mathcal{L} \mid S \subseteq [\![\psi]\!]\rho[S/x]\}} & \\
= & \\
\bigcup\{\overline{S} \subseteq \mathcal{L} \mid S \subseteq [\![\psi]\!]\rho[S/x]\} & \textit{– easily seen} \\
\subseteq & \\
\bigcup\{\overline{S} \subseteq \mathcal{L} \mid \overline{S} \subseteq [\![\psi]\!]\rho[\overline{S}/x]\} & \textit{– see below} \\
\subseteq & \\
\bigcup\{S \subseteq \mathcal{L} \mid S \subseteq [\![\psi]\!]\rho[S/x]\} & \textit{– obvious} \\
= & \\
[\![\nu\, x \cdot \psi]\!]\rho &
\end{array}$$

The proof depends on the property that

$$S \subseteq [\![\psi]\!]\rho[S/x] \quad \text{implies} \quad \overline{S} \subseteq [\![\psi]\!]\rho[\overline{S}/x]$$

This follows from the following reasoning. Assume that $S \subseteq [\![\psi]\!]\rho[S/x]$. Due to monotonicity we have that $[\![\psi]\!]\rho[S/x] \subseteq [\![\psi]\!]\rho[\overline{S}/x]$, so consequently $S \subseteq [\![\psi]\!]\rho[\overline{S}/x]$. But then obviously $\overline{S} \subseteq \overline{[\![\psi]\!]\rho[\overline{S}/x]} \subseteq [\![\psi]\!]\rho[\overline{S}/x]$. The last inclusion is due to the induction hypothesis.

■

The last theorem states a compositionality result which is the basis for the proof system to be presented later. It says that components of a system can be replaced by refinements, thereby obtaining a refinement of the system.

Theorem 6.26 (Compositionality) *For any formulae* $\psi_1, \psi'_1, \ldots, \psi_n, \psi'_n$ *and for any operator* $Op \in \{\sum_i \alpha_i;\_ ,\ \_;\_ ,\ \textbf{\textit{fork}}(\_)\ ,\ \textbf{\textit{forked}}(\_)\ ,\ (a)\_ ,\ \{a\}\_\}$ *with arity* n*,*

$$\begin{array}{ll} \textit{Whenever} & \psi_1 \Longrightarrow \psi_1' \textit{ and } \ldots \textit{ and } \psi_n \Longrightarrow \psi_n' \\ \textit{then} & Op(\psi_1, \ldots, \psi_n) \Longrightarrow Op(\psi_1', \ldots, \psi_n') \end{array}$$

Proof

Assume for some process p that $p \models Op(\psi_1, \ldots, \psi_n)$. This means that there exist processes $p_1, \ldots, p_n$ such that $p \equiv Op(p_1, \ldots, p_n)$ and such that $p_1 \models \psi_1$ and ... and $p_n \models \psi_n$. But then since each $\psi_i \Longrightarrow \psi_i'$, also $p_1 \models \psi_1'$ and ... and $p_n \models \psi_n'$. Consequently $p \models Op(\psi_1', \ldots, \psi_n')$.

∎

6.2.4 Derived Forms

We shall now define a collection of derived forms of formulae which are useful for writing specifications. The following derived forms are the obvious ones to define first:

$$\begin{array}{rcl} \textit{\textbf{ff}} & \stackrel{def}{=} & \neg \textit{\textbf{tt}} \\ \psi_1 \vee \psi_2 & \stackrel{def}{=} & \neg(\neg\psi_1 \wedge \neg\psi_2) \\ [\alpha]\psi & \stackrel{def}{=} & \neg <\alpha> \neg\psi \\ \mu\, x \cdot \psi[x] & \stackrel{def}{=} & \neg\nu\, x \cdot \neg\psi[\neg x] \end{array}$$

The formula $[\alpha]\psi$ is satisfied by any process that cannot perform an α-action without becoming a process that satisfies ψ. This includes processes that might not be able to perform an α-action at all. Note that $[\alpha]$***ff*** is satisfied by a process if this cannot perform an α-action. The $\mu\, x \cdot \psi[x]$ construct represents a minimal fixpoint. An understanding of this operator can be obtained by noting that a minimal fixpoint in restricted cases has the same meaning as an infinite disjunction. That is, $\mu x {\cdot} \psi$ means: $\bigvee_{n \in \mathbf{N}} \psi^n[\textit{\textbf{ff}}]$. Maximal fixpoints are related to universal quantification ($\forall$) over states, and are used to specify safety properties. Minimal fixpoints are related to existential quantification ($\exists$) over states, and are used to specify liveness properties.

As an example, consider the formula $\mu\, X \cdot [\tau]X$. A process satisfies this formula if it cannot perform an infinite sequence of τ-actions. To see this, note that this minimal fixpoint is equivalent to the following infinite disjunction: ***ff*** $\vee$ $[\tau]$***ff*** $\vee$ $[\tau][\tau]$***ff*** $\vee$ .... A process satisfying this must satisfy one of the disjuncts, thus after a finite number of τ-actions, it must reach a state where it satisfies $[\tau]$***ff***: it cannot perform yet a τ-action.

We shall often assume some finite set $ACT = \{\alpha_1, \ldots, \alpha_n\}$ of actions. Typically it will be the external actions (in contrast to internal actions on internal channels) of a given process under examination. Then we shall use the following two shorthands.

$$\begin{array}{rcl} <ACT>\psi & \stackrel{def}{=} & <\alpha_1>\psi \vee \ \ldots \ \vee <\alpha_n>\psi \\ [ACT]\psi & \stackrel{def}{=} & [\alpha_1]\psi \wedge \ \ldots \ \wedge [\alpha_n]\psi \end{array}$$

The following derived forms correspond to classical temporal modalities (assuming a fixed finite set ACT of actions):

$$\begin{array}{rcl} \Box\psi & \stackrel{def}{=} & \nu X \cdot \psi \wedge [ACT]X \\ \Box_w\psi & \stackrel{def}{=} & \nu X \cdot \psi \wedge ([ACT]\boldsymbol{ff} \vee <ACT>X) \\ \diamond\,\psi & \stackrel{def}{=} & \mu X \cdot \psi \vee (<ACT>\boldsymbol{tt} \wedge [ACT]X) \\ \diamond_w\,\psi & \stackrel{def}{=} & \mu X \cdot \psi \vee <ACT>X \\ \psi_1\,\mathcal{U}\,\psi_2 & \stackrel{def}{=} & \mu X \cdot \psi_2 \vee (\psi_1 \wedge <ACT>\boldsymbol{tt} \wedge [ACT]X) \\ \psi_1\,\mathcal{U}_w\,\psi_2 & \stackrel{def}{=} & \nu X \cdot \psi_2 \vee (\psi_1 \wedge [ACT]X) \end{array}$$

$\Box\psi$ (strong invariance) is satisfied by a process if for *any* computation of the process, all states in that computation satisfy ψ. By a computation we mean a path through the transition tree denoted by the process. Similarly $\Box_w\psi$ (weak invariance) is satisfied if for *some* computation, all states satisfy ψ. $\diamond\,\psi$ (eventually) is satisfied if for *any* computation, some state in that computation satisfies ψ. Similarly $\diamond_w\,\psi$ (possibly) is satisfied if for *some* computation, some state satisfies ψ. Finally, $\psi_1\,\mathcal{U}\,\psi_2$ (strong until) and $\psi_1\,\mathcal{U}_w\,\psi_2$ (weak until) are satisfied if for any computation ψ_1 holds until ψ_2 holds. For strong until, ψ_2 is required to hold in some state (in all computations). In contrast, for weak until, computations are allowed where ψ_2 never hold, in which case ψ_1 consequently must hold throughout such a computation.

Finally, the following derived forms are heavely used in the example in section 6.2.6. Let A be a set of actions:

$$\begin{array}{rcl} \ll A \gg & \stackrel{def}{=} & <A>\boldsymbol{tt} \wedge [ACT-A]\boldsymbol{ff} \\ \circ\{A\} & \stackrel{def}{=} & \mu X\cdot \ll A,\tau\gg \wedge [\tau]X \\ \circ_w\{A\} & \stackrel{def}{=} & \nu X\cdot \ll A,\tau\gg \wedge [\tau]X \\ \odot\,\psi & \stackrel{def}{=} & \mu X \cdot \psi \vee (\ll\tau\gg \wedge [\tau]X) \end{array}$$

$\ll A\gg$ is satisfied by a process if it can perform at least one of the actions in the set A, and it cannot perform actions outside A. $\circ\{A\}$ is satisfied by a process if it will perform one of the actions in A after a finite number of τ-actions. $\circ_w\{A\}$ is satisfied by a process if either it performs an infinite sequence of τ-actions, or if it performs one of the actions in A after a finite number of τ-actions. $\odot\,\psi$ is satisfied by a process if it after a finite number of τ-actions will reach a state satisfying ψ.

6.2.5 Proof Rules

In this section we axiomatise the refinement relation $\Longrightarrow$ between formulae in our logic. That is, we provide a rule based way of deducing statements of the form $\psi_1 \Longrightarrow \psi_2$. We shall use $\psi_1 \Longleftrightarrow \psi_2$ as short for $\psi_1 \Longrightarrow \psi_2$ and $\psi_2 \Longrightarrow \psi_1$. We shall not deal with general negation formulae $(\neg\psi)$ since this is regarded hard. Instead, the rules will concern derived forms such as for example minimal fixpoints and the necessity operator $[\alpha]\psi$. That is: limited forms of negation.

The first group of rules presented expresses how programming constructs are expanded into modal constructs. Typically, to the left of $\Longrightarrow$ we find some programming construct at the outermost level, while on the right hand side of $\Longrightarrow$, we find a modal construct at the outermost level.

(**Nil**$[\alpha]$) $$\textit{\textbf{nil}} \Longrightarrow [\alpha]\textit{\textbf{ff}}$$

(**Act**$<\alpha>$) $$\alpha \Longrightarrow <\alpha>\textit{\textbf{nil}}$$

(**Act**$[\alpha]$) $$\alpha \Longrightarrow [\alpha]\textit{\textbf{nil}}$$

(**Act**$[\beta]$) $$\alpha \Longrightarrow [\beta]\textit{\textbf{ff}} \qquad (\alpha \neq \beta)$$

(**Forked**$<\alpha>$) $$\frac{\psi \Longrightarrow <\alpha>\psi'}{\textit{\textbf{forked}}(\psi) \Longrightarrow <\alpha>\textit{\textbf{forked}}(\psi')}$$

(**Forked**$[\alpha]$) $$\frac{\psi \Longrightarrow [\alpha]\psi'}{\textit{\textbf{forked}}(\psi) \Longrightarrow [\alpha]\textit{\textbf{forked}}(\psi')}$$

(**Sum**$<\alpha>$) $$\textstyle\sum_i \alpha_i;\psi_i \Longrightarrow <\alpha_i>\psi_i$$

(**Sum**$[\alpha]$) $$\textstyle\sum_i \alpha_i;\psi_i \Longrightarrow [\alpha]\bigvee_{i:\alpha=\alpha_i}\psi_i \qquad ([\alpha]\textit{\textbf{ff}} \textit{ in case } \forall i\cdot\alpha\neq\alpha_i)$$

(**Seq**$<\alpha>$) $$\frac{\psi_1 \Longrightarrow <\alpha>\psi_1'}{\psi_1;\psi_2 \Longrightarrow <\alpha>(\psi_1';\psi_2)}$$

(**Seq**$_\alpha[\alpha]$) $$\alpha;\psi \Longrightarrow [\alpha]\psi$$

(**Seq**$_\alpha[\beta]$) $$\alpha;\psi \Longrightarrow [\beta]\textit{\textbf{ff}} \qquad (\alpha \neq \beta)$$

($\mathbf{Seq}_\Phi$ <α>) $$\frac{\psi_2 \Longrightarrow <\alpha> \psi_2'}{\boldsymbol{forked}(\psi_1); \psi_2 \Longrightarrow <\alpha> (\boldsymbol{forked}(\psi_1); \psi_2')}$$

($\mathbf{Seq}_\Phi$ <τ>) $$\frac{\psi_1 \Longrightarrow <c> \psi_1' \;,\; \psi_2 \Longrightarrow <rev(c)> \psi_2'}{\boldsymbol{forked}(\psi_1); \psi_2 \Longrightarrow <\tau> (\boldsymbol{forked}(\psi_1'); \psi_2')}$$

($\mathbf{Seq}_\Phi[c]$) $$\frac{\psi_1 \Longrightarrow [c]\psi_1' \;,\; \psi_2 \Longrightarrow [c]\psi_2'}{\boldsymbol{forked}(\psi_1); \psi_2 \Longrightarrow [c]((\boldsymbol{forked}(\psi_1'); \psi_2) \vee (\boldsymbol{forked}(\psi_1); \psi_2'))}$$

($\mathbf{Seq}_\Phi[\tau]$) $$\frac{\forall \alpha \in \mathcal{A} \cdot (\psi_1 \Longrightarrow [\alpha]\psi_1^\alpha \;,\; \psi_2 \Longrightarrow [\alpha]\psi_2^\alpha)}{\boldsymbol{forked}(\psi_1); \psi_2 \Longrightarrow [\tau]((\boldsymbol{forked}(\psi_1^\tau); \psi_2) \vee (\boldsymbol{forked}(\psi_1); \psi_2^\tau) \vee \bigvee_c (\boldsymbol{forked}(\psi_1^c); \psi_2^{rev(c)}))}$$

(**Allocated**<α>) $$\frac{\psi \Longrightarrow <\alpha> \psi'}{\{a\}\psi \Longrightarrow <\alpha> \{a\}\psi'}\; \alpha \neq a$$

(**Allocated**$[\alpha]_1$) $$\frac{\psi \Longrightarrow [\alpha]\psi'}{\{a\}\psi \Longrightarrow [\alpha]\{a\}\psi'}$$

(**Allocated**$[\alpha]_2$) $$\{a\}\psi \Longrightarrow [\alpha]\boldsymbol{ff} \qquad (\alpha = a)$$

The information-dense rule ($\mathbf{Seq}_\Phi[\tau]$) perhaps requires an explanation. The assumption above the line states that whenever ψ_i (for $i \in \{1,2\}$) allows an action α, then the result is called ψ_i^α. Below the line, it is stated what results after a τ-action. Either the τ-action comes from ψ_1 or ψ_2, in which case the result is in $(\boldsymbol{forked}(\psi_1^\tau); \psi_2) \vee (\boldsymbol{forked}(\psi_1); \psi_2^\tau)$. Or the τ-action comes from a communication between ψ_1 (offering c) and ψ_2 (offering $rev(c)$), in which case the result is in $\boldsymbol{forked}(\psi_1^c); \psi_2^{rev(c)}$. As there may be many such communications, we take the disjunction between all the results. The rules (**Nil**$[\alpha]$), (**Act**<α>), (**Act**$[\alpha]$), (**Act**$[\beta]$), ($\mathbf{Seq}_\alpha[\alpha]$) and ($\mathbf{Seq}_\alpha[\beta]$) are in fact derived from (**Sum**<α>) and (**Sum**$[\alpha]$).

The rules in the next group are inspired by the axiomatisation of FC_r, that was presented in section 6.1. They state equivalence ($\Longleftrightarrow$) between formulae where there is a programming construct at the outermost level, and they can be used to perform direct transformations of "programs".

($\mathbf{EqSequence}_1$) $$\psi; \boldsymbol{nil} \Longleftrightarrow \psi$$

($\mathbf{EqSequence}_2$) $$\boldsymbol{nil}; \psi \Longleftrightarrow \psi$$

($\mathbf{EqSequence}_3$) $$(\psi_1; \psi_2); \psi_3 \Longleftrightarrow \psi_1; (\psi_2; \psi_3)$$

($\mathbf{EqSequence}_4$) $$(\textstyle\sum_i \alpha_i; \psi_i); \psi \Longleftrightarrow \textstyle\sum_i \alpha_i; (\psi_i; \psi)$$

(**EqChoice**) $\sum_{i\,\in\,I-\{j\}} \alpha_i;\psi_i \Longrightarrow \sum_{i\,\in\,I} \alpha_i;\psi_i$ – *provided* $\exists i \in I-\{j\}\cdot(\alpha_i;\psi_i)=(\alpha_j;\psi_j)$

(**EqFork**$_1$) $$\boldsymbol{fork}(\psi) \Longleftrightarrow \tau;\boldsymbol{forked}(\psi)$$

(**EqFork**$_2$) $$\boldsymbol{forked}(\alpha;\boldsymbol{forked}(\psi)+\textstyle\sum_{\dots}\dots) \Longleftrightarrow \boldsymbol{forked}(\alpha;\psi+\textstyle\sum_{\dots}\dots)$$

(**EqFork**$_3$) $$\boldsymbol{forked}(\boldsymbol{nil}) \Longleftrightarrow \boldsymbol{nil}$$

(**EqChannel**$_1$) $$(a)\psi \Longleftrightarrow \tau;\{a\}\psi$$

(**EqChannel**$_2$) $$\{a\}\boldsymbol{nil} \Longleftrightarrow \boldsymbol{nil}$$

(**EqChannel**$_3$) $$\{a\}\boldsymbol{forked}(\psi) \Longleftrightarrow \boldsymbol{forked}(\{a\}\psi)$$

Note in rule (**EqChoice**) that in contrast to the other rules in this group, $\Longrightarrow$ is used instead of $\Longleftrightarrow$. That is: it does for example not hold that $\alpha;\boldsymbol{tt}+\alpha;\boldsymbol{tt} \Longrightarrow \alpha;\boldsymbol{tt}$. To see this, observe that the program $\alpha;\boldsymbol{nil}+\alpha;\alpha$ satisfies $\alpha;\boldsymbol{tt}+\alpha;\boldsymbol{tt}$, but it does not satisfy $\alpha;\boldsymbol{tt}$.

Also inspired by the axiomatisation of FC_r are the following expansion laws. The first two of these laws state how ***forked*** can be (partially) expanded into non-determinism. The last law states how channel allocation can be eliminated. Note that the rules are not only defined on normal forms, but on sums in general.

Let $F_\alpha \overset{def}{=} \sum_i \alpha_i;\psi_i$ and $F_\beta \overset{def}{=} \sum_j \beta_j;\psi'_j$. Then:

(**Expansion**$_1$) $$\boldsymbol{forked}(F_\alpha);F_\beta \Longrightarrow \left\{\begin{array}{l} \sum_i \alpha_i;\boldsymbol{forked}(\psi_i);F_\beta\ +\\ \sum_j \beta_j;\boldsymbol{forked}(F_\alpha);\psi'_j\ +\\ \sum_{\alpha_i=rev(\beta_j)} \tau;\boldsymbol{forked}(\psi_i);\psi'_j \end{array}\right.$$

(**Expansion**$_2$) $$\boldsymbol{forked}(F_\alpha);\boldsymbol{forked}(F_\beta) \Longrightarrow \left\{\begin{array}{l} \boldsymbol{forked}(\\ \quad\sum_i \alpha_i;\boldsymbol{forked}(\psi_i);\boldsymbol{forked}(F_\beta)\ +\\ \quad\sum_j \beta_j;\boldsymbol{forked}(F_\alpha);\boldsymbol{forked}(\psi'_j)\ +\\ \quad\sum_{\alpha_i=rev(\beta_j)} \tau;\boldsymbol{forked}(\psi_i);\boldsymbol{forked}(\psi'_j)) \end{array}\right.$$

(**Expansion**$_3$) $$\{a\}\textstyle\sum_i \alpha_i;\psi_i \Longleftrightarrow \sum_{\alpha_i\neq a} \alpha_i;\{a\}\psi_i$$

The following rules deal with recursion by stating how to manipulate maximal and minimal fixpoints.

(**UnfoldMax**) $$\nu\, x\cdot\psi \Longleftrightarrow \psi[\nu\, x\cdot\psi/x]$$

$$\textbf{(UnfoldMin)} \quad \mu\, x \cdot \psi \Longleftrightarrow \psi[\mu\, x \cdot \psi / x]$$

$$\textbf{(Maximal)} \qquad \frac{\psi' \Longrightarrow \psi[\psi'/x]}{\psi' \Longrightarrow \nu\, x \cdot \psi}$$

$$\textbf{(Minimal)} \qquad \frac{\psi[\psi'/x] \Longrightarrow \psi'}{\mu\, x \cdot \psi \Longrightarrow \psi'}$$

The rules (**UnfoldMax**) and (**UnfoldMin**) state that maximal as well as minimal fixpoints can be unfolded (or dually: infolded). The other two rules, (**Maximal**) and (**Minimal**), reflect closely the semantics of fixpoints given in definition (6.15). Take for example the rule (**Maximal**). The denotation of a maximal fixpoint is as follows: $[\![\nu\, x \cdot \psi]\!]\rho = \bigcup\{S \subseteq \mathcal{L} \mid S \subseteq [\![\psi]\!]\rho[S/x]\}$. So for any set $S \subseteq \mathcal{L}$, if it holds that $S \subseteq [\![\psi]\!]\rho[S/x]$, then $S \subseteq [\![\nu\, x \cdot \psi]\!]\rho$, since the maximal fixpoint is the union of such S. This is exactly what is reflected in the proof rule, where ψ' represents S. A similar explanation can be given for the (**Minimal**) rule.

As an example, suppose we want to prove that $a!; a? \Longrightarrow \nu X \cdot [\tau]\boldsymbol{f\!f} \wedge [ACT]X$ (the process can never perform a τ-action). First, we prove that: $(a!; a? \vee a? \vee \boldsymbol{nil}) \Longrightarrow [\tau]\boldsymbol{f\!f} \wedge [ACT](a!; a? \vee a? \vee \boldsymbol{nil})$, which is easy. This corresponds to formulate the state space of reachable states as a disjunction, and then to reason about this state space. As a consequence $(a!; a? \vee a? \vee \boldsymbol{nil}) \Longrightarrow \nu\, X \cdot [\tau]\boldsymbol{f\!f} \wedge [ACT]X$. Finally the result follows due to transitivity (se below) and the fact that $a!; a? \Longrightarrow a!; a? \vee a? \vee \boldsymbol{nil}$.

In certain proofs in the example to be given later, we shall use the rule: $\mu\, x \cdot \psi[x] \Longleftrightarrow \bigvee_{n \in \mathbf{N}} \psi^n[\boldsymbol{f\!f}]$. That is, to prove a property of a minimal fixpoint, it is first expanded into an infinite disjunction, where after we use induction over the natural numbers to prove that a property holds for each disjunct. This trick is only valid if ψ is continuous. In general this holds if ψ does not contain a nested mixture of maximal and minimal fixpoints such that inner minimal fixpoints refer to outer maximal (or vice verca).

The following rules concern the interplay between propositional logic and modal constructs:

$$\textbf{(Or}{<}\alpha{>}\textbf{)} \quad {<}\alpha{>}(\psi_1 \vee \psi_2) \Longleftrightarrow {<}\alpha{>}\psi_1 \vee {<}\alpha{>}\psi_2$$

$$\textbf{(Or}[\alpha]\textbf{)} \quad [\alpha](\psi_1 \vee \psi_2) \Longrightarrow [\alpha]\psi_1 \vee {<}\alpha{>}\psi_2$$

$$\textbf{(And}[\alpha]\textbf{)} \quad [\alpha](\psi_1 \wedge \psi_2) \Longleftrightarrow [\alpha]\psi_1 \wedge [\alpha]\psi_2$$

$$(\textbf{And}{<}\alpha{>}) \qquad {<}\alpha{>}\,\psi_1 \,\wedge\, [\alpha]\psi_2 \Longrightarrow {<}\alpha{>}(\psi_1 \,\wedge\, \psi_2)$$

$$(\textbf{False}{<}\alpha{>}) \qquad {<}\alpha{>}\,\textit{\textbf{ff}} \Longleftrightarrow \textit{\textbf{ff}}$$

$$(\textbf{True}[\alpha]) \qquad [\alpha]\,\textit{\textbf{tt}} \Longleftrightarrow \textit{\textbf{tt}}$$

The following rule says that the operators of FC_r distribute over logical "or" ($\vee$). Let $Op \in \{\sum_i \alpha_i;\_ ,\ \_;\_ ,\ \textit{\textbf{fork}}(\_)\ ,\ \textit{\textbf{forked}}(\_)\ ,\ (a)\_ ,\ \{a\}\_\}$. Then:

$$(\vee\textbf{-Distr}) \quad Op(\ldots, \textstyle\bigvee_{i\,\in\,I}\psi_i, \ldots) \Longleftrightarrow \textstyle\bigvee_{i\,\in\,I} Op(\ldots, \psi_i, \ldots)$$

If we interpretate $\bigvee_{i\,\in\,\{\}}\psi_i$ as $\textit{\textbf{ff}}$, then a special case of the above rule is the following where $I = \{\}$:

$$(\textbf{Strict}) \quad Op(\ldots, \textit{\textbf{ff}}, \ldots) \Longleftrightarrow \textit{\textbf{ff}}$$

The monotonicity of our operators is stated as follows. Assume that OP ranges over the operators $\{\_\vee\_,\ \_\wedge\_,\ {<}\alpha{>}\_,\ [\alpha]\_,\ \sum_i \alpha_i;\_,\ \_;\_,\ \textit{\textbf{fork}}(\_),\ \textit{\textbf{forked}}(\_),\ (a)\_,\ \{a\}\_\}$, then:

$$(\textbf{Monotone}) \quad \frac{\psi_1 \Longrightarrow \psi_2}{OP(\ldots, \psi_1, \ldots) \Longrightarrow OP(\ldots, \psi_2, \ldots)}$$

Think of the propositional logic constants and operators as sets and operations on sets. Thats is, $\textit{\textbf{tt}}$ is the set ($\mathcal{L}$) of all processes, $\textit{\textbf{ff}}$ is the empty set ($\{\}$), the logical connective $\wedge$ is set intersection ($\cap$), and $\vee$ is set union ($\cup$). Finally $\Longrightarrow$ is subset inclusion ($\subseteq$). Then a mass of rules hold (corresponding to the rules of traditional propositional logic), for example the following:

$$(\textbf{Reflexive}) \qquad \psi \Longrightarrow \psi$$

$$(\textbf{Transitive}) \qquad \frac{\psi_1 \Longrightarrow \psi_2\ ,\ \psi_2 \Longrightarrow \psi_3}{\psi_1 \Longrightarrow \psi_3}$$

$$(\textbf{Empty}) \qquad \textit{\textbf{ff}} \Longrightarrow \psi$$

$$(\textbf{All}) \qquad \psi \Longrightarrow \textit{\textbf{tt}}$$

$$\textbf{(And)} \qquad \frac{\psi \Longrightarrow \psi_1 \,,\, \psi \Longrightarrow \psi_2}{\psi \Longrightarrow \psi_1 \wedge \psi_2}$$

Theorem 6.27 *The refinement rules are sound. That is, whenever the refinement rules above allow us to deduce $\psi_1 \Longrightarrow \psi_2$, for some formulae ψ_1 and ψ_2 in Ψ, then $\mathcal{L}(\psi_1) \subseteq \mathcal{L}(\psi_2)$.*

Proof

Left to the reader.

∎

We give two small examples showing how to apply the proof rules. A more realistic example is given in section 6.2.6.

Example 6.28 *Suppose we want to implement the specification (please ignore the lack of practical flavor of this example): $<\tau>(<\alpha>\boldsymbol{tt} \wedge\ <\beta>\boldsymbol{tt})$. We refine this specification in two steps. In the first step, we decide to construct our program as a sequential composition beginning with a fork:*
$\boldsymbol{fork}(<\alpha>\boldsymbol{tt}); <\beta>\boldsymbol{tt}$. We thus have to show that

$$\boldsymbol{fork}(<\alpha>\boldsymbol{tt}); <\beta>\boldsymbol{tt} \Longrightarrow <\tau>(<\alpha>\boldsymbol{tt} \wedge\ <\beta>\boldsymbol{tt})$$

As a kind of lemma, we first show the general result that $\boldsymbol{fork}(\psi) \Longrightarrow <\tau>$ $\boldsymbol{forked}(\psi)$:

(a) $\boldsymbol{fork}(\psi) \Longrightarrow \tau; \boldsymbol{forked}(\psi)$– by $\mathbf{EqFork}_1$
(b) $\tau \Longrightarrow <\tau>\boldsymbol{nil}$– by $\mathbf{Act}<\alpha>$
(c) $\tau; \boldsymbol{forked}(\psi) \Longrightarrow <\tau>(\boldsymbol{nil}; \boldsymbol{forked}(\psi))$– from (b) and $\mathbf{Seq}<\alpha>$
(d) $\boldsymbol{nil}; \boldsymbol{forked}(\psi) \Longrightarrow \boldsymbol{forked}(\psi)$– by $\mathbf{EqSequence}_2$
(e) $<\tau>(\boldsymbol{nil}; \boldsymbol{forked}(\psi)) \Longrightarrow <\tau>\boldsymbol{forked}(\psi)$– from (d) and **Monotone**
(f) $\boldsymbol{fork}(\psi) \Longrightarrow <\tau>\boldsymbol{forked}(\psi)$– from (a, c, e) and **Transitive**

We now proceed as follows:

(1)	$\boldsymbol{fork}(<\alpha> \boldsymbol{tt}) \Longrightarrow <\tau> \boldsymbol{forked}(<\alpha> \boldsymbol{tt})$	– see above lemma
(2)	$\boldsymbol{fork}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt} \Longrightarrow <\tau> (\boldsymbol{forked}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt})$	– from (1) and **Seq**$<\alpha>$
(3)	$<\alpha> \boldsymbol{tt} \Longrightarrow <\alpha> \boldsymbol{tt}$	– by **Reflexive**
(4)	$\boldsymbol{forked}(<\alpha> \boldsymbol{tt}) \Longrightarrow <\alpha> \boldsymbol{forked}(\boldsymbol{tt})$	– from (3) and **Forked**$<\alpha>$
(5)	$\boldsymbol{forked}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt} \Longrightarrow <\alpha> (\boldsymbol{forked}(\boldsymbol{tt}); <\beta> \boldsymbol{tt})$	– from (4) and **Seq**$<\alpha>$
(6)	$\boldsymbol{forked}(\boldsymbol{tt}); <\beta> \boldsymbol{tt} \Longrightarrow \boldsymbol{tt}$	– by **All**
(7)	$<\alpha> (\boldsymbol{forked}(\boldsymbol{tt}); <\beta> \boldsymbol{tt}) \Longrightarrow <\alpha> \boldsymbol{tt}$	– from (6) and **Monotone**
(8)	$\boldsymbol{forked}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt} \Longrightarrow <\alpha> \boldsymbol{tt}$	– (5),(7) and **Transitive**
(9)	$<\beta> \boldsymbol{tt} \Longrightarrow <\beta> \boldsymbol{tt}$	– by **Reflexive**
(10)	$\boldsymbol{forked}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt} \Longrightarrow <\beta> (\boldsymbol{forked}(<\alpha> \boldsymbol{tt}); \boldsymbol{tt})$	– from (9) and **Seq**$_{\Phi}$ $<\alpha>$
(11)	$\boldsymbol{forked}(<\alpha> \boldsymbol{tt}); \boldsymbol{tt} \Longrightarrow \boldsymbol{tt}$	– by **All**
(12)	$<\beta> (\boldsymbol{forked}(<\alpha> \boldsymbol{tt}); \boldsymbol{tt}) \Longrightarrow <\beta> \boldsymbol{tt}$	– from (11) and **Monotone**
(13)	$\boldsymbol{forked}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt} \Longrightarrow <\beta> \boldsymbol{tt}$	– from (10),(12) and **Transitive**
(14)	$\boldsymbol{forked}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt} \Longrightarrow <\alpha> \boldsymbol{tt} \wedge\ <\beta> \boldsymbol{tt}$	– from (8),(13) and **And**
(15)	$<\tau> \boldsymbol{forked}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt} \Longrightarrow <\tau> (<\alpha> \boldsymbol{tt} \wedge\ <\beta> \boldsymbol{tt})$	– from (14) and **Monotone**
(16)	$\boldsymbol{fork}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt} \Longrightarrow <\tau> (<\alpha> \boldsymbol{tt} \wedge\ <\beta> \boldsymbol{tt})$	– from (2),(15) and **Transitive**

We can now further refine the specification $\boldsymbol{fork}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt}$ *into* $\boldsymbol{fork}(\alpha); \beta$. *We shall show that this is a correct refinement. That is, we shall show that:*

$$\boldsymbol{fork}(\alpha); \beta \Longrightarrow \boldsymbol{fork}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt}$$

We proceed as follows:

(17)	$\alpha \Longrightarrow <\alpha> \boldsymbol{nil}$	– by **Act**$<\alpha>$
(18)	$\boldsymbol{nil} \Longrightarrow \boldsymbol{tt}$	– by **All**
(19)	$<\alpha> \boldsymbol{nil} \Longrightarrow <\alpha> \boldsymbol{tt}$	– from (18) and **Monotone**
(20)	$\alpha \Longrightarrow <\alpha> \boldsymbol{tt}$	– (17),(19) and **Transitive**
(21)	$\beta \Longrightarrow <\beta> \boldsymbol{tt}$	– similar to (20)
(22)	$\boldsymbol{fork}(\alpha) \Longrightarrow \boldsymbol{fork}(<\alpha> \boldsymbol{tt})$	– from (20) and **Monotone**
(23)	$\boldsymbol{fork}(\alpha); \beta \Longrightarrow \boldsymbol{fork}(<\alpha> \boldsymbol{tt}); \beta$	– from (22) and **Monotone**
(24)	$\boldsymbol{fork}(<\alpha> \boldsymbol{tt}); \beta \Longrightarrow \boldsymbol{fork}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt}$	– from (21) and **Monotone**
(25)	$\boldsymbol{fork}(\alpha); \beta \Longrightarrow \boldsymbol{fork}(<\alpha> \boldsymbol{tt}); <\beta> \boldsymbol{tt}$	– from (23),(24) and **Transitive**

Now, the final program implements the initial specification:

(26)	$\boldsymbol{fork}(\alpha); \beta \Longrightarrow <\tau> (<\alpha> \boldsymbol{tt} \wedge\ <\beta> \boldsymbol{tt})$	– from (16),(25) and **Transitive**

Example 6.29 *We want to show that*

$$\boldsymbol{fork}(\alpha); \beta \Longrightarrow \tau; (<\beta> \boldsymbol{forked}(\alpha))$$

Note that in example 6.17 we proved a very related result, namely that $\boldsymbol{fork}(\alpha); \beta \models \tau; (<\beta> \boldsymbol{forked}(\alpha))$. *We proceed as follows:*

(1)	$\boldsymbol{fork}(\alpha) \Longrightarrow \tau; \boldsymbol{forked}(\alpha)$	– by **EqFork**$_1$
(2)	$\boldsymbol{fork}(\alpha); \beta \Longrightarrow (\tau; \boldsymbol{forked}(\alpha)); \beta$	– from (1) and **Monotone**
(3)	$(\tau; \boldsymbol{forked}(\alpha)); \beta \Longrightarrow \tau; (\boldsymbol{forked}(\alpha); \beta)$	– by **EqSequence**$_3$
(4)	$\boldsymbol{fork}(\alpha); \beta \Longrightarrow \tau; (\boldsymbol{forked}(\alpha); \beta)$	– from (2),(3) and **Transitive**
(5)	$\beta \Longrightarrow <\beta> \boldsymbol{nil}$	– by **Act**$<\alpha>$
(6)	$\boldsymbol{forked}(\alpha); \beta \Longrightarrow <\beta> (\boldsymbol{forked}(\alpha); \boldsymbol{nil})$	– from (5) and **Seq**$_\Phi$ $<\alpha>$
(7)	$\boldsymbol{forked}(\alpha); \boldsymbol{nil} \Longrightarrow \boldsymbol{forked}(\alpha)$	– by **EqSequence**$_1$
(8)	$<\beta> (\boldsymbol{forked}(\alpha); \boldsymbol{nil}) \Longrightarrow <\beta> \boldsymbol{forked}(\alpha)$	– from (7) and **Monotone**
(9)	$\boldsymbol{forked}(\alpha); \beta \Longrightarrow <\beta> \boldsymbol{forked}(\alpha)$	– from (6),(8) and **Transitive**
(10)	$\tau; (\boldsymbol{forked}(\alpha); \beta) \Longrightarrow \tau; (<\beta> \boldsymbol{forked}(\alpha))$	– from (9) and **Monotone**
(11)	$\boldsymbol{fork}(\alpha); \beta \Longrightarrow \tau; (<\beta> \boldsymbol{forked}(\alpha))$	– from (4),(10) and **Transitive**

6.2.6 Example

In this section we shall apply the presented refinement calculus to an example. That is, we shall provide a sequence of specifications, each (except the first) postulated to be a refinment of its predecessor, and with the final one being a program in FC_r. We shall also prove the postulated refinements.

The example taken is that of a protocol, and for this we will provide an initial specification, a design and a program. The initial specification will be a pure modal logic formula containing no programming constructs. The design will be a mixture of modal logic and programming constructs, while finally the program will consist solely of programming constructs.

In the first section we present the specification, the design and the program. In the next section we prove that the design refines the specification. In the third section we prove that the program refines the design.

Specification, Design and Program

Specification

The protocol is very simple in that it just transmits messages. There are two actions: *accept*? and *deliver*!, and the behaviour of the protocol is supposed to be an infinite sequence of $accept? - deliver!$ communications (disregarding the τ-action). The protocol is illustrated in figure 6.1.

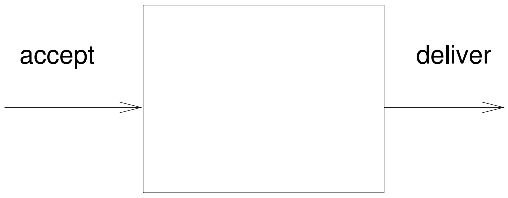

Figure 6.1: Configuration Diagram for *Protocol*

The protocol specification is as follows:

$$\begin{array}{lcl} Protocol & \stackrel{def}{=} & \circ\{accept?\} \wedge \\ & & \Box([accept?] \circ_w \{deliver!\}) \wedge \\ & & \Box([deliver!] \circ \{accept?\}) \end{array}$$

The first line says that the next action (after a finite number of τ's) must be *accept*?. The second line says that whenever an *accept*? is performed, then the next action will be *deliver*!, alternatively the protocol may diverge with an infinite number of τ's. This divergence could correspond to the repeated loss of the message by an unreliable medium. Since we later introduce an unreliable medium, we allow divergence at this stage. The third line says that whenever a *deliver*! is performed, then the next action will be *accept*?.

Design

In the next design, we implement the protocol as three processes composed in parallel: a sender, a medium and a receiver. This is illustrated in figure 6.2.

The sender accepts a message from the external world by an *accept*?-action and passes it to the medium by an *inmed*!-action, after which it waits for either an acknowledgement, *ack*?, or an error message, *err*?, indicating that the message is lost. If the sender receives an acknowledgement it returns to its initial state, otherwise it tries to resend the message.

After the medium has received a message, *inmed*?, it either looses its information which is signaled by the *err*!-action, or the message gets to the receiver by an *outmed*!-action. In both cases the medium returns to its initial state.

The receiver receives a message by *outmed*?, delivers it to the external environment by *deliver*!, then it sends an acknowledgement, *ack*!, and returns to its initial state.

The protocol design is as follows. Note that the silent versions of the forking

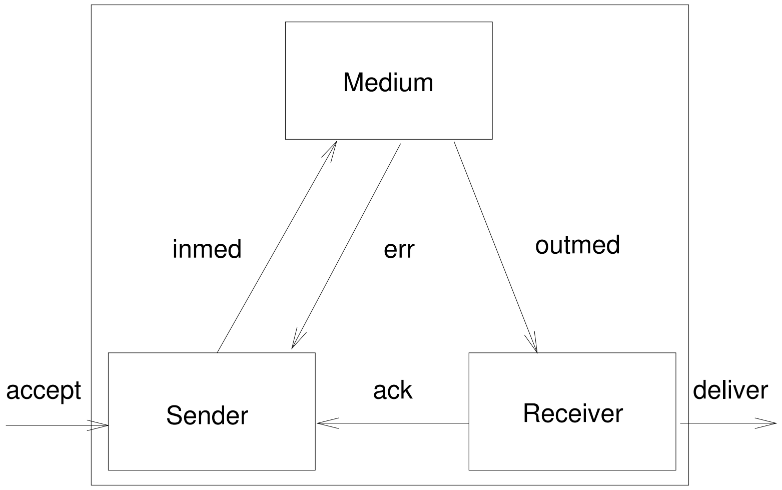

Figure 6.2: Configuration Diagram for *Protocol'*

and channel allocation operators have been used in this example. This is just to avoid uninteresting proof steps concerned with the τ-transitions coming from the alternative non-silent versions.

$$
\begin{array}{rcl}
\mathit{Protocol'} & \stackrel{def}{=} & \{inmed\}\{outmed\}\{ack\}\{err\} \\
& & \boldsymbol{forked}(Receiver); \boldsymbol{forked}(Medium); \boldsymbol{forked}(Sender) \\
\mathit{Sender} & \stackrel{def}{=} & \nu\, S \cdot accept?; \nu\, S_1 \cdot inmed!; (ack?; S + err?; S_1) \\
\mathit{Receiver} & \stackrel{def}{=} & \nu\, R \cdot outmed?; deliver!; ack!; R \\
\mathit{Medium} & \stackrel{def}{=} & \circ\{inmed?\} \wedge \\
& & \Box([inmed?] \circ \{outmed!, err!\}) \wedge \\
& & \Box([outmed!, err!] \circ \{inmed?\})
\end{array}
$$

The channels *inmed*, *outmed*, *ack* and *err* are all local. The sender and the receiver are given as programs in FC_r, while the medium is underspecified in terms of a formula in pure modal logic. This is then an example of how programming constructs can be mixed with specification constructs. The medium specification says that the next action must be *inmed*?, that whenever an *inmed*?-action is performed, the next action will be either *outmed*! or *err*! (divergence is not allowed), and that whenever an *outmed*! or *err*!-action is performed, the next action will be *inmed*?.

Program

In the last step, we implement the medium as two processes composed in parallel: an unstable medium and a recovery system. The final protocol is illustrated in figure 6.3.

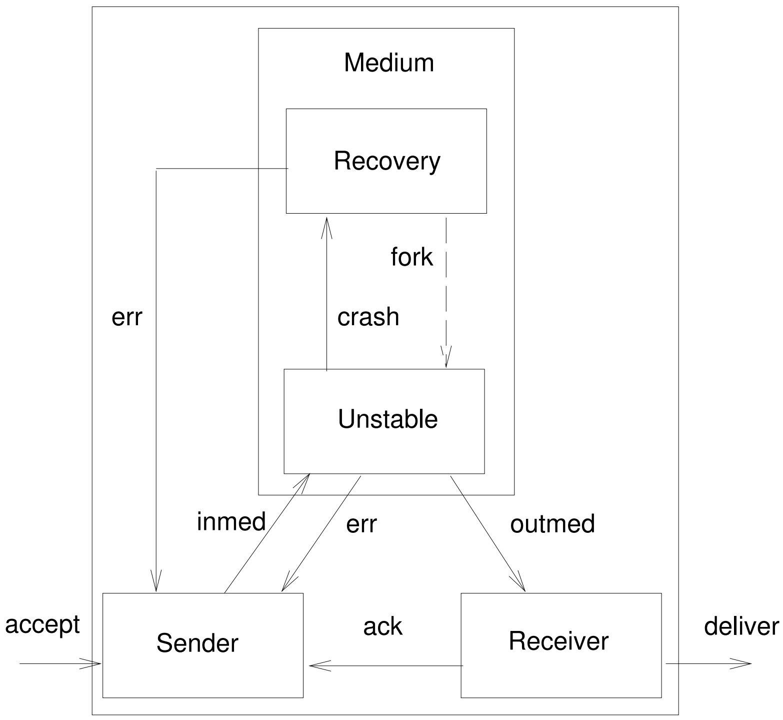

Figure 6.3: Configuration Diagram for *Protocol"*

After the unstable medium has received a message, $inmed?$, it can loose the message which is signaled by the $err!$-action, or the message gets to the receiver by an $outmed!$-action. In both cases the unstable medium returns to its initial state. A third possibility is that the unstable medium crashes, which is signaled to the environment by an $crash!$-action. After a crash, the unstable medium is dead.

The recovery system receives the $crash?$ signal, sends an error message, $err!$, to the sender (telling that the message is lost), and finally starts a new unstable medium.

The implementation of the medium is as follows:

$$
\begin{array}{lcl}
\mathit{Medium'} & \stackrel{def}{=} & \{crash\}\boldsymbol{forked}(\mathit{Unstable}); \mathit{Recovery} \\
\mathit{Unstable} & \stackrel{def}{=} & \nu\, U \cdot inmed?; (outmed!; U + err!; U + crash!; \boldsymbol{nil}) \\
\mathit{Recovery} & \stackrel{def}{=} & \nu\, R \cdot crash?; err!; \boldsymbol{forked}(\mathit{Unstable}); R
\end{array}
$$

We can finally obtain the protocol program by replacing the implementation of the medium for the medium specification in the design (we will not repeat the definitions of the sender and the receiver):

$$\textit{Protocol''} \stackrel{def}{=} \textit{Protocol'}[\textit{Medium'}/\textit{Medium}]$$

Proving that the Design Refines the Specification

We shall prove that *Protocol'* $\Longrightarrow$ *Protocol*. For this purpose, we shall first examine and describe the phases, or "states", that *Protocol'* goes through during execution (there finitely many such). That is, we identify a set of formulae $\{S_0, \ldots, S_n\}$ such that *Protocol'* $\Longrightarrow S_0$ and such that for any $i \in \{0, \ldots, n\}$ it holds that $S_i \Longrightarrow [ACT](S_0 \vee \ldots \vee S_n)$, where $ACT \stackrel{def}{=} \{accept?, deliver!, \tau\}$. That is, "no matter what move is taken, we stay within the states $S_0, \ldots, S_n$". For each of these states S_i we shall in addition carefully select a transition property P_i such that $S_i \Longrightarrow P_i$. With this framework, presented in the first subsection, we shall be well prepared when we in the second subsection prove that *Protocol'* $\Longrightarrow$ *Protocol*.

States of Design

Before describing the states of *Protocol'* we shall first describe and name the states of respectively *Sender*, *Receiver* and *Medium*. Since each of these are sequential (not a parallel composition of several processes), this is just a matter of naming what remains after each action. Concerning the sender, we introduce the auxiliary name $Sender_1$ for the part of *Sender* that follows the action $accept?$:

$$Sender_1 \stackrel{def}{=} \nu\, S_1 \cdot inmed!; (ack?; Sender + err?; S_1)$$

Then the states of *Sender* are as follows (unfolding maximal fixpoints):

$$\begin{aligned} \textit{Accept?} &\stackrel{def}{=} accept?; \textit{Inmed!} \\ \textit{Inmed!} &\stackrel{def}{=} inmed!; \textit{Ack?\_Err?} \\ \textit{Ack?\_Err?} &\stackrel{def}{=} ack?; Sender + err?; Sender_1 \end{aligned}$$

The states of *Receiver* are:

$$\begin{aligned} \textit{Outmed?} &\stackrel{def}{=} outmed?; \textit{Deliver!} \\ \textit{Deliver!} &\stackrel{def}{=} deliver!; \textit{Ack!} \\ \textit{Ack!} &\stackrel{def}{=} ack!; Receiver \end{aligned}$$

Finally, the states of *Medium* are:

$$
\begin{aligned}
\mathit{Tau\_Inmed?} \;&\stackrel{def}{=}\; \circ\{inmed?\} \wedge \\
&\quad \Box([inmed?] \circ \{outmed!, err!\}) \wedge \\
&\quad \Box([outmed!, err!] \circ \{inmed?\})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Tau\_Outmed!\_Err!} \;&\stackrel{def}{=}\; \circ\{outmed!, err!\} \wedge \\
&\quad \Box([inmed?] \circ \{outmed!, err!\}) \wedge \\
&\quad \Box([outmed!, err!] \circ \{inmed?\})
\end{aligned}
$$

Note that the 'invariance' formulae ($\Box$) are the same in the two states of the medium, while the 'next' formula ($\circ\{\_\}$) is different.

We now compose these states of the individual processes into the states of *Protocol'*. It has the five states $S_0 - S_4$ shown below, where *Protocol'* $\Longrightarrow S_0$. We use the shorthand $\{I\}$ for $\{inmed\}\{outmed\}\{ack\}\{err\}$.

$$
\begin{aligned}
S_0 &\stackrel{def}{=} \{I\}\boldsymbol{forked}(\mathit{Outmed?}); \boldsymbol{forked}(\mathit{Tau\_Inmed?}); \boldsymbol{forked}(\mathit{Accept?}) \\
S_1 &\stackrel{def}{=} \{I\}\boldsymbol{forked}(\mathit{Outmed?}); \boldsymbol{forked}(\mathit{Tau\_Inmed?}); \boldsymbol{forked}(\mathit{Inmed!}) \\
S_2 &\stackrel{def}{=} \{I\}\boldsymbol{forked}(\mathit{Outmed?}); \boldsymbol{forked}(\mathit{Tau\_Outmed!\_Err!}); \boldsymbol{forked}(\mathit{Ack?\_Err?}) \\
S_3 &\stackrel{def}{=} \{I\}\boldsymbol{forked}(\mathit{Deliver!}); \boldsymbol{forked}(\mathit{Tau\_Inmed?}); \boldsymbol{forked}(\mathit{Ack?\_Err?}) \\
S_4 &\stackrel{def}{=} \{I\}\boldsymbol{forked}(\mathit{Ack!}); \boldsymbol{forked}(\mathit{Tau\_Inmed?}); \boldsymbol{forked}(\mathit{Ack?\_Err?})
\end{aligned}
$$

These states are related as illustrated by the transition diagram shown in figure 6.4, which we shall explain informally below. The transition diagram is a graphical presentation of the transition properties presented in the following lemma.

Lemma 6.30 (Transition Properties for *Protocol'*)

The states $S_0 - S_4$ satisfy the following transition properties:

$$
\begin{aligned}
S_0 &\Longrightarrow \ll\tau, accept?\gg \wedge [\tau]S_0 \wedge [accept?]S_1 \wedge \circ\{accept?\} \\
S_1 &\Longrightarrow \ll\tau\gg \wedge [\tau](S_1 \vee S_2) \\
S_2 &\Longrightarrow \ll\tau\gg \wedge [\tau](S_1 \vee S_2 \vee S_3) \\
S_3 &\Longrightarrow \ll\tau, deliver!\gg \wedge [\tau]S_3 \wedge [deliver!]S_4 \\
S_4 &\Longrightarrow \ll\tau\gg \wedge [\tau](S_0 \vee S_4) \wedge \odot S_0
\end{aligned}
$$

Proof

See appendix D.

■

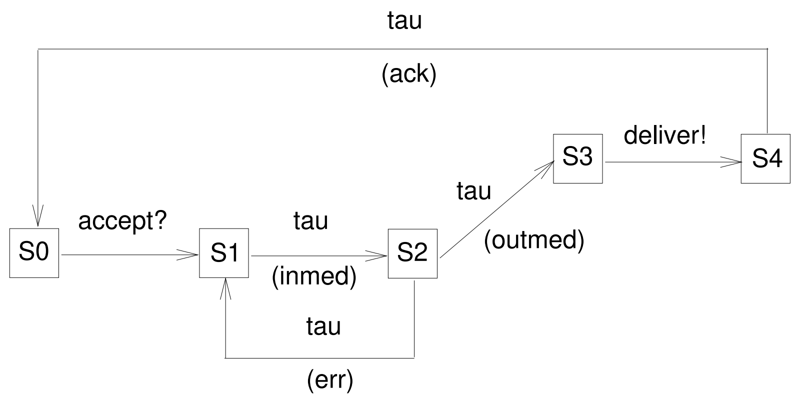

Figure 6.4: Transition Diagram for *Protocol'*

The following is an informal description of figure 6.4. In state S_0 the protocol accepts a message from the external world by an $accept?$-action, where after it enters state S_1. Here it passes the message to the medium by an $inmed$-action: the sender performs an $inmed!$-action and the medium performs an $inmed?$-action, together resulting in a τ-action. The protocol is now in state S_2, from which there are two possible routes. Either the medium looses the message, communicating an error message , $err!$, to the sender, resulting in a τ-action, and the protocol returns to state S_1. Or the medium correctly sends the message, $outmed!$, to the receiver, also resulting in a τ-action, and the protocol enters state S_3. From here the message is delivered to the outside world, $deliver!$, where after state S_4 is entered. In this state the sender is just waiting for an acknowledgement, $ack?$, from the receiver, and after this has been sent, resulting in a τ-action, state S_0 is entered again, and the protocol is ready to accept a new message.

In any of the states S_i ($i \in \{0, \ldots, 4\}$), there is a possibility of a τ-action that leaves us again in S_i. The boxes around the states represents this fact. These τ-actions come from the medium, which we recall is underspecified to allow at any time for a finite number of τ-actions before response. So also for each state, only finitely many such τ-actions can occur before another state is entered. We say that each state is convergent. In the transition properties it has only been necessary to record this convergence for S_0 and S_4 in terms of respectively $\circ\{accept?\}$ and $\odot\, S_0$. For the other states it is unimportant since the circle between S_2 and S_1 may break the convergence anyway: between $accept?$ and $deliver!$ the protocol may diverge.

Frequently it is needed to show that the protocol satisfies some maximal fixpoint: *Protocol'* $\Longrightarrow \nu\, X \cdot F(X)$, for some F. We will use the (**Maximal**) proof rule as follows. Let $S \stackrel{def}{=} S_0 \vee S_1 \vee S_2 \vee S_3 \vee S_4$. Show first that: $S \Longrightarrow F(S)$. Then apply (**Maximal**) to obtain: $S \Longrightarrow \nu\, X \cdot F(X)$. Finally since *Protocol'* $\Longrightarrow S_0 \Longrightarrow S$ we get that: *Protocol'* $\Longrightarrow \nu\, X \cdot F(X)$

Properties of the Design

We shall prove the following theorem:

Theorem 6.31 (Design refines Specification)

$$\textit{Protocol'} \Longrightarrow \textit{Protocol}$$

Proof

We shall show that:

$$\begin{aligned} \textit{Protocol'} \;\Longrightarrow\; & \circ\{accept?\} \wedge \\ & \Box([accept?] \circ_w \{deliver!\}) \wedge \\ & \Box([deliver!] \circ \{accept?\}) \end{aligned}$$

We deal with each of the three conjuncts separately.

$\underline{\textit{Protocol'} \Longrightarrow \circ\{accept?\}}$

This case is easy, as it follows directly from the transition properties in lemma (6.30): $\textit{Protocol'} \Longrightarrow S_0 \Longrightarrow \circ\{accept?\}$.

$\underline{\textit{Protocol'} \Longrightarrow \Box([accept?] \circ_w \{deliver!\})}$

The property to be proved stands for the following maximal fixpoint:

$$\Box([accept?] \circ_w \{deliver!\}) = \nu\, X \cdot \underbrace{[accept?] \circ_w \{deliver!\} \;\wedge\; [ACT]X}_{F(X)}$$

To show that $\textit{Protocol'} \Longrightarrow \nu\, X \cdot F(X)$, we show first $S \Longrightarrow \nu\, X \cdot F(X)$, where $S = S_0 \vee S_1 \vee S_2 \vee S_3 \vee S_4$, and then since $\textit{Protocol'} \Longrightarrow S$ we have the result. To show that $S \Longrightarrow \nu\, X \cdot F(X)$, we show that $S \Longrightarrow F(S)$, and then apply the (**Maximal**) proof rule. So we show that:

$$\begin{aligned} &(S_0 \;\vee\; S_1 \;\vee\; S_2 \;\vee\; S_3 \;\vee\; S_4) \Longrightarrow \\ &\quad [accept?] \circ_w \{deliver!\} \;\wedge\; [ACT](S_0 \;\vee\; S_1 \;\vee\; S_2 \;\vee\; S_3 \;\vee\; S_4) \end{aligned}$$

This reduces to showing the following two properties for any $i \in \{0, \ldots, 4\}$:

1. $S_i \Longrightarrow [accept?] \circ_w \{deliver!\}$

2. $S_i \Longrightarrow [ACT](S_0 \vee S_1 \vee S_2 \vee S_3 \vee S_4)$

The properties in group (2) follow immediately from the transition properties in lemma (6.30). Concerning the properties in group (1), we have, again due to lemma (6.30), that for all $i \neq 0$: $S_i \Longrightarrow [accept?]\boldsymbol{ff}$ and thereby that $S_i \Longrightarrow [accept?] \circ_w \{deliver!\}$. The last deduction follows from the fact that $\boldsymbol{ff}$ refines any formula (**Empty**), and then from applying the (**Monotone**) proof rule.

Concerning the proof of $S_0 \Longrightarrow [accept?]\circ_w\{deliver!\}$ we proceed as follows. From lemma (6.30) we have that $S_0 \Longrightarrow [accept?]S_1$. So obviously $S_0 \Longrightarrow [accept?] \circ_w \{deliver!\}$ if $S_1 \Longrightarrow \circ_w\{deliver!\}$. The formula $\circ_w\{deliver!\}$ stands for a maximal fixpoint:

$$\circ_w\{deliver!\} = \nu\, X \cdot \ll deliver!, \tau \gg \wedge [\tau]X$$

Since $S_1 \Longrightarrow S_1 \vee S_2 \vee S_3$ we have succeeded if we can prove that: $(S_1 \vee S_2 \vee S_3) \Longrightarrow \nu\, X \cdot \ll deliver!, \tau \gg \wedge [\tau]X$. We can prove this by proving that:

$$(S_1 \vee S_2 \vee S_3) \Longrightarrow \ll deliver!, \tau \gg \wedge [\tau](S_1 \vee S_2 \vee S_3)$$

We prove each of the cases:

$$\begin{array}{lcl} S_1 & \Longrightarrow & \ll deliver!, \tau \gg \wedge [\tau](S_1 \vee S_2 \vee S_3) \\ S_2 & \Longrightarrow & \ll deliver!, \tau \gg \wedge [\tau](S_1 \vee S_2 \vee S_3) \\ S_3 & \Longrightarrow & \ll deliver!, \tau \gg \wedge [\tau](S_1 \vee S_2 \vee S_3) \end{array}$$

This will hold since by lemma (6.30) we have that:

$$\begin{array}{lcl} S_1 & \Longrightarrow & \ll \tau \gg \wedge [\tau](S_1 \vee S_2) \\ S_2 & \Longrightarrow & \ll \tau \gg \wedge [\tau](S_1 \vee S_2 \vee S_3) \\ S_3 & \Longrightarrow & \ll \tau, deliver! \gg \wedge [\tau]S_3 \end{array}$$

Note that $\ll \tau \gg \Longrightarrow \ll deliver!, \tau \gg$.

Protocol' $\Longrightarrow \Box([deliver!] \circ \{accept?\})$

Note that the property we want to prove stands for the following maximal fixpoint:

$$\Box([deliver!] \circ \{accept?\}) = \nu\, X \cdot [deliver!] \circ \{accept?\} \wedge [ACT]X$$

We proceed as in the previous case, thus leaving us with the following proof obligation for any $i \in \{0, \ldots, 4\}$:

$$S_i \Longrightarrow [deliver!] \circ \{accept?\}$$

We have due to lemma (6.30) that for all $i \neq 3$: $S_i \Longrightarrow [deliver!]\boldsymbol{ff}$ and thereby that $S_i \Longrightarrow [deliver!] \circ \{accept?\}$.

Concerning the proof of $S_3 \Longrightarrow [deliver!] \circ \{accept?\}$ we proceed as follows. From lemma (6.30) we have that $S_3 \Longrightarrow [deliver!]S_4$, so we are left with proving that $S_4 \Longrightarrow \circ\{accept?\}$. Now from lemma (6.30) we have that $S_4 \Longrightarrow \odot S_0$, so we are done if we can show that $\odot S_0 \Longrightarrow \circ\{accept?\}$. This we show by first noting that $\odot S_0$ stands for a minimal fixpoint:

$$\odot S_0 = \mu X \cdot \underbrace{S_0 \vee (\ll\tau\gg \wedge [\tau]X)}_{G(X)}$$

Due to the (**Minimal**) proof rule, is suffices to prove: $G(\circ\{accept?\}) \Longrightarrow \circ\{accept?\}$. That is:

$$S_0 \vee (\ll\tau\gg \wedge [\tau] \circ \{accept?\}) \Longrightarrow \circ\{accept?\}$$

This amounts to showing that:

1. $S_0 \Longrightarrow \circ\{accept?\}$
2. $\ll\tau\gg \wedge [\tau] \circ \{accept?\} \Longrightarrow \circ\{accept?\}$

Property (1) follows directly from lemma (6.30). The proof of (2) proceeds as follows.

$$\begin{array}{lll} & \ll\tau\gg \wedge [\tau] \circ \{accept?\} & \\ \Longrightarrow & \ll\tau, accept?\gg \wedge [\tau] \circ \{accept?\} & \text{(straight forward)} \\ \Longrightarrow & \circ\{accept?\} & (\textbf{UnfoldMin}) \end{array}$$

■

Proving that the Program Refines the Design

We shall prove that *Protocol"* $\Longrightarrow$ *Protocol'*. Recall that *Protocol"* only differs from *Protocol'* in that *Medium'* has been substituted for *Medium*. Due to the compositionality of the proof system via the (**Monotone**) proof rule, it then suffices to show that *Medium'* $\Longrightarrow$ *Medium*, which is done in the remainder of this section. We proceed as before by first identifying the states that the programmed medium goes through during execution.

States of the Programmed Medium

Let us first name the possible states of each of the components of the programmed medium. The states of the recovery system, *Recovery*, are as follows:

$$\begin{aligned} \mathit{Crash?} &\stackrel{def}{=} \mathit{crash}?; \mathit{Err!} \\ \mathit{Err!} &\stackrel{def}{=} \mathit{err}!; \boldsymbol{forked}(\mathit{Unstable}); \mathit{Recovery} \end{aligned}$$

The states of the unstable medium, *Unstable*, are as follows:

$$\begin{aligned} \mathit{Inmed?} &\stackrel{def}{=} \mathit{inmed}?; \mathit{Outmed!\_Err!\_Crash!} \\ \mathit{Outmed!\_Err!\_Crash!} &\stackrel{def}{=} \mathit{outmed}!; \mathit{Unstable} + \mathit{err}!; \mathit{Unstable} + \mathit{crash}!; \boldsymbol{nil} \end{aligned}$$

The states of the individual processes above are composed into the states of *Medium'*. It has three states M_0, M_1, M_2 where *Medium'* $\Longrightarrow M_0$:

$$\begin{aligned} M_0 &= \{\mathit{crash}\}\boldsymbol{forked}(\mathit{Inmed?}); \mathit{Crash?} \\ M_1 &= \{\mathit{crash}\}\boldsymbol{forked}(\mathit{Outmed!\_Err!\_Crash!}); \mathit{Crash?} \\ M_2 &= \{\mathit{crash}\}\mathit{Err!} \end{aligned}$$

These states are related as illustrated by the transition diagram shown in figure 6.5. The transition diagram is a graphical presentation of the transition properties presented in the following lemma.

Lemma 6.32 (Transition Properties for *Medium'*)

The states M_0, M_1, M_2 satisfy the following transition properties:

$$\begin{aligned} M_0 &\Longrightarrow \ll \mathit{inmed}? \gg \wedge [\mathit{inmed}?]M_1 \\ M_1 &\Longrightarrow \ll \mathit{outmed}!, \mathit{err}!, \tau \gg \wedge [\tau]M_2 \wedge [\mathit{outmed}!, \mathit{err}!]M_0 \\ M_2 &\Longrightarrow \ll \mathit{err}! \gg \wedge [\mathit{err}!]M_0 \end{aligned}$$

Proof

See appendix D.

■

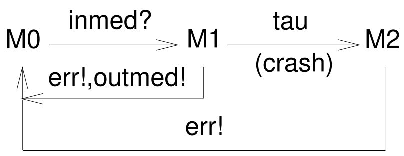

Figure 6.5: Transition Diagram for *Medium'*

The following is an informal description of figure 6.5. In state M_0 the medium receives a message (from the sender) by an *inmed*?-action, where after it enters state M_1. Here it can loose the message which is signaled by the $err!$-action, or the message is transferred (to the receiver) by an $outmed!$-action. In both cases the unreliable medium returns to its initial state M_0. A third possibility is that the unreliable medium crashes, sending a $crash!$ signal to the recovery system. The recovery system sends an error message, $err!$ (to the sender, telling that the message is lost), and finally starts a new unreliable medium: M_0 is entered again.

Properties of the Programmed Medium

We shall prove the following theorem:

Theorem 6.33

$$Medium' \Longrightarrow Medium$$

Proof

We shall show that:

$$\begin{array}{rcl} Medium' & \Longrightarrow & \circ\{inmed?\} \wedge \\ & & \Box([inmed?] \circ \{outmed!, err!\}) \wedge \\ & & \Box([outmed!, err!] \circ \{inmed?\}) \end{array}$$

We deal with each of the three conjuncts separately.
Let $ACT \stackrel{def}{=} \{inmed?, outmed!, err!, \tau\}$.

$\underline{Medium' \Longrightarrow \circ\{inmed?\}}$

We can unfold the property:

$$\circ\{inmed?\} = \mu\, X \cdot \ll inmed?, \tau \gg \ \wedge [\tau]X \Longleftrightarrow \ll inmed?, \tau \gg \ \wedge [\tau] \circ \{inmed?\}$$

We proceed as follows:

$$\begin{array}{lll} & M_0 & \\ \Longrightarrow & \ll inmed? \gg & \text{(lemma (6.32))} \\ = & < inmed? > \boldsymbol{tt} \wedge [ACT - \{inmed?\}]\boldsymbol{ff} & \\ \Longrightarrow & < inmed?, \tau > \boldsymbol{tt} \wedge [ACT - \{inmed?, \tau\}]\boldsymbol{ff} \wedge [\tau]\boldsymbol{ff} & \text{(straight forward)} \\ = & \ll inmed?, \tau \gg \ \wedge [\tau]\boldsymbol{ff} & \\ \Longrightarrow & \ll inmed?, \tau \gg \ \wedge [\tau] \circ \{inmed?\} & (\textbf{Empty}) \\ \Longrightarrow & \circ\{inmed?\} & (\textbf{UnfoldMin}) \end{array}$$

<u>*Medium'* $\Longrightarrow \Box([inmed?] \circ \{outmed!, err!\})$</u>

The property to be proved stands for the following maximal fixpoint:

$$\Box([inmed?] \circ \{outmed!, err!\}) = \nu\, X \cdot [inmed?] \circ \{outmed!, err!\} \;\wedge\; [ACT]X$$

We proceed as in the previous maximal fixpoint cases, using the (**Maximal**) proof rule, thus leaving us to show that:

$$(M_0 \;\vee\; M_1 \;\vee\; M_2) \Longrightarrow [inmed?] \circ \{outmed!, err!\} \;\wedge\; [ACT](M_0 \;\vee\; M_1 \;\vee\; M_2)$$

Due to lemma (6.32) we can concentrate on showing that for any $i \in \{0, 1, 2\}$:

$$M_i \Longrightarrow [inmed?] \circ \{outmed!, err!\}$$

We have that for $i \in \{1, 2\}$: $M_i \Longrightarrow [inmed?]\boldsymbol{ff}$ and thereby that $M_i \Longrightarrow [inmed?] \circ \{outmed!, err!\}$.

Concerning the proof of $M_0 \Longrightarrow [inmed?] \circ \{outmed!, err!\}$ we proceed as follows. From the transition properties we have that $M_0 \Longrightarrow [inmed?]M_1$. So obviously $M_0 \Longrightarrow [inmed?] \circ \{outmed!, err!\}$ if $M_1 \Longrightarrow \circ\{outmed!, err!\}$. The formula $\circ\{outmed!, err!\}$ stands for a minimal fixpoint:

$$\circ\{outmed!, err!\} = \mu\, X \cdot \ll outmed!, err!, \tau \gg \;\wedge\; [\tau]X$$

We proceed as follows:

$$\begin{array}{lll}
 & M_1 & \\
\Longrightarrow & \ll outmed!, err!, \tau \gg \;\wedge\; [\tau]M_2 & \text{(lemma (6.32))} \\
\Longrightarrow & \ll outmed!, err!, \tau \gg \;\wedge\; [\tau] \circ \{outmed!, err!\} & (M_2 \Longrightarrow \circ\{outmed!, err!\}, \text{ see below}) \\
\Longrightarrow & \circ\{outmed!, err!\} & (\textbf{UnfoldMin})
\end{array}$$

The above refinement depends on the truth of $M_2 \Longrightarrow \circ\{outmed!, err!\}$, so this we prove:

$$\begin{array}{lll}
 & M_2 & \\
\Longrightarrow & \ll err! \gg \;\wedge\; [err!]M_0 & \text{(lemma (6.32))} \\
\Longrightarrow & \ll err! \gg & \\
= & \langle err! \rangle\, \boldsymbol{tt} \;\wedge\; [ACT - \{err!\}]\boldsymbol{ff} & \\
\Longrightarrow & \langle outmed!, err!, \tau \rangle\, \boldsymbol{tt} \;\wedge\; [ACT - \{outmed!, err!, \tau\}]\boldsymbol{ff} \;\wedge\; [\tau]\boldsymbol{ff} & \\
= & \ll outmed!, err!, \tau \gg \;\wedge\; [\tau]\boldsymbol{ff} & \\
\Longrightarrow & \ll outmed!, err!, \tau \gg \;\wedge\; [\tau] \circ \{outmed!, err!\} & \\
\Longrightarrow & \circ\{outmed!, err!\} & (\textbf{UnfoldMin})
\end{array}$$

<u>*Medium'* $\Longrightarrow \Box([outmed!, err!] \circ \{inmed?\})$</u>

The property to be proved stands for the following maximal fixpoint:

$$\Box([outmed!, err!] \circ \{inmed?\}) = \nu\, X \cdot [outmed!, err!] \circ \{inmed?\} \;\wedge\; [ACT]X$$

We proceed as in the previous section, thus leaving us with the following proof obligation for any $i \in \{0, 1, 2\}$:

$$M_i \Longrightarrow [outmed!, err!] \circ \{inmed?\}$$

We have that $M_0 \Longrightarrow [outmed!, err!]\boldsymbol{ff}$ and thereby that $M_0 \Longrightarrow [outmed!, err!] \circ \{inmed?\}$.

Concerning the proof of $M_1 \Longrightarrow [outmed!, err!] \circ \{inmed?\}$ we proceed as follows. From lemma (6.32) we have that $M_1 \Longrightarrow [outmed!, err!]M_0$, and from earlier we have that $M_0 \Longrightarrow \circ\{inmed?\}$, and so we are done.

Concerning the proof of $M_2 \Longrightarrow [outmed!, err!] \circ \{inmed?\}$ we proceed as follows:

$$\begin{array}{lll}
 & M2 & \\
\Longrightarrow & \ll err! \gg \;\wedge\, [err!]M_0 & \text{(lemma (6.32))} \\
= & <err!>\boldsymbol{tt} \;\wedge\; [ACT - \{err!\}]\boldsymbol{ff} \;\wedge\; [err!]M_0 & \\
\Longrightarrow & [outmed!]\boldsymbol{ff} \;\wedge\; [err!]M_0 & \\
\Longrightarrow & [outmed!]M_0 \;\wedge\; [err!]M_0 & (\textbf{Empty}) \\
= & [outmed!, err!]M_0 & \\
\Longrightarrow & [outmed!, err!] \circ \{inmed?\} & (M_0 \Longrightarrow \circ\{inmed?\}, \text{ proved earlier})
\end{array}$$

∎

6.2.7 Concluding Remarks and Related Work

Other attempts have been made to define refinement logics for process calculi. In [Hol89] as well as in [GS86] such logics have been defined for variants of CCS. In the case of [GS86] there are though no operators for parallel composition and channel restriction, since the important operator is regarded to be non-deterministic choice.

Disregarding the different process calculi (FC versus CCS-variants), there are some differences in the logics in the present work and in the two papers above. To begin with, there is no negation operator in [Hol89], while in the present work and in [GS86] there is negation. Note, however, that we have not given rules for negation, whereas in [GS86] there are equational rules. Since we have regarded it too troublesome to give (non-equational) rules for negation, we have defined derived forms of restricted negation, and given rules just for these forms. So our proof system is closer to [Hol89] where all these derived forms are basic (non-derived) constructs.

In [Hol89] there are operators for action prefixing $(\alpha.\psi)$ as well action possibility $<\!\alpha\!>\psi$. This is in principle also our case, though we have sequential composition instead of action prefixing. In [GS86] these two constructs have been merged into a single construct on the form $A.\psi$ where $A \subseteq ACT$ is a set of actions, and the construct is satisfied by a process p, if $p \xrightarrow{\alpha} \ldots$ for some $\alpha \in A$ and, whenever $p \xrightarrow{\alpha} p'$, then $\alpha \in A$ and p' satisfies ψ. We have chosen to distinguish between the constructs since we prefer to investigate a pure combination of FC_r and traditional Hennessy-Milner Logic.

Finally, concerning fixpoints, in [Hol89] there is no minimal fixpoint operator, while this is present in our work and in [GS86]. Concerning maximal fixpoint operators [Hol89] differs notably due to the general definition of the satisfaction relation between processes and formulae: in [Hol89] the structure of a formula – what concerns programming constructs – must be maintained by any program satisfying the formula. This means that for any programming construct Op, a program p satisfies a specification $Op(\psi_1, \ldots, \psi_n)$ iff p 'has the form' $Op(p_1, \ldots, p_n)$ where each p_i satisfies ψ_i. This is restricted compared to the concept of satisfaction that we present: recall that we just require that $p \equiv Op(p_1, \ldots, p_n)$. So while programming constructs in [Hol89] specifies internal structure, we just use them to specify behaviour. We follow [GS86] in this respect.

This structure preserving semantics in [Hol89] seems too restrictive in our opinion since writing abstract programs appears to be a concise way of specifying concrete programs that has a different structure. If, however, problem decomposition is the only purpose, then his logic is satisfactory, since problem decomposition requires structure to be kept.

The two approaches can be formulated as two slogans, the first one being the one we defend here: "programming constructs are for specification as well as for implementation" versus "programming constructs are for implementation only". Figure 6.6 illustrates that we allow more programs to satisfy a specification.

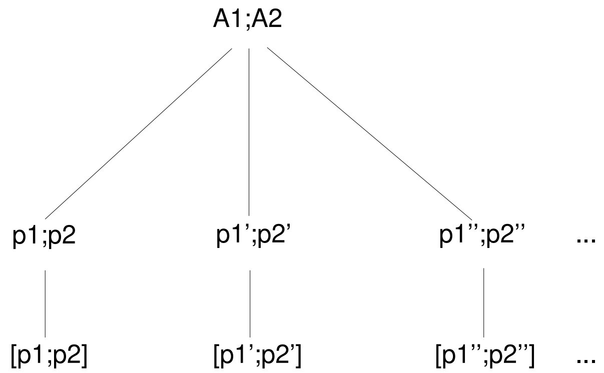

Figure 6.6: Structure versus equivalence preservation

The programs `p1;p2`, `p1';p2'`, `p1'';p2''`, ... are the programs that satisfy the specification `A1;A2` if we follow [Hol89]. For each such program `p1;p2` we additionally allow all the programs in the equivalence class `[p1;p2]` to satisfy the specification.

Another problem in [Hol89] is that one can generally not use maximal fixpoint formulae to write recursive programs of the form $\nu X \cdot p$ where p is a program. The problem is, that such maximal fixpoints denote the empty set (when p contains X). For example $\nu X \cdot \alpha.X$ denotes the empty set. To see this, observe for example that $\nu X \cdot \alpha.X$ can be unfolded into the equivalent formula $\alpha.(\nu X \cdot \alpha.X)$; equivalent in the sense that the two formulae denote the same sets of processes. This means for example that the program $\boldsymbol{fix}\, x \cdot \alpha.x$ does not satisfy the specification since it has not the form $\alpha.q$ for some q. The solution in [Hol89] is to introduce a $\boldsymbol{fix}\, x \cdot p$ construct in the logic in addition to the maximal fixpoint construct, where p must be a program. $\boldsymbol{fix}\, x \cdot p$ denotes the singleton process set $\{\boldsymbol{fix}\, x \cdot p\}$. In our logic (as in [GS86]) maximal fixpoints cover the pure programming case also.

In addition to having two kinds of "maximal" fixpoints, a second consequence of the structure preserving semantics in [Hol89] is , that one cannot always mix programming and loose specification within recursion as one would like, as for example in $\nu\ X \cdot \alpha.[\beta]X$. No infinitely behaving process satisfies this formula, although we would expect the program $\boldsymbol{fix}\, x \cdot \alpha.\beta.x$ to satisfy the formula. Many finite processes satisfy the formula: $\alpha.\boldsymbol{nil}$, $\alpha.\beta.\alpha.\boldsymbol{nil}$, $\alpha.\gamma.\boldsymbol{nil}$, $\ldots$.

Chapter 7

A Calculus of Mobile Processes

In this chapter an extension of the Fork Calculus is presented, which allows processes to change their configuration dynamically. By configuration we mean the set of channels that connect the process with its context. This is obtained by allowing processes to communicate channels over channels, also channels that are initially local, and which thereby become global (visible in the context). The π-calculus [MPW89a, MPW89b, MPW91] is such an extension of CCS, and is a basis for the work presented here. Another basis is the work on Facile [PGM90, Pra91] where dynamic reconfiguration is examined in the Facile language.

We first present the π-calculus in section 7.1 in order to motivate the design of the FC extension. This presentation is relatively detailed compared to the fact that we will not make a formal comparison with our calculus. On the other hand, our calculus can best be motivated (be it informally) on the basis of a knowledge of the π-calculus.

Section 7.2 presents the syntax and informal semantics of the calculus. Section 7.3 defines its formal semantics, including a definition of equivalence. Section 7.4 discusses alternative definitions of equivalence. Finally, section 7.5 defines a logic for the new calculus. The logic will make it possible to specify properties of programs that create processes with the ***fork***-operator, and which dynamically change configuration by communicating channels over channels. We shall, in contrast to earlier chapters, not give any congruence theorems, nor provide an axiomatisation, nor prove any other theorems – "just" semantics and logic.

7.1 The π-calculus

Our presentation is mainly based on [MPW91]. The presentation ought to be self-contained, at least at a formal level. The section is not essential for the understanding of the later extension of the Fork Calculus – it is rather a motivation.

Assume an infinite set $\mathcal{N}$ of names and let u, v, w, x, y, z range over names. Note that there are only names, hence there is no distinction between for example constants and variables. The syntax for agents (or processes using our terminology), defining the language $\mathcal{L}$, is as follows.

$$p \quad ::= \quad \boldsymbol{nil} \mid \tau.p \mid x!y.p \mid x(y).p \mid p_1 + p_2 \mid p_1|p_2 \mid (y)p \mid [x = y]p$$

The processes $\boldsymbol{nil}$, $\tau.p$, $p_1 + p_2$ and $p_1|p_2$ emulate those of CCS. That is, they represent the inactivity, silent prefix, choice, and parallel composition. The process $x!y.p$ outputs the name y on x and then behaves like p. The process $x(y).p$ inputs an arbitrary name, say z, on x, and then behaves like $p\{z/y\}$. The process $(y)p$, corresponding to restriction $p\backslash y$ in CCS, behaves like p except that communications over the name y are prohibited (but communications between components of p along the name y are not prohibited, since they become τ actions. Finally, the process $[x = y]p$, called a *match*, behaves like p if the names x and y are identical, and otherwise like $\boldsymbol{nil}$. Concerning the order of precedence among the operators it is enough here to say that Parallel Composition and Choice bind weakest. Note that recursion (and *replication* which is a variant of recursion found in some papers on the π-calculus) is not included as a primitive in this presentation. Recursion is not essential to the problems in hand.

Note that the name y is bound in the two agents $x(y).p$ and $(y)p$. That is, a name with parenthesis around stands for a binding occurrence, just like x is bound in the lambda abstraction $\lambda x.e$. We write $fn(p)$ for the set of names occurring free in p. For example $fn(x(y).p) = \{x\} \cup fn(p) - \{y\}$.

A labelled transition system $(\mathcal{L}, Act, \longrightarrow)$ is defined as follows. The set Act contains the actions α presented in figure 7.1.

α	*kind*	$fn(\alpha)$	$bn(\alpha)$
τ	silent	$\emptyset$	$\emptyset$
$x!y$	free output	$\{x, y\}$	$\emptyset$
$x!(y)$	bound output	$\{x\}$	$\{y\}$
$x(y)$	bound input	$\{x\}$	$\{y\}$

Figure 7.1: The actions

Note that these actions are the labels on semantic transitions, and not the actions occurring in a program text. The actions are divided into *free* respectively *bound* actions, depending on whether they contain free names only, that must be known by the context, or whether they also contain bound names as explained below.

The *silent* action τ corresponds to an internal computation. The *free* output action $x!y$ corresponds to the transmission of the free name y along the free name x. The *bound* output action $x!(y)$ means that a local name designated by y is exported along the free x. The brackets around y means a binding occurrence, so y is not a particular name, but represents *some* name, it is bound. The bound input action $x(y)$ means that any name can be received along x, and (y) designates the places where the received name will go. The bound parameter will be instantiated at a later stage. In figure 7.1 it is shown for each action α, what the free names $fn(\alpha)$ and bound names $bn(\alpha)$ are. We write $n(\alpha)$ for $fn(\alpha) \cup bn(\alpha)$.

A structural congruence $\equiv$ is defined on agents. Its definition depends on the standard notion of alpha-equivalence: two agents p and q are alpha-equivalent, written $p \equiv_\alpha q$, if p and q only differ in the choice of bound names. Now $\equiv$ is the least congruence satisfying the following clauses:

$$\begin{array}{rcll} p & \equiv & q & \quad \textit{provided } p \equiv_\alpha q \\ p|q & \equiv & q|p & \\ p+q & \equiv & q+p & \\ [x=x]p & \equiv & p & \end{array}$$

A *variant* of the transition $p \xrightarrow{\alpha} q$ is a transition which only differs in that p and q have been replaced by structurally congruent agents, and α has been alpha-converted, where a name bound in α includes q in its scope.

Below we give the rules defining the transition relation. In each rule, the transition in the conclusion stands for all variants of the transition. As is stated in [MPW91]: "The use of variants and structural congruence makes it possible to formulate the rules more concisely than would otherwise be possible".

$$\textbf{(Action)} \qquad \frac{}{\alpha.p \xrightarrow{\alpha} p}$$

$$\textbf{(Sum)} \qquad \frac{p \xrightarrow{\alpha} p'}{p+q \xrightarrow{\alpha} p'}$$

$$\textbf{(Parallel)} \qquad \frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q}\ bn(\alpha) \cap fn(q) = \emptyset$$

$$\textbf{(Communication)} \qquad \frac{p \xrightarrow{x!y} p' \;,\; q \xrightarrow{x(z)} q'}{p|q \xrightarrow{\tau} p'|q'[y/z]}$$

$$\textbf{(Restriction)} \qquad \frac{p \xrightarrow{\alpha} p'}{(y)p \xrightarrow{\alpha} (y)p'}\ y \notin n(\alpha)$$

$$(\textbf{Open}) \qquad \frac{p \xrightarrow{x!y} p'}{(y)p \xrightarrow{x!(y)} p'}\ y \neq x$$

$$(\textbf{Close}) \quad \frac{p \xrightarrow{x!(y)} p'\ ,\ q \xrightarrow{x(y)} q'}{p|q \xrightarrow{\tau} (y)(p'|q')}$$

The name bound by an input prefix form $x(y).p$ becomes instantiated in (**Communication**) and in (**Close**) when a communication between two agents is inferred. The rules (**Open**) and (**Close**) describe how *scope extrusion* is inferred: an agent communicates a local name to another agent, where after that name is know to both agents. A special rule for matching is unnecessary because of the principles of structural congruence and variants.

The semantics is said to define *late input instantiation* in the sense that the rule (**Action**) for the input case, where $\alpha = x(y)$, does not fix the input value as is the case in CCS. This is forced due to the combination of restriction and the possibility of communicating names. Recall the value-passing CCS rule for input, which defines *early input instantiation* (see [Mil89] for a translation of value-passing CCS to basic CCS without value passing):

$$\frac{}{a(y).p \xrightarrow{av} p[v/y]}$$

In the π-calculus the instantiation happens in the rule for communication. The semantics gets slightly un-operational because of this (note for example the use of structural rules). In the approach that we shall apply, and which is the approach taken in [Pra91], new channels are generated instead of operating with textual scope manipulations. We can therefore apply *early* input instantiation. In [Pra91] it is stated: “The uniqueness of generated channels prohibits unforeseen complications arising on exporting formerly hidden channels such as two different channels having identical values”.

In [BB90], a semantics is presented for the π-calculus using the *Chemical Abstract Machine*. This definition is very elegant, and is based on the same idea of forgetting about α-conversion and scope extension by generating new unique names using a name server. They, however, give semantics to a subset of the π-calculus not involving the choice operator, and other examples in [BB90] illustrate that the Chemical Abstract Machine is not very suited for giving semantics to the choice operator as found in CCS and the π-calculus. The authors in [BB90] use this as an argument against the choice operator.

As an example of how to use the semantic rules, consider the following parallel composition of two agents p and q:

$$\underbrace{(x)y!x.p'}_{p} \mid \underbrace{y(z).q'}_{q}$$

The example illustrates the important concept of scope extrusion, which is the fundamental problem solved in the π-calculus. The process p has a private name x, which it wants to communicate to q via the name y. After the communication, yielding a τ transition, the name x is known to (the remainders of) both p and q. This is represented by the fact that the scope of x is extended to cover (the remainders of) both p and q. Hence, as in CCS, textual scope manipulation is the way to deal with locality of channels. We shall for the purpose of illustration complicate the example by assuming that the name x occurs free in q. This namely gives rise to an alpha-conversion of the x in p into x' as illustrated by the following inferences:

$$\begin{array}{lll}
(1) & y!x.p' \xrightarrow{y!x} p' & (\textbf{Action}) \\
(2) & (x)y!x.p' \xrightarrow{y!(x)} p' & (1), (\textbf{Open}) \\
(3) & (x)y!x.p' \xrightarrow{y!(x')} p'[x'/x] & (2), \text{variant principle} \\
(4) & y(z).q' \xrightarrow{y(z)} q' & (\textbf{Action}) \\
(5) & y(z).q' \xrightarrow{y(x')} q'[x'/z] & (4), \text{variant principle} \\
(6) & (x)y!x.p'|y(z).q' \xrightarrow{\tau} (x')(p'[x'/x]|q'[x'/z]) & (3,5), (\textbf{Close})
\end{array}$$

In [MPW91] two different notions of equivalence are defined, called respectively *late* and *early bisimulation.* We give the two definitions below.

Definition 7.1 (Late bisimulation) *A binary relation S on agents is a* late simulation *if $(p,q) \in S$ implies that*

1. *If $p \xrightarrow{\alpha} p'$ and α is τ, $x!z$ or $x!(y)$ with $y \notin fn(p) \cup fn(q)$, then for some q', $q \xrightarrow{\alpha} q'$ and $(p',q') \in S$.*

2. *If $p \xrightarrow{x(y)} p'$ and $y \notin fn(p) \cup fn(q)$, then for some q', $q \xrightarrow{x(y)} q'$, and for all w, $(p'[w/y], q'[w/y]) \in S$.*

The relation S is a late bisimulation *if both S and S^{-1} are late simulations. We define $p \dot{\sim}_L q$ to mean that $(p,q) \in S$ for some late bisimulation S.*

Note the quantifications over q' and w in clause (2). They are on the form $\exists q' \forall w$. Early bisimulation is obtained by commuting the quantifiers into $\forall w \exists q'$ as in the following definition, where clause (1) is as before.

Definition 7.2 (Early bisimulation) *A binary relation S on agents is an* early simulation *is $(p, q) \in S$ implies that*

1. *If $p \xrightarrow{\alpha} p'$ and α is τ, $x!z$ or $x!(y)$ with $y \notin fn(p) \cup fn(q)$, then for some q', $q \xrightarrow{\alpha} q'$ and $(p', q') \in S$.*
2. *If $p \xrightarrow{x(y)} p'$ and $y \notin fn(p) \cup fn(q)$, then for all w, there is a q' such that $q \xrightarrow{x(y)} q'$ and $(p'[w/y], q'[w/y]) \in S$.*

The relation S is an early bisimulation *if both S and S^{-1} are early simulations. We define $p \dot{\sim}_E q$ to mean that $(p, q) \in S$ for some early bisimulation S.*

It can be seen that $\dot{\sim}_L \subseteq \dot{\sim}_E$. To see that this inclusion is strict, consider the following example:

$$\begin{array}{lcl} p & \stackrel{def}{=} & x(u).\tau.\mathbf{nil} + x(u).\mathbf{nil} \\ q & \stackrel{def}{=} & p + x(u).[u = z]\tau.\mathbf{nil} \end{array}$$

Then $p \dot{\sim}_E q$, but not $p \dot{\sim}_L q$. This can be seen by observing that q is an extension of p and that therefore the distinguishing transition is when q performs the move $q \xrightarrow{x(u)} [u = z]\tau.\mathbf{nil}$, and we have to find a corresponding transition of p. Then it matters what is the moment of quantification over w in clause (2) in the two definitions of simulation. Note that p can perform the following two transitions: $p \xrightarrow{x(u)} \tau.\mathbf{nil}$ and $p \xrightarrow{x(u)} \mathbf{nil}$. For example, it does not hold that for all w, $(\tau.\mathbf{nil})[w/u] \dot{\sim}_L ([u = z]\tau.\mathbf{nil})[w/u]$ – take any $w \neq z$.

The notion of early bisimulation is close to the notion of bisimulation in CCS. However, in [MPW89b] the late bisimulation is chosen as the basis for an axiomatisation since, as it is stated: "the algebraic theory of $\dot{\sim}_E$ is complicated ...". In other words: the axiom system gets simpler for late bisimulation. In the calculus that we shall provide, our semantics will naturally lead to early bisimulation. One can also argue that early bisimulation perhaps is closer to what we from an intuitive point of view would find correct: the two processes p and q above appear to behave equally from an observer's point of view. We restain from giving an axiomatisation.

Neither $\dot{\sim}_L$, nor $\dot{\sim}_E$ are congruences due to the input construct. This can be seen from the fact that for example $x!y.\mathbf{nil}|v(w).\mathbf{nil} \dot{\sim}_E x!y.v(w).\mathbf{nil}+v(w).x!y.\mathbf{nil}$ but it does not hold that $z(v)(x!y.\mathbf{nil}|v(w).\mathbf{nil}) \dot{\sim}_E z(v)(x!y.v(w).\mathbf{nil}+v(w).x!y.\mathbf{nil})$. Suppose namely that any of the two agents are put in parallel with $z!x$. The first process, now $x!y.\mathbf{nil}|x(w).\mathbf{nil}$, is then capable of performing a τ-transition caused by a communication of y on x. The other process, now $x!y.x(w).\mathbf{nil}+x(w).x!y.\mathbf{nil}$, cannot perform such a τ-transition.

The solution is to introduce new relations $\sim_L$ and $\sim_E$ such that for example $p \sim_E q$ iff. $p[\sigma] \dot{\sim}_E q[\sigma]$ for all name substitutions $\sigma : \mathcal{N} \to \mathcal{N}$. Likewise for $\dot{\sim}_L$. We shall not pursue this further in the extension of the Fork Calculus, and we shall not provide any congruence results.

7.2 The ψ-calculus: Syntax and Informal Semantics

The extension of FC is called the ψ-calculus, a pun referring to its relationship with the π-calculus and the fact that it includes a fork operator, the gastronomic version of which has the shape of a ψ. The syntax, generating the language $\mathcal{L}$, is as follows.

$$p \quad ::= \quad \textbf{\textit{nil}} \mid \tau \mid x!y \mid x(y).p \mid p_1 + p_2 \mid p_1 ; p_2 \mid \textbf{\textit{fork}}(p) \mid (y)p \mid [x = y]p \mid \textbf{\textit{fix}}\, x \cdot p \mid x$$

The processes $\textbf{\textit{nil}}$, $p_1 + p_2$, $x(y).p$, $[x = y]p$ and $(y)p$, corresponding to inaction, choice, input, match and restriction, emulate those of the π-calculus. Since sequential composition $p_1 ; p_2$ has been introduced, the silent action prefix τ and the free output prefix $x!y$ from the π-calculus have both become processes. The $\textbf{\textit{fork}}(p)$ process and the fixpoint construct $\textbf{\textit{fix}}\, x \cdot p$ are to be interpreted with the usual meaning. We assume that processes of the form $\textbf{\textit{fix}}\, x \cdot p$ are well-guarded (x is guarded in p) as assumed for the previous calculus in section 5.1.1. Concerning the order of precedence among the operators it is enough here to say that sequential composition binds tightest.

As mentioned, τ and $x!y$ are individual processes while input $x(y).p$ is formulated as an action-prefix as in the π-calculus. This is because in the latter case, (y) is a binding name-occurrence with p as scope. So (y) and p must occur in the same syntactic construct. It is similar to the constructs for task communication in Ada [Bar84]. Ada is an imperative-concurrent programming language allowing sequential composition $S_1 ; S_2$ of statements with side effects. It includes the notion of a task for CCS/CSP-like concurrent programming based on message-passing. The construct for calling an entry E in a task T with a message M (in our terminology: sending message M over the channel $T.E$) is:

$$T.E(M)$$

The construct for an accept statement occurring within some task (process) is:

$$\begin{array}{l} \textbf{accept}\ E(X : ...)\ \textbf{do} \\ \quad S \\ \textbf{end} \end{array}$$

This is similar to our input statement: X is a binding occurrence having S as scope. It describes in terms of S what happens when entry (channel) E is called (communicated to) from outside the task.

As an example of a program in the ψ-calculus, consider the following *one-place-buffer generator*.

$$
\begin{array}{lll}
generator & \stackrel{def}{=} & new(res). \\
 & & \quad (in)(out) \\
 & & \qquad \textbf{\textit{fork}}(\textbf{\textit{fix}}\, buf \cdot in(x).out!x; buf); \\
 & & \qquad res!in; res!out; \\
 & & \qquad generator
\end{array}
$$

The program *generator* repeatedly inputs a channel *res* on the channel *new*. Each such input is followed by the allocation of two new channels *in* and *out*, whereupon a one-place-buffer is forked, that repeatedly inputs from *in* and outputs to y. In order for the context to use this one-place-buffer, the *generator* communicates the channels *in* and *out* to the context on the channel *res*.

7.3 Formal Semantics

The semantics is as usual divided into two layers, one for processes in isolation, and one for multisets of processes.

Processes

We define the labelled transition system $(\mathcal{L}, Lab, \hookrightarrow)$. Concerning the set of labels Lab, this is defined as follows:

$$
\begin{array}{lll}
Com & = & \{k_1?k_2 \mid k_1, k_2 \in Chan\} \cup \{k_1!k_2 \mid k_1, k_2 \in Chan\} \\
Act & = & Com \cup \{\tau\} \\
Lab & = & Act \cup \{\phi(p) \mid p \in \mathcal{L}\} \cup \{\lambda(k) \mid k \in Chan\}
\end{array}
$$

We now define the transition relation $\hookrightarrow \subseteq \mathcal{L} \times Lab \times \mathcal{L}$. Before defining this transition relation we define the predicate $Stop \subseteq \mathcal{L}$. $Stop$ is defined as the least subset of $\mathcal{L}$ satisfying the following:

$$
\begin{array}{lll}
 & Stop(\textbf{\textit{nil}}) & \\
Stop(p_1; p_2) & \Leftarrow & Stop(p_1) \wedge Stop(p_2)
\end{array}
$$

$$\begin{array}{rcl} Stop(p_1 + p_2) & \Leftarrow & Stop(p_1) \wedge Stop(p_2) \\ Stop([x = y]p) & \Leftarrow & x \neq y \\ Stop([x = x]p) & \Leftarrow & Stop(p) \\ Stop(\boldsymbol{fix}\, x \cdot p) & \Leftarrow & Stop(p) \end{array}$$

The operational semantics of processes is then as follows.

Definition 7.3 (The transition relation $\hookrightarrow$) *Let $\hookrightarrow$ be the smallest subset of $\mathcal{L} \times Lab \times \mathcal{L}$ closed under the following rules:*

$$\textbf{(Free Action)} \quad \frac{}{\alpha \overset{\alpha}{\hookrightarrow} \boldsymbol{nil}}\ \alpha = \tau, x!y$$

$$\textbf{(Sequence}_1\textbf{)} \quad \frac{p_1 \overset{l}{\hookrightarrow} p_1'}{p_1; p_2 \overset{l}{\hookrightarrow} p_1'; p_2}$$

$$\textbf{(Sequence}_2\textbf{)} \quad \frac{p_2 \overset{l}{\hookrightarrow} p_2'}{p_1; p_2 \overset{l}{\hookrightarrow} p_2'}\ Stop(p_1)$$

$$\textbf{(Fork)} \quad \frac{}{\boldsymbol{fork}(p) \overset{\phi(p)}{\hookrightarrow} \boldsymbol{nil}}$$

$$\textbf{(Choice}_1\textbf{)} \quad \frac{p_1 \overset{l}{\hookrightarrow} p_1'}{p_1 + p_2 \overset{l}{\hookrightarrow} p_1'}$$

$$\textbf{(Choice}_2\textbf{)} \quad \frac{p_2 \overset{l}{\hookrightarrow} p_2'}{p_1 + p_2 \overset{l}{\hookrightarrow} p_2'}$$

$$\textbf{(Input Action)} \quad \frac{}{x(y).p \overset{x?k}{\hookrightarrow} p[k/y]}\ k \in Chan$$

$$\textbf{(Match)} \quad \frac{p \overset{l}{\hookrightarrow} p'}{[x = x]p \overset{l}{\hookrightarrow} p'}$$

$$\textbf{(Channel)} \quad \frac{}{(y)p \overset{\lambda(k)}{\hookrightarrow} p[k/y]}\ k \in Chan$$

$$\textbf{(Recursion)} \quad \frac{p[(\boldsymbol{fix}\, x \cdot p)/x] \overset{l}{\hookrightarrow} p'}{\boldsymbol{fix}\, x \cdot p \overset{l}{\hookrightarrow} p'}$$

The rules should be fairly simple to read. Note that in the (**Input Action**) rule, a channel k is chosen, hence being substituted for y. In [MPW91] this is referred to as early instantiation. Compare this with the late instantiation in the π-calculus, represented by the rule (**Action**) in section 7.1.

We shall need to refer to the set $CV[p]$ of free channels occurring in a process p:

$$\begin{array}{rcl}
CV[\boldsymbol{nil}] & = & \{\} \\
CV[\tau] & = & \{\} \\
CV[x!y] & = & \{x, y\} \\
CV[x(y).p] & = & \{x\} \cup (CV[p] - \{y\}) \\
CV[p_1 + p_2] & = & CV[p_1] \cup CV[p_2] \\
CV[p_1; p_2] & = & CV[p_1] \cup CV[p_2] \\
CV[\boldsymbol{fork}(p)] & = & CV[p] \\
CV[(y)p] & = & CV[p] - \{y\} \\
CV[[x = y]p] & = & \{x, y\} \cup CV[p] \\
CV[\boldsymbol{fix}\, x \cdot p] & = & CV[p] \\
CV[x] & = & \{\}
\end{array}$$

The channels that are occurring free in a multiset P is calculated as follows:

$$CV[P] \stackrel{def}{=} \bigcup\{CV[p] \mid p \in P\}$$

Configurations

A program is a multiset of processes. We let $\mathcal{M}$ denote the set of programs ($\mathcal{M} = MS(\mathcal{L})$). As usual, we define a K-configuration $K \triangleright P$ to consist of a record K (containing all channels allocated up to a certain point) and a program P.

$$KCon \stackrel{def}{=} \{K \triangleright P \in Record \times \mathcal{M} \mid CV[P] \subseteq K\}$$

We need the function $Msg : Act \to \mathcal{P}(Chan)$ for defining the transition relation. It extracts the (empty or singleton) set of channels that are transmitted in a communication:

$$\begin{aligned}
Msg(\tau) &= \{\} \\
Msg(k_1?k_2) &= \{k_2\} \\
Msg(k_1!k_2) &= \{k_2\}
\end{aligned}$$

The semantics of K-configurations is given in terms of the labelled transition system $(KCon, Act, \longrightarrow)$, where Act was defined in the previous section. The transition relation $\longrightarrow KCon \times Act \times KCon.$ is defined as follows:

Definition 7.4 (The transition relation $\longrightarrow$) *Let $\longrightarrow$ be the smallest subset of $KCon \times Act \times KCon$ closed under the following rules:*

$$(\mathbf{Action}^{\{\!|\,|\!\}}) \qquad \frac{p \overset{\alpha}{\hookrightarrow} p'}{K \triangleright \{\!|p|\!\} \xrightarrow{\alpha} K \cup Msg(\alpha) \triangleright \{\!|p'|\!\}}$$

$$(\mathbf{Fork}^{\{\!|\,|\!\}}) \qquad \frac{p \overset{\phi(q)}{\hookrightarrow} p'}{K \triangleright \{\!|p|\!\} \xrightarrow{\tau} K \triangleright \{\!|p', q|\!\}}$$

$$(\mathbf{Channel}^{\{\!|\,|\!\}}) \qquad \frac{p \overset{\lambda(k)}{\hookrightarrow} p'}{K \triangleright \{\!|p|\!\} \xrightarrow{\tau} K \cup \{k\} \triangleright \{\!|p'|\!\}} \; k \notin K$$

$$(\mathbf{Parallel}_1^{\{\!|\,|\!\}}) \qquad \frac{K \triangleright P_1 \xrightarrow{\alpha} K' \triangleright P_1'}{K \triangleright P_1 \cup P_2 \xrightarrow{\alpha} K' \triangleright P_1' \cup P_2}$$

$$(\mathbf{Parallel}_2^{\{\!|\,|\!\}}) \qquad \frac{K \triangleright P_1 \xrightarrow{c} K \triangleright P_1' \,,\; K \triangleright P_2 \xrightarrow{rev(c)} K \triangleright P_2'}{K \triangleright P_1 \cup P_2 \xrightarrow{\tau} K \triangleright P_1' \cup P_2'}$$

Note in the $(\mathbf{Action}^{\{\!|\,|\!\}})$ rule that the channel transmitted is added to the set K. This is really just to cover the case of an input action $k_1?k_2$ where the channel k_2 is not already in K.

We need to extend the semantics further. In order to internalise dynamic channels (the channels that are introduced by $(\_)\_$), we need a component in the semantics, that identifies the channels that are visible to the context. We refer to the new component as the *window*, and we represent it as the set of visible channels: $Window \overset{def}{=} \mathcal{P}(Chan)$. We let W range over $Window$. The window may change throughout the execution of a program: when some process outputs a local channel, the window is extended with this channel. Recall that the notion of window in chapter 5 were static: windows did not change throughout the execution of programs.

A window together with a record is referred to an an *environment*: $Env \stackrel{def}{=} Window \times Record$.

A configuration $(W, K) \triangleright P$ consists of an environment (W, K) and a program P. We shall only consider well formed configurations, where the window (and the set of free channels in the program) is a subset of the record:

$$Con \stackrel{def}{=} \{(W, K) \triangleright P \in Env \times \mathcal{M} \mid (W \cup CV[P]) \subseteq K\}$$

We shall extend the set of actions by what we call window-extending actions of the form $k_1! <k_1>$, meaning that on channel k_1 is output the channel k_2, and this channel is not already in the window. Consequently the window will be extended with k_2.

$$\begin{array}{rcll} Com^w & = & \{k_1?k_2 & \mid k_1, k_2 \in Chan\} \;\cup \\ & & \{k_1!k_2 & \mid k_1, k_2 \in Chan\} \;\cup \\ & & \{k_1! <k_2> & \mid k_1, k_2 \in Chan\} \\ \\ Act^w & = & Com^w \cup \{\tau\} & \end{array}$$

We need a predicate $\_\,allows\,\_ : Env \times Act$ and a function $\_ \otimes \_ : (Window \times Act) \rightarrow Act^w$, defined as follows.

$$\begin{array}{rcl} (W, K)\; allows\; \tau & = & true \\ (W, K)\; allows\; k_1?k_2 & = & k_1 \in W \wedge k_2 \notin K - W \\ (W, K)\; allows\; k_1!k_2 & = & k_1 \in W \end{array}$$

and

$$\begin{array}{rcll} W \otimes (k1!k2) & = & k_1! <k_2> & if\; k2 \notin W \\ W \otimes \alpha & = & \alpha & otherwise \end{array}$$

$(W, K)\; allows\, \alpha$ holds if the channel transmitted upon is in the window, and additionally in the case of an input: the transmitted channel is not a local one. The application $W \otimes \alpha$ maps the action α into an action where the transmitted channel is marked, if it is output, and not already in the window. This corresponds to the bound output actions in the π-calculus (see figure 7.1). Note, however, that in our case the output channel is not bound, but free: it is a particular channel that is output.

The semantics of configurations is given in terms of the labelled transition system $(Con, Act^w, \longrightarrow)$. The dynamic behaviour of configurations is defined as follows.

Definition 7.5 (The transition relation $\longrightarrow$) *Let $\longrightarrow$ be the smallest subset of $Con \times Act^w \times Con$ satisfying the following rule:*

$$(\textbf{Config}) \quad \frac{K \triangleright P \stackrel{\alpha}{\rightarrowtail} K' \triangleright P'}{(W,K) \triangleright P \stackrel{W \otimes \alpha}{\longrightarrow} (W \cup Msg(\alpha), K') \triangleright P'} \; (W,K) \textit{ allows } \alpha$$

As an example, let us observe the execution of a configuration corresponding to the process $(x)y!x;p$ for some p. The process allocates a new channel x and then outputs this on y, where after it continues as p. We choose the initial environment (window and record) to contain the free channel name y. The channel allocation will give rise to the allocation of a new channel k different from y. Let $I \stackrel{def}{=} \{y\}$ represent the initial set of channels and let $E \stackrel{def}{=} I \cup \{k\}$ represent the initial set extended with the new channel k. We will show that:

$$(I,I) \triangleright \{\!|(x)y!x;p|\!\} \stackrel{\tau}{\longrightarrow} (I,E) \triangleright \{\!|y!k;p[k/x]|\!\} \stackrel{y!<k>}{\longrightarrow} (E,E) \triangleright \{\!|\boldsymbol{nil};p[k/x]|\!\}$$

The τ-transition is derived in steps $(1,2,3)$ below, while the $y!<k>$-transition is derived in steps $(4,5,6)$.

$$\begin{array}{lll}
(1) & (x)y!x;p \stackrel{\lambda(k)}{\hookrightarrow} y!k;p[k/x] & (\textbf{Channel}) \\
(2) & I \triangleright \{\!|(x)y!x;p|\!\} \stackrel{\tau}{\rightarrowtail} E \triangleright \{\!|y!k;p[k/x]|\!\} & (\textbf{Channel}^{\{\!|\,|\!\}}) \\
(3) & (I,I) \triangleright \{\!|(x)y!x;p|\!\} \stackrel{\tau}{\longrightarrow} (I,E) \triangleright \{\!|y!k;p[k/x]|\!\} & (\textbf{Config}) \\
 & & \\
(4) & y!k;p[k/x] \stackrel{y!k}{\hookrightarrow} \boldsymbol{nil};p[k/x] & (\textbf{Free Action}) \\
(5) & E \triangleright \{\!|y!k;p[k/x]|\!\} \stackrel{y!k}{\rightarrowtail} E \triangleright \{\!|\boldsymbol{nil};p[k/x]|\!\} & (\textbf{Action}^{\{\!|\,|\!\}}) \\
(6) & (I,E) \triangleright \{\!|y!k;p[k/x]|\!\} \stackrel{y!<k>}{\longrightarrow} (E,E) \triangleright \{\!|\boldsymbol{nil};p[k/x]|\!\} & (\textbf{Config})
\end{array}$$

We end this section by introducing an operation for creating an environment from a process. Let $Env[\_] : \mathcal{L} \rightarrow Env$ be defined as follows $Env[p] \stackrel{def}{=} (CV[p], CV[p])$. This environment contains (in window as well as in record) all the channels that occur free in p.

Process Equivalence

We shall now define an equivalence relation $\equiv \subseteq \mathcal{L} \times \mathcal{L}$ between processes. We begin with defining an equivalence relation $\smile \subseteq Con \times Con$ between configurations.

We note that the semantics of configurations is a labelled transition system $(Con, Act^w, \longrightarrow)$. As described in chapter 2, such a labelled transition system generates a bisimulation equivalence between the configurations in Con. Applied to configurations, the definition of a bisimulation would be the following: a relation $S \subseteq Con \times Con$ is a bisimulation is whenever $(C_1, C_2) \in S$ then

1. Whenever $C_1 \xrightarrow{\alpha} C_1'$ for some C_1' then $C_2 \xrightarrow{\alpha} C_2'$ for some C_2' and $(C_1', C_2') \in S$
2. Whenever $C_2 \xrightarrow{\alpha} C_2'$ for some C_2' then $C_1 \xrightarrow{\alpha} C_1'$ for some C_1' and $(C_1', C_2') \in S$

It turns out, that we *cannot* use the bisimulation equivalence generated in this way, since it distinguishes programs that output local channels (scope extrusion). To see this, consider the following two processes p and q that in this way become different, although they show the same behaviour to the observer:

$$\begin{array}{lcl} p & \stackrel{def}{=} & (x)\tau; a!x \\ q & \stackrel{def}{=} & \tau; (x)a!x \end{array}$$

Assuming $W = K = \{a\}$ we can "bisimulate" the two configurations for p and q as follows (K^x stands for $K \cup \{x\}$):

$$\begin{array}{cc} \mathit{Config}[p] & \mathit{Config}[q] \\ = & = \\ (W, K) \triangleright \{\!|(x)\tau; a!x|\!\} & (W, K) \triangleright \{\!|\tau; (x)a!x|\!\} \\ \downarrow \tau & \downarrow \tau \\ (W, K^m) \triangleright \{\!|\tau; a!m|\!\} & (W, K) \triangleright \{\!|\boldsymbol{nil}; (x)a!x|\!\} \\ \downarrow \tau & \downarrow \tau \\ (W, K^m) \triangleright \{\!|\boldsymbol{nil}; a!m|\!\} & (W, K^n) \triangleright \{\!|a!n|\!\} \end{array}$$

In the first transition p allocates a new channel m, and q just performs a τ-transition. In the second transition, let q move first; then it chooses an arbitrary new channel for x, say $n \neq m$. Obviously $a!m$ and $a!n$ show different behaviour if requiring identity of m and n. Hence, when channel allocation is "out of synchronisation" we cannot guarantee that identical channels are allocated for the same "variables".

We proceed with a modified notion of bisimulation. Note, that also in the π-calculus, the standard notion of bisimulation has been replaced with a more involved one (see definitions 7.1 and 7.2). The definition involves renaming of configurations. That is, $((W, K) \triangleright \{\!|p_1, \ldots, p_n|\!\})[y/x]$ stands for $((W[y/x], K[y/x]) \triangleright \{\!|p_1[y/x], \ldots, p_n[y/x]|\!\})$ where the renaming $S[y/x]$ of a set S (be it K or W) stands for S if $x \notin S$ and for $S - \{x\} \cup \{y\}$ if $x \in S$.

Definition 7.6 (Bisimulation) *S is a simulation if whenever $(C_1, C_2) \in S$ then*

1. *if $C_1 \xrightarrow{\alpha} C'_1$ where $\alpha \neq k_1! \lt k_2 \gt$ then $C_2 \xrightarrow{\alpha} C'_2$ and $(C'_1, C'_2) \in S$*

2. *if $C_1 \xrightarrow{k!\lhd m\rhd} C'_1$ then $C_2 \xrightarrow{k!\lhd n\rhd} C'_2$ and $(C'_1[l/m], C'_2[l/n]) \in S$, for some $l \notin CV[C'_1] \cup CV[C'_2]$*

S is a bisimulation if S as well as S^{-1} are simulations. $C_1 \smile C_2$ iff. $(C_1, C_2) \in S$ for some bisimulation S.

Observe that in clause (2) we allow two configurations to generate and output different channels, it is just required that the continuations are bisimilar up to renaming of the output channels m and n into a common *new* channel l.

Two processes are equivalent, if they are equivalent when “lifted” to configurations. Formally, we associate to each process its ‘initial configuration’:

Definition 7.7 (Initial configuration of a process) *The initial configuration $Config[p]$ of a process p is defined as:*

$$Config[p] \stackrel{def}{=} (CV[p], CV[p]) \triangleright \{\!|p; \pi|\!\}$$

The event π in fact stands for a communication $\pi!\pi$, but we shall abstract away from that detail. We are now able to give the following formal definition of the process equivalence $\equiv$:

Definition 7.8 (Process equivalence $\equiv$) *The relation $\equiv \subseteq \mathcal{L} \times \mathcal{L}$ is defined as:*

$$p \equiv q \Leftrightarrow Config[p] \smile Config[q]$$

The notion of bisimulation defined here should correspond to early bisimulation for the π-calculus, as presented in definition 7.2.

As an example, let us reconsider variants of the two processes p and q from above that were not equivalent according to the traditional notion of bisimulation, but which will be equivalent according to the notion of bisimulation in definition 7.6. First we bisimulate the two processes until and including their output of (different) local channels m and n on the channel a.

$$
\begin{array}{cc}
Config[p] & Config[q] \\
= & = \\
(W,K) \triangleright \{|(x)\tau; a!x; x(y).\boldsymbol{nil}|\} & (W,K) \triangleright \{|\tau; (x)a!x; x(y).\boldsymbol{nil}|\} \\
\downarrow \tau & \downarrow \tau \\
(W,K^m) \triangleright \{|\tau; a!m; m(y).\boldsymbol{nil}|\} & (W,K) \triangleright \{|\boldsymbol{nil}; (x)a!x; x(y).\boldsymbol{nil}|\} \\
\downarrow \tau & \downarrow \tau \\
(W,K^m) \triangleright \{|\boldsymbol{nil}; a!m; m(y).\boldsymbol{nil}|\} & (W,K^n) \triangleright \{|a!n; n(y).\boldsymbol{nil}|\} \\
\downarrow a!<m> & \downarrow a!<n> \\
(W^m,K^m) \triangleright \{|\boldsymbol{nil}; m(y).\boldsymbol{nil}|\} & (W^n,K^n) \triangleright \{|\boldsymbol{nil}; n(y).\boldsymbol{nil}|\} \\
= & = \\
C'_p & C'_q
\end{array}
$$

Observe that clause (2) in definition 7.6 allows the two configurations to generate and output different channels m and n, it is just required that the continuations C'_p and C'_q are bisimilar up to renaming of the output channels m and n into a common *new* channel l. Obviously we have that:

$$C'_p[l/m] \quad = \quad (W^l, K^l) \triangleright \{|\boldsymbol{nil}; l(y).\boldsymbol{nil}|\} \quad = \quad C'_q[l/n]$$

In the Facile equivalence defined in [PGM90, Pra91], it is possible to ask whether two behaviours (processes) B_1 and B_2 are equivalent with respect to a given window W, written as $B_1 \overset{w}{\approx} B_2$. This is obtained by indexing the equivalence relation with windows. In Facile, two behaviours are equivalent $B_1 \approx B_2$ if $\forall W \cdot B_1 \overset{w}{\approx} B_2$. We have taken the more traditional approach from process algebra where equivalence relations are normally not indexed: either two processes are equivalent, or they are not. Since we, however, need windows, we just let them be part of configurations.

Also in Facile, the K-component is universally quantified over in the definition of the bisimulation relation as discussed below in section 7.4. What we do is to fix the K and the W to the minimal ones in the initial configuration of a process. Then we avoid the universal quantifications, and our equivalence notion gets more operational.

7.4 Alternative Bisimulations

In definition 7.6 of bisimulation, a renaming of whole configurations takes place. Prasad mentions in [Pra91] that it may not be viable in real systems to do this kind of renaming. Since we are developing a *theory* of equivalence, we are satisfied with our definition. If one on the other hand is thinking in terms of an automated equivalence checker, surely such renaming is out of the question. A solution may be to make the renaming explicit by indexing bisimulations with channel-correspondence relations. Another alternative is to do as Prasad does for Facile, an approach that we claim is not satisfactory. We shall comment on the two approaches below.

Indexed Bisimulations

We shall consider relations $R \subseteq Chan \times Chan$ between channels. The intuition is as follows. Suppose R is such a relation, then $(m, n) \in R$ if m and n has been output in corresponding window-expanding actions such as for example $k!\,<m>$ and $k!\,<n>$ from two different configurations being under examination for bisimilarity. For such a relation R we define the relation $\_ \overset{R}{\approx} \_ \subseteq Act^w \times Act^w$ between actions as the least relation satisfying the following:

$$
\begin{array}{rcl}
\tau \overset{R}{\approx} \tau & & \\
k?m \overset{R}{\approx} l?n & \Leftarrow & (k,l) \in R \wedge (m,n) \in R \\
k!m \overset{R}{\approx} l!n & \Leftarrow & (k,l) \in R \wedge (m,n) \in R \\
k!\,<m> \overset{R}{\approx} l!\,<n> & \Leftarrow & (k,l) \in R
\end{array}
$$

This relation naturally relates actions that carry R-related channels. We also need a function $\_ \oplus (\_,\_) : (\mathcal{P}(Chan \times Chan) \times Act^w \times Act^w) \rightarrow \mathcal{P}(Chan \times Chan)$ that extends a relation with the exported channels from two window-expanding actions:

$$
\begin{array}{rcl}
R \oplus (k!\,<m>, l!\,<n>) & = & R \cup \{(m,n)\} \\
R \oplus (\alpha_1, \alpha_2) & = & R \quad \textit{otherwise}
\end{array}
$$

Definition 7.9 (Indexed Bisimulation) *A ternary relation $S \subseteq (\mathcal{P}(Chan \times Chan)) \times Con \times Con$ is a bisimulation if whenever $(R, C_1, C_2) \in S$ then*

1. *Whenever $C_1 \xrightarrow{\alpha_1} C_1'$ for some α_1 and C_1' then $C_2 \xrightarrow{\alpha_2} C_2'$ for some α_2 and C_2' where $\alpha_1 \overset{R}{\approx} \alpha_2$ and $(R \oplus (\alpha_1, \alpha_2), C_1', C_2') \in S$*

2. *Whenever* $C_2 \xrightarrow{\alpha_2} C_2'$ *for some* α_2 *and* C_2' *then* $C_1 \xrightarrow{\alpha_1} C_1'$ *for some* α_1 *and* C_1' *where* $\alpha_1 \overset{R}{\approx} \alpha_2$ *and* $(R \oplus (\alpha_1, \alpha_2), C_1', C_2') \in S$

$C_1 \overset{R}{\smile} C_2$ *if* $(R, C_1, C_2) \in S$ *for some bisimulation* S. $C_1 \smile C_2$ *if* $C_1 \overset{\{\}}{\smile} C_2$.

We conjecture that the here defined notion of bisimulation is the same as defined in definition 7.6.

As an example, let us bisimulate the two processes p and q from before. The bisimulation starts with the triple $(\{\}, \mathit{Config}[p], \mathit{Config}[q])$:

$$\begin{array}{ccc}
 & \mathit{Config}[p] & \mathit{Config}[q] \\
 & = & = \\
\{\} & (W,K) \triangleright \{\!|(x)\tau; a!x; x(y).\boldsymbol{nil}|\!\} & (W,K) \triangleright \{\!|\tau; (x)a!x; x(y).\boldsymbol{nil}|\!\} \\
 & \downarrow \tau & \downarrow \tau \\
\{\} & (W,K^m) \triangleright \{\!|\tau; a!m; m(y).\boldsymbol{nil}|\!\} & (W,K) \triangleright \{\!|\boldsymbol{nil}; (x)a!x; x(y).\boldsymbol{nil}|\!\} \\
 & \downarrow \tau & \downarrow \tau \\
\{\} & (W,K^m) \triangleright \{\!|\boldsymbol{nil}; a!m; m(y).\boldsymbol{nil}|\!\} & (W,K^n) \triangleright \{\!|a!n; n(y).\boldsymbol{nil}|\!\} \\
 & \downarrow a! < m > & \downarrow a! < n > \\
\{(m,n)\} & (W^m,K^m) \triangleright \{\!|\boldsymbol{nil}; m(y).\boldsymbol{nil}|\!\} & (W^n,K^n) \triangleright \{\!|\boldsymbol{nil}; n(y).\boldsymbol{nil}|\!\} \\
 & = & = \\
 & C_p' & C_q'
\end{array}$$

C_p' and C_q' are certainly bisimilar relative to $\{(m,n)\}$ which is written $C_p' \overset{\{(m,n)\}}{\smile} C_q'$. That is, if $C_p' \xrightarrow{m?k} (\ldots,\ldots) \triangleright \{\!|\boldsymbol{nil}|\!\}$ for some k then $C_q' \xrightarrow{n?k} (\ldots,\ldots) \triangleright \{\!|\boldsymbol{nil}|\!\}$ and vice versa.

Note, however, that since channel allocation is amongst infinitely many channels and since any of them may be chosen, bisimulations tend to be infinite when involving channel allocating processes. This can be dealt with by turning channel allocation into a deterministic function, for example by representing the record K of configurations as a counter over natural numbers.

The Facile Solution

Prasad provides a different "solution" to the problem of separate channel allocation. He, however, formulates the problem a bit differently, thereby in our opinion obtaining a non-satisfactory solution. He writes:

> Another spurious distinction between programs arises from examining whether one program can generate a *particular* channel that the other cannot, because the latter was already using this as a local channel.

An example of this situation can arise when comparing the two programs $p^{yx} \stackrel{def}{=} (y)(x)a!x$ and $q^{\tau x} \stackrel{def}{=} \tau;(x)a!x$. In the first transition of p^{yx}, a channel, say m, is allocated for y, and $q^{\tau x}$ just performs a τ-transition. In the second transition for $q^{\tau x}$, a channel, let it also be m, is allocated for x. Now in the second transition for p^{yx} it is not possible to allocate m for x since m has already been allocated for y.

Prasad continues:

> Instead we insist that in comparing the behaviour of two programs P and Q through a window W, that P will not generate any channels that are local to Q and vice versa. We do so by starting with configurations where the set of existing channels *[free translation: K includes the channels of both programs]*.

His solution is always to compare two configurations where the K-component is the same, and such that it contains all the channels in the two configurations together. This is obtained by choosing a new K after each corresponding pair of transitions in a bisimulation, a K that contains the channels of the two configurations obtained so far – in fact he universally quantifies over all such K. As a consequence, each process always knows what the other process has allocated.

In the above example, after the process p^{yx} has allocated m for y, it is no longer possible for the process $q^{\tau x}$ to allocate m for x. Instead, $q^{\tau x}$ is forced to allocate a different channel, say n, and this time p^{yx} is able to follow. This solution works for the above example, where the allocation for x in both processes happens "at the same time": if one process allocates n for x then we let the other process also allocate n for x (note that we are "in control" of corresponding transitions when "performing" a bisimulation).

If, however, the allocation for x happens at different times as is the case in the processes $p \stackrel{def}{=} (x)\tau;p'$ and $q \stackrel{def}{=} \tau;(x)q'$, then the solution does in fact not work. The reason is that when p allocates m for x, then in the next transition q will be prevented in also allocating m for x, since p's allocation beforehand has been made globally know. Surely, one would expect p and q to be equivalent, and so they are according to our definition 7.6. This definition can be regarded as a solution to the problem formulated as follows in [Pra91] page 162-163:

> Our formulation takes the viewpoint that since distinct channels are different as values, we will also consider expansions to a window using different channels as yielding different new windows. Of course, there are alternative formulations of behavioural equivalence where one need not require that the window be expanded identically. It is conceivable that such an approach may be more general, allowing for

possibly more identifications to be made between processes. For example, one may identify processes that exhibit behaviour that would be equivalent *up to isomorphic channel renaming*, i.e., they exhibit similar behavioural patterns but on different channels. However, the corresponding notions of observability and equivalence are harder to formulate, and it is not clear what questions such a formulation would address.

7.5 A Logic

In [MPW91] a modification of Hennessy-Milner logic is defined for the π-calculus. This logic allows to formulate scope extrusion: the output of a local channel. Below is presented a similar modification of Hennessy-Milner logic for the ψ-calculus. The syntax, generating the language Ψ, is as follows.

$$
\begin{array}{lcl}
\psi & ::= & \boldsymbol{tt} \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid \ <\alpha> \psi \mid [x = y]\psi \\
\alpha & ::= & \tau \mid x!y \mid x!(y) \mid x(y)
\end{array}
$$

The propositional connectives have their obvious meaning. A process satisfies $<\alpha> \psi$ if it can perform the action α, and then reach a state which satisfies ψ. A process satisfies $[x = x]\psi$ if it satisfies ψ. If x and y are distinct, then any process satisfies $[x = y]\psi$.

There are four kinds of actions. The τ represents an internal computation as usual. The $x!y$ action represents the output of the channel y on the channel x. These are free actions in that they contain only free channel names. The remaining two (kinds of) actions are bound in that (y) is a binding occurrence which has ψ as scope in the formulae $<x!(y)> \psi$ and $<x(y)> \psi$.

The $x!(y)$ action represents the output of *some local* channel, referred to as y, on the channel x. Hence, this is the modality for scope extrusion. Which channel is output is left open, but y will be bound to the real channel being output, when checking a particular process against the modality.

Finally, the $x(y)$ action represents the input of *some* channel on the channel x. Which channel is input is left open, but y will be bound to the real channel being input, when checking a particular process against the modality.

We shall define the satisfaction relation $\models$ between processes and properties in two steps; first we define a satisfaction relation $\models \subseteq Con \times \Psi$ between configurations and properties.

Definition 7.10 (The satisfaction relation $\models$) *The relation $\models\ \subseteq Con \times \Psi$ is the least relation such that for any configuration C in Con, and for any properties ψ, ψ_1 and ψ_2 in Ψ, and for any channel names x, y in Chan,*

$$
\begin{array}{lllll}
C & \models & \textit{tt} & & \\
C & \models & \psi_1 \wedge \psi_2 & \textit{iff} & P \models \psi_1 \textit{ and } P \models \psi_2 \\
C & \models & \neg\psi & \textit{iff} & \textit{not } P \models \psi \\
C & \models & {<}\alpha{>}\,\psi & \textit{iff} & C \xrightarrow{\alpha} C' \textit{ for some } C' \textit{ and } C' \models \psi, \textit{ for } \alpha = \tau, x!y \\
C & \models & {<}x!(y){>}\,\psi & \textit{iff} & C \xrightarrow{x!{<}k{>}} C' \textit{ for some } k \textit{ and } C', \textit{ and } C'[l/k] \models \psi[l/y], \\
 & & & & \textit{for some } l \notin CV[C'] \cup CV[\psi] \\
C & \models & {<}x(y){>}\,\psi & \textit{iff} & C \xrightarrow{x?k} C' \textit{ for some } k \textit{ and } C', \textit{ and } C' \models \psi[k/y] \\
C & \models & [x = y]\psi & \textit{iff} & x = y \textit{ implies } C \models \psi
\end{array}
$$

The renaming to some new l in the definition of ${<}x!(y){>}\,\psi$ is necessary in order to deal with the situation where the exported channel k is already occurring free in the formula ψ.

The satisfaction relation between processes and formulae is now defined as follows.

Definition 7.11 (The satisfaction relation $\models$) *The relation $\models\ \subseteq \mathcal{L} \times \Psi$ is defined as:*

$$p \models \psi \quad \textit{iff} \quad \textit{Config}[p] \models \psi$$

It is conjectured that one can replace the existential quantification over l in the semantics of ${<}x!(y){>}\,\psi$ with a universal quantification without changing the meaning. Further, one could even avoid the renaming by defining process satisfaction as follows: $p \models \psi$ iff $(\textit{Config}[p] \uplus CV[\psi]) \models \psi$; where $((W, K) \triangleright P) \uplus S$ stands for $(W, K \cup S) \triangleright P$. So no free channels (S) in ψ can now be allocated by the process p.

The derived forms ***ff***, $\psi_1 \vee \psi_2$, $\psi_1 \Rightarrow \psi_2$, and $[\alpha]\psi$ can be defined as usual. Note in particular the definition of the latter:

$$[\alpha]\psi \quad \stackrel{def}{=} \quad \neg\, {<}\alpha{>}\, \neg\psi$$

The interesting cases are $[x!(y)]\psi$ and $[x(y)]\psi$, which get the interpretation below, by simple manipulation with negation in definition 7.10:

$$
\begin{array}{lllll}
C & \models & [x!(y)]\psi & \text{iff} & \text{for any } k \text{ and } C', \text{ if } C \xrightarrow{x!{<}k{>}} C' \text{ then } C'[l/k] \models \psi[l/y], \\
 & & & & \text{for any } l \notin CV[C'] \cup CV[\psi] \\
C & \models & [x(y)]\psi & \text{iff} & \text{for any } k \text{ and } C', \text{ if } C \xrightarrow{x?k} C' \text{ then } C' \models \psi[k/y]
\end{array}
$$

We shall also illustrate how forking can be specified. The technique is exactly as for FC in section 4.2. Recall that the initial configuration of a process contains the π action, which signals termination of the "main" process under observation. There is only one such π in a configuration. When a process satisfies $<\pi>\,\boldsymbol{tt}$ it means that the process may terminate. As was done in section 4.2, the following derived form is introduced:

$$\bullet\psi \quad \stackrel{def}{=} \quad <\pi>\psi$$

It means that the process terminates, and what has been forked satisfies ψ.

In [MPW91], a Hennessy-Milner like logic is defined for the π-calculus. Various subsets of this logic correspond to various notions of equivalence: each of these subsets is adequate with respect to a certain equivalence. That is: two processes are equivalent iff. they satisfy the same formulae in the subset. In particular a logic (called $\mathcal{BM}$, page 17 theorem 2) being a subset similar to our logic is proved to be adequate wrt. their early bisimulation (see definition 7.2 above). Since our bisimulation should correspond to their early bisimulation, we therefore conjecture that the logic presented here is adequate wrt. to the bisimulation defined.

Two small examples are now presented that illustrate the logic. First, consider the following program:

$$p \quad \stackrel{def}{=} \quad start(res).(out)\boldsymbol{fork}(out!f);res!out$$

The program inputs a channel res on $start$, allocates a new channel out, forks a process outputting the free channel f on out, and finally informs via res the context about the new channel out. The process satisfies the formula below:

$$p \quad \models \quad <start(r)><\tau><\tau><r!(o)>\bullet<o!f>\,\boldsymbol{tt}$$

As a second example, recall the program given earlier, which we repeat here:

$$\begin{array}{lcl} generator & \stackrel{def}{=} & new(res). \\ & & \quad (in)(out) \\ & & \quad\quad \boldsymbol{fork}(\boldsymbol{fix}\, buf \cdot in(x).out!x;buf); \\ & & \quad\quad res!in;res!out; \\ & & \quad\quad generator \end{array}$$

This program satisfies the following formula:

$$\begin{array}{lcl} generator & \models & <new(r)><\tau><\tau><\tau><r!(i)> \\ & & \quad ((<r!(o)><i(m)><o!m>\,\boldsymbol{tt}) \\ & & \quad\ \wedge \\ & & \quad\ (<i(m)><r!(o)> \\ & & \quad\quad (<o!m>\,\boldsymbol{tt}\wedge\ <new(r')>\,\boldsymbol{tt}))) \end{array}$$

Part IV

Towards a Logic for CML

Chapter 8

A Family of ML Languages

In the following two chapters we shall try to approach the initial problem that inspired the work on the Fork Calculus: to define a specification logic for Concurrent ML (CML). ML [MTH90] is basically a functional programming language. CML [Rep91a, Rep92] is an extension of ML with FC-like concurrency primitives based on synchronous communication as in the CCS/CSP family of languages. For ML there further exists an extension with specification logic, called Extended ML (EML) [San89], [ST90], [Kaz91], [SdST90], and [KST93]. This is ML extended with specification primitives allowing one to specify properties about functional ML programs.

The objective of the two chapters is to outline an Extended CML (ECML) analog to Extended ML. The idea is illustrated in figure 8.1.

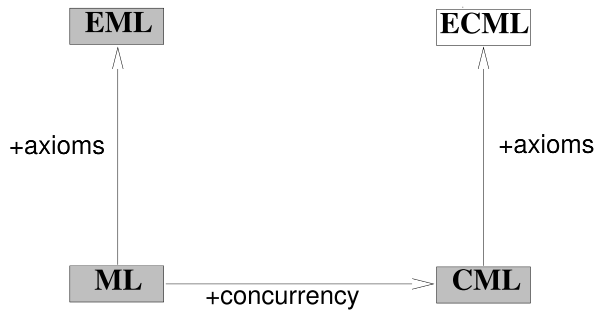

Figure 8.1: The ML Landscape

We shall draw on the experiences with the Fork Calculus for which we have defined various logics. The central problems treated in the Fork Calculus are those of process termination in the presence of a ***fork***-operator (modelled by an occurrence of the π event in FC), and communication of channels over channels. To this comes, in CML, communication of processes. That is, the logic must also allow to state that a process communicates a process satisfying a certain property. An extension of CCS where processes can communicate processes is found in [Tho90, Tho89]. Further, since ML values like integers, lists and pure functions can be communicated, our logic must also try to combine some sort of value logic, like predicate logic, with a model logic.

As we have seen, the definition of logics have been based on the general idea that processes denote *labelled* transition systems, where transitions are labelled with actions. These logics could be called *action based*. We want to define an action based logic for CML as well. There exist at least two operational semantics of CML ([Rep91c] and [BMT92]). None of these, however, associate action labelled transition systems with CML programs. So in order to define an action based logic, we need to define a new CML semantics, where transitions are labelled with actions. Such a semantics is defined in the following.

We state a warning here: although we define a semantics of CML and although we sketch a semantics of ECML, the two chapters have an informal flavor with respect to semantics, in spite of its formality with respect to syntax. Future work will put more formality behind, and the current work can be seen as an example-driven language design. The formality in previous chapters allow us to "play" at this point. We believe that even just an idea about what a CML logic would look like is of interest, since no such logic exists, as far as we are aware.

Section 8.1 is a short introduction to respectively ML, EML and CML. Section 8.2 defines a semantics for CML in terms of an action-labelled transition system. Chapter 9 contains our proposal for a CML logic.

8.1 An Introduction to ML, EML and CML

The examples used in the presentation of ML and CML are inspired by [Rep91a].

8.1.1 ML

ML consists of two sublanguages: the *core language* and the *module language*. The core language contains constructs for programming in "the small". These constructs allow one to define *types* and their *values*. Values are denoted by *expressions* – traditional statement constructs known from other languages, such as blocks, conditionals, case statements and assignment, etc., are represented as expressions in ML.

The module language contains constructs for programming in "the large", essentially allowing to group together definitions of types and values.

Although ML is a mostly *functional* language, it does provide *mutable cells*, pointed to by references belonging to reference types. Cells can be updated by assignment, and expressions can be composed with semicolon (`;`). Another non-functional facet is that expressions can raise *exceptions*. ML is often called a functional language, since its constructs very much support functional programming with pure functions. That is, it is higher order in that functions are first-class values. It has a powerful *datatype* mechanism: datatype declarations introduce constructors that may be used to build values in expressions and to decompose values in *patterns*. The patterns are typically used in equational definitions of functions, and in case expressions.

ML is strongly typed in that every expression has a type, which is inferred if not given explicitly. Types may be polymorphic. ML is lexically scoped.

A new type is defined by providing its name and its *constructors*. As an example, ML has a predefined boolean type `bool`:

```
datatype bool = true|false
```

This type has two nullary value constructors associated: `true` and false. There are other basic types that from a conceptual point of view are defined in the same way: for example integers `int`, and the type `unit` having the intended definition `datatype unit = ()`. It contains only one element, and is typically used as argument or result type of "functions" with side-effects on cells. As a more interesting example, ML has a predefined polymorphic list type constructor, defined by the declaration:

```
datatype 'a list = nil | :: of ('a * 'a list)
infix ::
```

In the first line, the polymorphic list type constructor `list` is introduced. `'a` is a type variable that can be instantiated with any type. It has associated two value constructors: `nil`, the empty list, and `::`, the cons-operator that creates a new list from an element and an old list. The second line defines `::` to be in infix operator. The expression: `true::false::nil` denotes a list with two boolean elements. The expression $[e_1, \ldots, e_n]$ is often used as a derived form for $e_1 :: \ldots :: e_n :: nil$. The following is the definition of a function `rev` that reverses a list; it is predefined in ML:

```
fun rev l = let
              fun rev'([],l) = l
                 |rev'(x::r,l) = rev'(r,x::l)
            in
              rev'(l,[])
            end
```

The function is defined in terms of an auxiliary function `rev'` introduced in a *let*-construct. This function has type `'a list * 'a list -> 'a list`, and is defined by two clauses, one for the case where the first list is empty and one where it is not. Each clause has the form *f pat*`=`*exp*, where *pat* is a pattern. The function is defined to avoid repeated list concatenation with the infix operator `@ : 'a list * 'a list -> 'a list` as in the fragment: `rev(x::l) = rev(l)@[x]`.

The module system allows to group type and value definitions. The basic grouping unit is the *structure*. A structure has a "type" which is called a *signature*. Signatures play the role as "interfaces" to structures, describing which value names are exported from the structure, together with their type. Also type names and exception names can occur in a signature. Below is given the signature and corresponding structure for a set of queue operations. First the signature:

```
signature QUEUE =
  sig
    type 'a queue
    exception EmptyQ
    val empty    : 'a queue
    val put      : 'a * 'a queue -> 'a queue
    val remove   : 'a queue -> 'a * 'a queue
    val head     : 'a queue -> 'a
end
```

A structure having this signature must provide a polymorphic type `'a queue`, an exception and four values. For example, the value `empty` denotes a queue, intended to be the empty one, and the function `put` takes an element and a queue as arguments (`*` is a binary type operator representing cartesian product), and returns a new queue. The following structure satisfies the signature:

```
structure Queue:QUEUE =
  struct
    datatype 'a queue = Q of {front : 'a list,rear : 'a list}
    exception EmptyQ
    val empty = Q{front=[],rear=[]}
    fun put(x,Q{front,rear}) = Q{front=front, rear=x::rear}
    fun remove(Q{front=[],rear=[]}) =
          raise EmptyQ
       |remove(Q{front=[],rear}) =
          remove(Q{front=rev rear,rear=[]})
       |remove(Q{front=x::r,rear}) =
          (x,Q{front=r,rear=rear})
    fun head q = let val (x,_) = remove q in x end
  end
```

The `datatype` definition defines queues to be constructed by applying the constructor `Q` to a *record* with two fields: a `front` and a `rear`, both fields being lists of queue elements (the implementation avoids costful list concatenation). Due to the `'a queue` definition in the signature, the record implementation cannot be "seen" by a user of the structure. This is ensured by the association `Queue:QUEUE` in the head of the structure definition.

The definition of `empty` shows how a record is created, here with empty lists as components. The definition of `remove` shows different forms of record patterns (to the left of the defining equalities). The pattern `Q{front=[],rear}` is short for `Q{front=[],rear=rear}`, and is matched by any record having a `front` that is empty, and upon match, the rear is bound to `rear`. The record pattern `Q{front=x::r,rear}` is matched by any record having at least one element in the front.

The example shows how an exception is raised when applying `remove` to the empty queue. In the definition of `head` there is a use of the *let*-expression and a cartesian product pattern `(x,_)`: let `x` be bound to the next element in the queue (and ignore the reduced queue, indicated by the *wildcard pattern* `_`).

The contents of the structure `Queue` can be obtained by "dot"-notation, for example `Queue.put(1,Queue.empty)` creates a queue of integers. The module language of ML is more powerful than described here. It also provides means of combining structures and it provides the notion of *functor*, which is a function from structures to structures.

Although function definitions normally are presented as above in terms of equations, ML provides what corresponds to a lambda abstraction of the form `fn`*match*, where a *match* is a pattern-expression pair sequence of the form pat_1`=>`$exp_1|\ldots|pat_n$`=>`exp_n. As an example we could inside the `Queue` structure have defined a function determining whether a queue is empty or not as follows:

```
val is_empty = fn Q{front=[],rear=[]} => true
                 |_ => false
```

As an example of how exceptions are handled, consider the definition of the same function, now outside the `Queue` structure where the record implementation is unknown:

```
val is_empty = fn q => (Queue.remove q;false)
                       handle Queue.EmptyQ => true
```

8.1.2 EML

Extended ML (EML) is an extension of ML, that allows formal development of modular ML programs which are correct with respect to a specification of their required behaviour. Specifications are written in higher-order equational logic. EML covers all of ML, except for reference types: it is not possible to specify properties about programs that perform assignments to global mutable cells.

Consider the ML signature `QUEUE` from the previous section. It contained a collection of type, exception and value names, and for each value name, a type. The intended meaning of these names were only indicated informally in english, and it was not part of the signature. EML allows to state such intentions as *axioms* in signatures, where axioms are just boolean ML expressions, augmented with a few extra constructs for giving some real specification power. That is: to the category *spec*, which is the contents of a signature, the alternative `axiom`*exp* is added, and to the category *exp* of expressions, one has basically added equivalence exp_1 `==` exp_2 and universal quantification `forall` *match*, where `forall x:int => x<=0 orelse x>0` is an example. The '`orelse`' construct is a derived form in ML, just as is '`andalso`'. A structure matches a signature if it matches each of the type requirements, and each of the axioms, the latter meaning that the axiom evaluates to nothing but `true` in the context of the definitions contained in the structure.

When matching a structure against an ML signature a mechanic type check is performed wrt. the type requirements, just as in ML. Axioms, however, cannot in general be checked mechanically, and will typically require human interaction with a theorem prover.

There are a number of derived forms In EML, for example negation (`not`) and implication (`implies`), existential quantification (`exists` *match*), constructs for specifying exceptions and constructs for specifying termination properties.

As an example, the signature `QUEUE` including a set of axioms is shown below.

```
signature QUEUE =
  sig
    type 'a  queue
    exception  EmptyQ
    val empty : 'a queue
    val put : 'a * 'a queue -> 'a queue
    val remove : 'a queue  -> 'a * 'a queue
    val head : 'a queue -> 'a
    axiom remove(empty) raises EmptyQ
    axiom remove(put(x,empty))   ==  (x,empty)
    axiom not(remove(q) raises EmptyQ)  implies
```

```
            remove(put(x,q))  ==
              let val (y,q') =  remove(q) in
                (y,put(x,q'))
              end
      axiom head q == let val (x,_) = remove q in x end
    end
```

All the axioms are supposed to be prefixed with a universal quantification over all the free variables. So, for example, the second axiom is short for:

```
forall x:'a => remove(put(x,empty)) == (x,empty)
```

The first axiom states that `remove(empty)` raises the exception `EmptyQ`. The '`raises`' construct, having the form: *exp* `raises` *pat*, can in fact be defined as a derived form in ML, and is not special to EML.

The equivalence predicate exp_1 `==` exp_2 is needed in the logic even though ML has a built-in equality function `=`. The ML equality, however, is not satisfactory for specification purposes; one reason is that it is a function, and with ML's call by value semantics, the expression: `raise EmptyQ = raise EmptyQ` itself raises the exception `EmptyQ`, while `raise EmptyQ == raise EmptyQ` yields `true`, which is the result that we want. So expressions having the same properties wrt. returned value, exceptions and non-termination must be equivalent.

Another aspect of equivalence is that it in fact represents *behavioural equivalence* as explained with the following example. From the equations we can derive that:

```
remove(put(1,put(2,empty))) ==  (2,put(1,empty))
```

Looking at the record-representation this does, however, not hold since:

```
remove(put(1,put(2,empty))) = (2,Q{front=[1],rear=[] })
           (2,put(1,empty)) = (2,Q{front=[] ,rear=[1]})
```

The point is that the first of the above three equations is only true in the sense that the two terms `remove(put(1,put(2,empty)))` and `(2,put(1,empty))` cannot be distinguished using any of the functions exported from the structure. In order for the above structure to match the signature, axioms in EML are interpretated *up to behavioural equivalence*, hence making the axiom hold. We shall ignore this difficulty in the remainder.

Auxiliary functions are often needed in order to specify some function, and we want these auxiliary functions to be local. The '`local` $spec_1$ `in` $spec_2$ `end`' construct provides a means for introducing auxiliary functions in a signature such that a matching structure is not forced to export these also.

As an example, consider the following:

```
signature SORT =
  sig
    val sort : int list -> int list
    local
      val ordered : int list -> bool
      axiom ordered(l) ==
              forall (l1,a,b,l2) =>
                l=l1@[a,b]@l2 implies a<=b
    in
      axiom ordered(sort(l))
    end
  end
```

The signature just exports the `sort` function, and this, in turn, is specified using the auxiliary function `ordered`.

We have shown how axioms can be written as part of signatures. In fact, one can also write axioms in structures, and this provides the tool for a program development method based on stepwise refinement of modules. EML is called a *wide spectrum* language since programming constructs and specification constructs can be mixed. We shall not go into detail about this aspect of EML, although it is regarded as an essential idea in EML. That is, our emphasis will just be on the possibility of writing axioms in signatures. Hence, we have a simplified *two-level* view of EML: signatures with axioms, and then structures with programs. We regard this simplification as theoretically acceptable, since our goal in the first place is simply just to define a logic for CML. It additionally seems to be a practically meaningful simplification of EML.

8.1.3 CML

Concurrent ML (CML) is an extension of ML with FC-like concurrency primitives. The model behind CML is that of a multiset of expressions evaluated in parallel – communicating via synchronous message passing on typed channels. There is a *fork* operator for starting a new evaluation of an expression, thereby extending the multiset with the new expression. Any of the expressions in the multiset can fork new expressions. Expressions communicate with each other on typed channels, and channels can be dynamically created by a *channel* operator. Communication between expressions is synchronised rendezvous-style, and occurs provided one expression performs a *transmit* operation and another expression performs an *accept* operation on the same channel. The CML primitives are provided as a set of "functions" in the same way that the ML primitives for handling mutable cells are "functions". So there is no special syntax for concurrency. We write "functions" in quotes, since surely they have side effects. The above mentioned primitives have the following signatures:

```
type 'a chan
val channel  : unit -> '1a chan
val fork     : (unit -> 'a) -> unit
val transmit : 'a chan * 'a -> unit
val accept   : 'a chan -> 'a
```

Note that there is a type `'a chan` of channels. The `channel` function returns a new channel when applied to the unit value. It is weakly polymorphic as explained in [Rep92]. The function `fork` takes as argument a function f (with unit argument type) and starts the parallel evaluation of $f()$. The result of a forked function can never be obtained, so the result type of f is in fact not important, and is left polymorphic only in order to be less restrictive. The function `transmit` sends a value on a channel. The function `accept` reads a value from a channel given as argument, and returns the value as result. As an example, consider the following function:

```
fun coin() =
      let val c = channel() in
        fork(fn () => transmit(c,true));
        fork(fn () => transmit(c,false));
        accept c
      end
```

It defines a *coin* in that each time it is applied, it returns non-deterministically either `true` or `false`. This is obtained since the two expressions `transmit(c,true)` and `transmit(c,false)` will run in parallel, hence competing to send on the `c` channel. Which one gets access is non-deterministic[1].

CML provides a *choice* operator corresponding to the choice operator $p+q$ in FC and CCS. Since this operator has to be a function, it must have a type: it must take a collection of *communication-offers* and then perform a choice amongst these. For this purpose, a special type of communication-offers, called *events*, is introduced: `'a event`, and the choice function gets the type
`choose : ('a event) list -> 'a event`. An event represents a potential,

[1]In John Reppy's sequential implementation of CML the function will, however, always return `true`.

delayed, computation *that begins with a communication*. Note that in an analog way, a function in CML also represents a potential computation; however, it is not ensured to begin with a communication. For implementation reasons, as well as for theoretic reasons, it is convenient that the communication-offers are easily accessible, which is the reason for not just using general functions as arguments to `choose`.

We are now ready to present all the CML primitives used in this work. These are provided by a structure `CML` which has the following signature:

```
signature CML_SIG =
  sig
    type 'a chan
    type 'a event
    val channel : unit -> '1a chan
    val fork    : (unit -> 'a) -> unit
    val send    : 'a chan * 'a -> 'a event
    val receive : 'a chan -> 'a event
    val wrap    : 'a event * ('a -> 'b) -> 'b event
    val choose  : 'a event * 'a event -> 'a event
    val sync    : 'a event -> 'a
  end
```

The functions `send` and `receive` are the event-valued versions of `transmit` and `accept`. They are the basic functions for creating events. Events are *activated* by applying `sync` to them. In fact, the functions `transmit` and `accept` are defined as derived forms as follows:

```
transmit = fn (c,x) => sync(send(c,x))
accept   = fn c => sync(receive(c))
```

The function `wrap` makes it possible to "wrap" a post-synchronisation action around an event. For example, the following event reads (when activated) a temperature `t` from the channel `temp`, and then passes `t` as argument to the function `fn x => check(x)`, resulting in `check(t)`.

```
wrap(receive temp,fn x => check(x))
```

It holds that:

```
sync(wrap(receive temp,fn x => check(x))) =
  check(accept temp)
```

let us now see how the `choose` function works. As an example, consider the following. Assume three channels `temp`, `alarm` and `wakeup`, and the function `check`. Then a choice can be formulated as follows:

```
choose[
  wrap(receive temp, fn x => check(x)),
  wrap(receive alarm, fn x => (transmit(wakeup,x);false))]
```

The second argument to `choose` illustrates how the post-synchronisation action can perform further communication - note, however, this time we want it to be executed, hence: we use `transmit` and not `send`. A choice between the events is in fact not performed at this stage. First when `sync` is applied, a choice will be performed, depending on the wishes of the environment. The following derived form is useful whenever we want the choice to be performed right away:

```
select = fn ev_list => sync(choose(ev_list))
```

We shall finally present a full example, showing how asynchronous communication can be programmed using the CML primitives. The example is that of buffered channels. Buffered channels provide a means of asynchronous communication. We want a way of generating new buffered channels, sending values to these in a non-blocking way, and finally for receiving values from these, blocking if no value in the buffered channel. The signature is as follows:

```
signature BUFFERED_CHAN =
  sig
    type 'a buffer_chan
    val buffer        : unit -> '1a buffer_chan
    val bufferTransmit : ('a buffer_chan * 'a) -> unit
    val bufferAccept   : 'a buffer_chan -> 'a
    val bufferReceive  : 'a buffer_chan -> 'a event
  end
```

The structure is as follows:

```
structure BufferChan:BUFFERED_CHAN =
  struct
    datatype 'a buffer_chan =
               BC of {inch : 'a chan, outch : 'a chan}
    fun buffer() =
          let val inCh  = channel()
              and outCh = channel()
              fun loop([],[]) = loop([accept inCh],[])
                | loop(front as (x::r),rear) =
                    select[wrap(receive inCh,
                               fn y  => loop(front,y::rear)),
                             wrap(send(outCh, x),
                               fn () => loop(r,rear))]
                | loop([],rear) = loop(List.rev rear, [])
          in
            fork(fn () => loop([],[]));
            BC{inch=inCh, outch=outCh}
          end
    fun bufferTransmit(BC{inch,...},x) = transmit(inch,x)
    fun bufferAccept(BC{outch,...}) = accept outch
    fun bufferReceive(BC{outch,...}) = receive outch
  end
```

Note the pattern '`front as (x::r)`' occurring in the definition of `loop` (second clause): it works as the pattern `(x::r)`, and in addition, in case a value matches, it is bound to `front`. Note that `bufferReceive` returns an event. It can thus occur in a choice:

```
choose[wrap(bufferReceive(bc), fn x => ...),...]
```

8.2 A New Semantics for CML

In this section we shall outline an operational semantics for CML. The result can be seen as a natural extension of the semantics of the ψ-calculus in chapter 7. Note, however, that we do not pretend to fully define CML. That is, we shall not formally define a notion of equivalence. There already exist at least two operational semantics for CML, namely [BMT92] and [Rep91c, Rep92]. In both of these, however, the transition system is *unlabelled*[2]. What we need is a

[2] The transitions in [BMT92] are labelled with sets of process identifiers, but this is not what we want.

labelled transition system for CML, where labels are observable actions, since only this will allow us to define a Hennessy-Milner like modal logic based on action modalities. The semantics below resembles the semantics of Facile [PGM90, Pra91] in important ways. We also mention the semantics of C-Lisp defined in [Jen92] as related work. At the time of writing, a very recent *action semantics* of CML has been given in [MM93]. We shall not define a semantics for the whole of CML. Instead we cut out a theoretical subset, in fact the subset defined in [BMT92].

8.2.1 The Syntax of CML Expressions

The language we present is big enough to capture the problems that are interesting in our context. We shall only define the language of expressions, which is as follows:

$$e \ ::= \ x \mid e_1\, e_2 \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid \texttt{rec}\ f(x)\ \texttt{=>}\ e \mid (e_1, e_2)$$

The x and f are identifiers. There are the usual constructs such as application, lambda abstraction, recursive lambda abstraction, and pairing. Amongst the identifiers are a collection of predefined ones, of which we focus on the following set: $\{\texttt{()}, \texttt{send}, \texttt{receive}, \texttt{choose}, \texttt{wrap}, \texttt{channel}, \texttt{fork}, \texttt{sync}\}$. The fact that the concurrency primitives are represented as identifiers is a consequence of the fact that the CML concurrency primitives are provided as a module; hence, they do not extend the syntax of ML.

The syntax presented above is, however, not quite appropriate for our purpose, which is to provide an operational semantics. The operational semantics operate with syntactic terms as the intermediate states between transitions. The syntax must therefore capture any state that can arise during evaluation. This is not the case with the above syntax. In addition, we would like our syntax to separate out certain subclasses of expressions that have particular treatment in the operational semantics, such as for example values. We therefore define the following "semantic" syntax, generating in particular the languages of expressions $\mathcal{L}$, ranged over by e, values $\mathcal{V}$, ranged over by v, and events $\mathcal{E}$, ranged over by ev.

$$\begin{array}{rcl}
e & ::= & x \mid v \mid e_1\, e_2 \mid \texttt{rec}\ f(x)\ \texttt{=>}\ e \mid (e_1, e_2) \\
v & ::= & \texttt{()} \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid (v_1, v_2) \mid b \mid ec \mid ev \mid k \\
b & ::= & \texttt{channel} \mid \texttt{fork} \mid \texttt{sync} \\
ec & ::= & \texttt{send} \mid \texttt{receive} \mid \texttt{choose} \mid \texttt{wrap} \\
ev & ::= & <ec, v>
\end{array}$$

The category $\mathcal{V}$ contains the values v that cannot be further evaluated. Amongst the values a the basic concurrency primitives, b, which are `channel`, `fork` and `sync`. These are called basic since they seem very fundamental in CML in contrast to the event constructors of which one could consider a variety. The event constructors, ec, are those that return events, $ev \in \mathcal{E}$, when applied: an event is just a pair consisting of the event constructor and its argument. Note, that the term (v_1, v_2) is either an e or a v. The ambiguity is resolved in favor of v.

The category $\mathcal{E}$ of events ev is "semantic" in that a programmer cannot write phrases in this class. This is also true for the class *Chan* of channel names k, that are generated in connection with channel allocation.

We shall need to refer to the set $CV[e]$ of channels k occurring in an expression e. Its definition should be obvious. The channels that occur in a multiset P is obtained by including the channels in each expression.

8.2.2 Semantics

The semantics is as usual divided into two layers, one for expressions in isolation, and one for multisets of expressions running in parallel.

Expressions

We define the labelled transition system $(\mathcal{L}, Lab, \hookrightarrow)$. Concerning the set of labels *Lab*, this is defined as follows:

$$\begin{array}{lcl} Com & = & \{k?v \mid k \in Chan, v \in \mathcal{V}\} \cup \{k!v \mid k \in Chan, v \in \mathcal{V}\} \\ Act & = & Com \cup \{\tau\} \\ Lab & = & Act \cup \{\phi(v) \mid v \in \mathcal{V}\} \cup \{\lambda(k) \mid k \in Chan\} \end{array}$$

Before defining the transition relation $\hookrightarrow \subseteq \mathcal{L} \times Lab \times \mathcal{L}$, we define the predicate $\Longrightarrow \subseteq \mathcal{E} \times Com \times \mathcal{L}$, which defines the communication potential of events. The relationship $ev \stackrel{c}{\Longrightarrow} e$ intuitively means that the event ev has the potential of performing the communication c (when `sync` is applied to the event) and then it will continue as the expression e. Note, however, that the event is not able to perform the indicated action – `sync` has to be applied before this can happen. We say that the event *offers* the action. Events offer communications in *Com*, and not general labels in *Lab*: an event always begins with a communication.

In [BMT92] as well as in [Rep91c], these communication potentials of events are defined in terms of a relation that matches *pairs* of events (yielding pairs of expressions). Our model is more appropriate for our purpose, and we find it simpler.

Definition 8.1 (The predicate $\Longrightarrow$) *Let $\Longrightarrow$ be the smallest subset of $\mathcal{E} \times Com \times \mathcal{L}$ closed under the following rules:*

$$(\textbf{Send}) \qquad \frac{}{<\!\texttt{send}, (k, v)\!> \stackrel{k!v}{\Longrightarrow} \texttt{()}}$$

$$(\textbf{Receive}) \qquad \frac{}{<\!\texttt{receive}, k\!> \stackrel{k?v}{\Longrightarrow} v} \; v \in \mathcal{V}$$

$$(\textbf{Choose}_1) \qquad \frac{ev_1 \stackrel{c}{\Longrightarrow} e}{<\!\texttt{choose}, (ev_1, ev_2)\!> \stackrel{c}{\Longrightarrow} e}$$

$$(\textbf{Choose}_2) \qquad \frac{ev_2 \stackrel{c}{\Longrightarrow} e}{<\!\texttt{choose}, (ev_1, ev_2)\!> \stackrel{c}{\Longrightarrow} e}$$

$$(\textbf{Wrap}) \qquad \frac{ev \stackrel{c}{\Longrightarrow} e}{<\!\texttt{wrap}, (ev, v)\!> \stackrel{c}{\Longrightarrow} v\ e}$$

The operational semantics of processes is then as follows.

Definition 8.2 (The transition relation $\hookrightarrow$) *Let $\hookrightarrow$ be the smallest subset of $\mathcal{L} \times Lab \times \mathcal{L}$ closed under the following rules:*

$$(\textbf{Application}_1) \qquad \frac{e_1 \stackrel{l}{\hookrightarrow} e_1'}{e_1\ e_2 \stackrel{l}{\hookrightarrow} e_1'\ e_2}$$

$$(\textbf{Application}_2) \qquad \frac{e \stackrel{l}{\hookrightarrow} e'}{v\ e \stackrel{l}{\hookrightarrow} v\ e'}$$

$$(\textbf{Beta}_1) \qquad \frac{}{(\texttt{fn}\ x\ \texttt{=>}\ e)\ v \stackrel{\tau}{\hookrightarrow} e[v/x]}$$

$$(\textbf{Beta}_2) \qquad \frac{}{ec\ v \stackrel{\tau}{\hookrightarrow} <\!ec, v\!>}$$

$$(\textbf{Recursion}) \qquad \frac{}{\texttt{rec}\ f(x)\ \texttt{=>}\ e \stackrel{\tau}{\hookrightarrow} \texttt{fn}\ x\ \texttt{=>}\ e[(\texttt{rec}\ f(x)\ \texttt{=>}\ e)/f]}$$

$$(\textbf{Pair}_1) \qquad \frac{e_1 \stackrel{l}{\hookrightarrow} e_1'}{(e_1, e_2) \stackrel{l}{\hookrightarrow} (e_1', e_2)}$$

$$(\textbf{Pair}_2) \qquad \frac{e \stackrel{l}{\hookrightarrow} e'}{(v, e) \stackrel{l}{\hookrightarrow} (v, e')}$$

$$(\textbf{Channel}) \qquad \frac{}{\texttt{channel}\,() \overset{\lambda(k)}{\hookrightarrow} k}\ k \in \mathit{Chan}$$

$$(\textbf{Fork}) \qquad \frac{}{\texttt{fork}\,v \overset{\phi(v)}{\hookrightarrow} ()}$$

$$(\textbf{Sync}) \qquad \frac{ev \overset{c}{\Longrightarrow} e}{\texttt{sync}\,ev \overset{c}{\hookrightarrow} e}$$

The rules should be fairly simple to read.

Configurations

A program is a multiset of expressions. We let $\mathcal{M}$ denote the set of programs ($\mathcal{M} = MS(\mathcal{L})$). As usual, we define a K-configuration $K \triangleright P$ to consist of a record K (containing all channels allocated up to a certain point) and a program P.

$$\mathit{KCon} \overset{def}{=} \{K \triangleright P \in \mathit{Record} \times \mathcal{M} \mid CV[P] \subseteq K\}$$

We need the function $\mathit{Msg} : \mathit{Act} \to \mathcal{P}(\mathit{Chan})$ for defining the transition relation. It extracts the set of channels that are transmitted in a communication:

$$\begin{array}{rcl} \mathit{Msg}(\tau) & = & \{\} \\ \mathit{Msg}(k?v) & = & CV[v] \\ \mathit{Msg}(k!v) & = & CV[v] \end{array}$$

The semantics of K-configurations is given in terms of the labelled transition system $(\mathit{KCon}, \mathit{Act}, \longrightarrow)$. The transition relation $\longrightarrow\ \mathit{KCon} \times \mathit{Act} \times \mathit{KCon}$. is defined as follows:

Definition 8.3 (The transition relation $\longrightarrow$) *Let $\longrightarrow$ be the smallest subset of $\mathit{KCon} \times \mathit{Act} \times \mathit{KCon}$ closed under the following rules:*

$$(\textbf{Action}^{\{\!|\,|\!\}}) \qquad \frac{e \overset{\alpha}{\hookrightarrow} e'}{K \triangleright \{\!|e|\!\} \overset{\alpha}{\longrightarrow} K \cup \mathit{Msg}(\alpha) \triangleright \{\!|e'|\!\}}$$

$$(\textbf{Channel}^{\{\!|\,|\!\}}) \qquad \frac{e \overset{\lambda(k)}{\hookrightarrow} e'}{K \triangleright \{\!|e|\!\} \overset{\tau}{\longrightarrow} K \cup \{k\} \triangleright \{\!|e'|\!\}}\ k \notin K$$

$$(\textbf{Fork}^{\{\!|\,|\!\}}) \qquad \frac{e \overset{\phi(f)}{\hookrightarrow} e'}{K \triangleright \{\!|e|\!\} \overset{\tau}{\longrightarrow} K \triangleright \{\!|e', f()|\!\}}$$

$$(\textbf{Communication}^{\{\!|\,|\!\}}) \quad \frac{e_1 \overset{k!v}{\hookrightarrow} e_1',\ e_2 \overset{k?v}{\hookrightarrow} e_2'}{K \triangleright \{\!|e_1, e_2|\!\} \overset{\tau}{\hookrightarrow} K \triangleright \{\!|e_1', e_2'|\!\}}$$

$$(\textbf{Isolation}^{\{\!|\,|\!\}}) \quad \frac{K \triangleright P_1 \overset{\alpha}{\rightarrowtail} K' \triangleright P_1'}{K \triangleright P_1 \cup P_2 \overset{\alpha}{\rightarrowtail} K' \triangleright P_1' \cup P_2}$$

Note in the ($\textbf{Action}^{\{\!|\,|\!\}}$) rule that the channels transmitted are added to the set K. This is really just to cover the case of an input action $k?v$ where some channels in v are not already in K.

We finally extend configurations with windows as we did for the Ψ-calculus. The idea comes from the Facile semantics [PGM90], where the notion of equivalence is indexed with windows. Although the basic idea is the same, we instead embed the windows in the semantics, rather than index the equivalence relation with windows. We define $\mathit{Window} \stackrel{def}{=} \mathcal{P}(\mathit{Chan})$. We let W range over Window. The window may change throughout the execution of a program: when some expression outputs local channels, the window is extended with these channels. A window together with a record is referred to an an *environment*: $\mathit{Env} \stackrel{def}{=} \mathit{Window} \times \mathit{Record}$.

A configuration $(W, K) \triangleright P$ consists of an environment (W, K) and a program P. We shall only consider well formed configurations, where the window (and the set of free channels in the program) is a subset of the record:

$$\mathit{Con} \stackrel{def}{=} \{(W, K) \triangleright P \in \mathit{Env} \times \mathcal{M} \mid (W \cup \mathit{CV}[P]) \subseteq K\}$$

The predicate $\_\ \mathit{allows}\ \_ : \mathit{Env} \times \mathit{Act}$ is defined as follows:

$$\begin{array}{lcl} (W, K)\ \mathit{allows}\ \tau & = & \mathit{true} \\ (W, K)\ \mathit{allows}\ k?v & = & k_1 \in W \ \wedge\ \mathit{CV}[v] \cap (K - W) = \emptyset \\ (W, K)\ \mathit{allows}\ k!v & = & k_1 \in W \end{array}$$

$(W, K)\ \mathit{allows}\ \alpha$ holds if the channel transmitted upon is in the window, and in the case of an input: the transmitted channels are not local.

The semantics of configurations is given in terms of the labelled transition system $(\mathit{Con}, \mathit{Act}, \longrightarrow)$. The dynamic behaviour of configurations is defined as follows.

Definition 8.4 (The transition relation $\longrightarrow$) *Let $\longrightarrow$ be the smallest subset of $\mathit{Con} \times \mathit{Act} \times \mathit{Con}$ satisfying the following rule:*

$$(\textbf{Config}) \quad \frac{K \triangleright P \overset{\alpha}{\rightarrowtail} K' \triangleright P'}{(W, K) \triangleright P \overset{\alpha}{\longrightarrow} (W \cup \mathit{Msg}(\alpha), K') \triangleright P'}\ (W, K)\ \mathit{allows}\ \alpha$$

8.2.3 Relationships with Similar Semantics

In this section we shall shortly relate the presented semantics with four other similar semantics: the two existing semantics of CML [BMT92] and [Rep91c], the semantics of Facile [PGM90], and the semantics of C-Lisp [Jen92]. All of these semantics have influenced the presented semantics.

First, concerning the transition system for configurations, our system is labelled with observable actions. This is not the case for the other two CML semantics, and this was the basic reason for writing a new CML semantics. Our semantics is quite similar to the one in [PGM90] since their transitions of behaviour expressions are labelled with observable actions, and behaviour expressions can be said to be a syntactic representation of multisets. C-Lisp configuration transitions are not labelled.

Second, our semantics is divided into two layers, resulting in two transition systems: one for individual processes (expressions), and one for configurations. Again, this is not the case for the two CML semantics, but it is the case for the Facile semantics, and for the C-Lisp semantics. The advantage of two levels is that as much compositionality is factored out as possible: the first level is compositional in the same way as is the semantics of CCS. We also think that the rules become clearer. Consider for example the following transition rule from [BMT92], which defines how an application expression $e_1\, e_2$ is evaluated:

$$\frac{K, P[p : e_1] \xrightarrow{S} K', P'[p : e_1']}{K, P[p : e_1\, e_2] \xrightarrow{S} K', P'[p : e_1'\, e_2]}$$

$P[p : e]$ is a mapping from expression identifiers to expressions, where p is mapped to e. S is the set of identifiers of expressions involved in the transition. The semantics of $K, P[p : e_1\, e_2]$ is not given purely in terms of the semantics of its subcomponents, but rather: e_1 is "picked out", and then "inserted in" the context $K, P[p : e_1]$ which is then evaluated (above the line). What kind of semantics to prefer is certainly debatable since two levels also yield a further complication. Note that, since at our first level there is no K-component, channel allocation yields a special label that is then dealt with at the second level. This is different from the Facile semantics, where the K-component already occurs in the first level. In order to make the first level really simple, we have pushed also the K-component into the second level.

Third, concerning the treatment of events, our approach is different from the one taken in the other two CML semantics (there are no events in Facile or C-Lisp). In [BMT92] (and similarly in [Rep91c]), a relation between pairs of events and pairs of expressions is defined, defining which events can communicate with each other, and what the resulting expressions of the communication are. In our treatment, the semantics of an event is given in isolation without considering the interacting event. This seems to be convenient in order to make configuration transitions labelled: at the outermost level, there is not an interacting event.

Fourth and finally, our global rules for multisets differ from the ones for the corresponding behaviour expressions in [PGM90] in that we have only one rule with a configuration transition in the premise. This rule corresponds to the chemical law in [BB90] which says how a subset of a solution can react freely on its own. In [PGM90] also the rule for communication has configuration transitions in the premise.

Chapter 9

Extended CML

We shall now extend CML into a combined specification and programming language. We shall name this language ECML (Extended CML) in order to emphasize the parallel to EML (Extended ML). Just as EML is a small, natural and elegant extension of ML, we claim that our extension of CML can be characterised to have the same properties.

Various logics have been designed for specifying value passing processes. In [Mil89] a CCS value passing calculus is defined as a derived form of a basic CCS calculus without value passing. The values that can be communicated are simple values like integers etc. The Hennessy-Milner Logic for the basic form then consequently applies to the value passing calculus. It turns out, that our logic is closest to this work in the sense that the obtained logic is "just" a straight forward combination of predicate logic and modal logic. Another recent development is [HL93d] which is also quite related to our logic. Finally, [CR91] defines a logic based on temporal logic, but the ideas are still very relevant to the present work, it is just a combination of predicate logic and temporal logic. In addition, any sort of value can be communicated, for example processes.

As a different track we can mention the logics defined in [IT91] for value passing CCS, [Tho90] for CHOCS and [LT92a] for Facile. These logics are characterised by having binary modalities. For example, a process p satisfies $<c!>(F,G)$ if p on channel c can output a value satisfying F, and then continue to satisfy G. The language of F predicates depends on what values can be communicated. In [IT91] it is just simple values like integers. In [Tho90] it is solely processes, while in [LT92a] it is any kind of values, including channels and processes. We find that binary modalities are unnecessary, but admit that it is a matter of choice.

Finally, we should not forget the logic in [MPW91] from which we have obtained a technique for specifying communication of channels.

Note that although the process specification language LOTOS [BB89] combines process algebra with datatype specification, there is no notion of a modal or temporal logic. This is also the case for [Kap89, KP87].

Section 9.1 introduces ECML, in terms of a formal syntax and an informal tutorial. We in fact suggest a fully-fletched realistic ECML language, rather than a theoretic ECML calculus. This gives us the possibility to write real examples, and makes an example-driven language design possible. Section 9.2 outlines a semantics of ECML. Section 9.3 illustrates the logic through seven examples. Finally, section 9.4 ends the chapter with a discussion of some design decisions and their consequences.

9.1 Syntax and Informal Semantics

In this section we shall describe a proposal for an ECML syntax. ECML includes EML (Extended ML) by including basically two constructs from EML for specifying purely applicative programs: a binary equivalence predicate and universal quantification. Our contribution is to add modal logic to this. That is, if we take a look at process algebra, then we must add an equivalent to the statement $p \models \psi$ where p is a process and ψ is a (modal) formula. Recall that the modal formulae ψ for the ψ-calculus were $\boldsymbol{tt}$, $\psi_1 \wedge \psi_2$, $\neg\psi$, $[x = y]\psi$ and $<\alpha>\psi$. All of these, except the modality $<\alpha>\psi$ are already expressible in ML as we shall see later, so as a second construct, we must add an equivalent to the modality $<\alpha>\psi$. Finally, in order to specify infinite behaviour, we add a fixpoint construct corresponding to $\nu X \cdot \psi$ from the refinement calculus.

The basic idea is the same as in EML: to the category of *spec* which is the contents of a signature expression, we add the notion of an axiom, which is just a boolean expression, see figure 9.1.

spec	::=	...
		`axiom` *exp*

Figure 9.1: Syntax for ECML specifications

Any expression of type `bool` may be used as an axiom in EML. Such use amounts to an assertion that the expression terminates and evaluates deterministically to the value `true` (rather than to the value `false`, an exception, or a communication).

In order to obtain specification power, we add to the category of expressions a selection of specification constructs. We go right ahead, and present the full syntax for ECML *bare* expressions in figure 9.2, that is: it is an extension of the

ML bare language for expressions where the ML bare language is the minimal kernel into which all derived forms of expressions in ML are mapped.

$$
\begin{array}{lll}
exp & ::= & atexp \\
 & & exp\ atexp \\
 & & exp_1\ id\ exp_2 \\
 & & exp : ty \\
 & & exp\ \texttt{handle}\ match \\
 & & \texttt{raise}\ exp \\
 & & \texttt{fn}\ match \\
 & & exp_1\ \texttt{==}\ exp_2 \\
 & & \texttt{forall}\ match \\
 & & exp_1\ \texttt{|-}\ exp_2 \\
 & & \texttt{<:}exp\ dir\texttt{:>}match \\
 & & \texttt{<::>}exp \\
 & & \texttt{max}\ id\ match
\end{array}
$$

Figure 9.2: Syntax for ECML bare expressions

The first seven alternatives make up the syntax of ML bare expressions. We shall not go into details about that part, except for saying that amongst the atomic expressions *atexp* are constants, identifiers, let-expressions and record expressions. The other ML expressions are: function application, infix-operator application, typed expressions, exception handling and raising, and finally lambda abstraction.

After the ML expressions follow our addition: equivalence and universal quantification (from EML), process-formula satisfaction, two modalities and finally the maximal fixpoint construct. These constructs are explained below. The category *dir*, defined as $dir ::= ?|!$, and used in one of the modalities, indicates the direction of a communication: input or output.

The EML Constructs

The following two constructs come from EML, equivalence and universal quantification:

$$
\begin{array}{l}
exp_1\ \texttt{==}\ exp_2 \\
\texttt{forall}\ match
\end{array}
$$

The "logical" equality predicate, or equivalence predicate as we shall call it, is provided to complement the "computational" equality `=` provided by ML. The expression $exp_1 \texttt{==} exp_2$ is well-formed whenever exp_1 and exp_2 have the same type, in contrast to exp_1 `=` exp_2 which requires this to be an equality type. Also, in ML exp_1 `=` exp_2 is really the application of the function '`=`' to the pair (exp_1, exp_2), and with the call by value semantics, it means for example that the expression `transmit(c,5) = transmit(c,7)` would offer two communications before evaluating to `true` (both sub-expressions return `()`). This we cannot be satisfied with, so, as in EML, `==` is introduced such that for example `transmit(c,5) == transmit(c,7)` offers no communication and evaluates to `false`. Put differently, `==` is not a function, but a syntactic construct in the language.

Equivalence is extensional equivalence on function types, meaning that if f and g are both of type $ty_1 \rightarrow ty_2$, then f`==`g is defined as `forall`$x : ty_1$`=>`$f(x)$`==`$g(x)$. It is also extensional for exceptions, non-termination and (in our extended version) communication. For example, expressions are equivalent if they are bisimular in the sense discussed in section 8.2.2. So for example exp `==` exp is `true` even if exp raises an exception, communicates or fails to terminate. As examples of desired equivalences we mention the following three:

```
transmit(c,5) == sync(choose[send(c,5),send(c,4+1)])

9 == let val c : int chan = channel() in
        fork(fn () => transmit(c,9));
        accept c
     end

fork(f);fork(g) == fork(g);fork(f)
```

Note that these equivalences can only hold if τ-actions are not observable. See section 3.3 for a treatment of τ-actions as not observable in the context of FC.

Concerning universal quantification, we can write quantified formulae such as for example `forall` $x : ty$ `=>` exp. In case ty is a known type such as `int` the interpretation is relatively obvious, disregarding the possibility of exp to raise exceptions etc, fail to terminate, etc. But in case ty is a polymorphic type variable `'a` it gets more problematic: what to quantify over? We shall not go into a deep discussion about the interpretation of quantification, since it is not our objective in this presentation to be fully formal. In EML a quantified formula has a defined value `true` only, if it has that value for all instances of the type of the bound variable. If no type for the bound variable is indicated, as for example in `forall` x `=>` exp then we can associate a type, perhaps polymorphic, so in any case we can think of quantifications as having the form `forall` $x : ty$ `=>` exp. The meaning is as follows: in order for `forall` $x : ty$ `=>` exp to be `true`, the expression $exp[v/x]$ must evaluate to `true` for every expressible value v of every instance of ty. Note that we only range over the values of a type that are expressible in ML (as an expression).

The Modal Constructs

The basic two constructs for specifying communicating expressions are the 'process-satisfies-formula' predicate and the 'can do' modality, the latter representing an existential quantification over transitions:

$$\mathit{exp}_1 \texttt{ |- } \mathit{exp}_2$$
$$\texttt{<:}\mathit{exp}\ \mathit{dir}\texttt{:>}\mathit{match}$$

These two expressions are in some way related to the expressions exp_1 `==` exp_2 and `forall` *match* for the applicative part of the language. The first expression exp_1 `|-` exp_2 further corresponds to the $p \models \psi$ statements in traditional process algebra (the exp_1 corresponds to p and exp_2 corresponds to ψ), and it states that the process represented by exp_1 satisfies the formula represented by exp_2. We require that exp_1 is a CML expression not containing any specification constructs. Typically exp_2 will contain one or more modalities, where a modality is of the form `<:`*exp dir*`:>`*match* (the second alternative above). Here *exp* evaluates to a channel, and *dir* indicates a direction (input or output) of communication. The *match* has the form pat_1`=>`$\mathit{exp}_1|\ldots|\mathit{pat}_n$`=>`$\mathit{exp}_n$. Take the simplest case where the channel valued expression is just the name of a channel and where the match has the form x`=>`*exp*: Then there are two kinds of modalities corresponding to input and output. The modality:

$$\texttt{<:}c!\texttt{:>}x \texttt{ => } \mathit{exp}$$

says that a value, name it x can be output on the channel c, where after the formula *exp* holds. The name x is a pattern in ML's sense, so it is a binding occurrence with *exp* as scope. The output value is matched against the pattern x, and then *exp* must hold for the remainder of the process, and within the scope of this binding. As an example, consider the formula:

```
<c!>x => x>0
```

It states that "the process" can output an integer `x` on the channel `c` where `x` is greater than zero.

Similarly, the modality:

$$\texttt{<:}c?\texttt{:>}x \texttt{ => } \mathit{exp}$$

says that a value, name it x, can be input on the channel c, where after the formula *exp* holds. As an example, consider the formula:

```
<c?>x => (<c!>y => y==sqrt(x))
```

It states that "the process" can input a real number `x` on the channel `c`, and then output a real `y`, also on `c` where `y` is the square root of `x` (`sqrt` is part of ML's standard functions).

As a complete example, let us consider the specification of a structure which provides a function that satisfies the latter formula. That is, we want to provide a structure that satisfies the following signature:

```
signature SQUARE =
  sig
    val square : real chan -> unit
    axiom forall c : real chan =>
            square(c) |- <c?>x => (<c!>y => y==sqrt(x))
  end
```

The function `square` takes as parameter a channel, and then behaves as specified by the axiom. The expression `square(c) |- ...` states that the expression `square(c)` satisfies the "formula" `...`. An example of a structure satisfying this signature is the following:

```
structure Square:SQUARE =
  struct
    fun square(c) = let val r = accept c in
                      transmit(c,sqrt(r));
                      square(c)
                    end
  end
```

In case a modality contains several entries in the *match*: pat_1`=>`$exp_1|\ldots|pat_n$`=>`exp_n, the intended meaning is a disjunction. For example. `<:`$c!$`:>1 =>` P_1 | `2 =>` P_2 has the same meaning as `<:`$c!$`:>1 =>` P_1 `orelse` `<:`$c!$`:>2 =>` P_2.

The other modality

`<::>`*exp*

makes it possible to state properties about internal transitions. The formula says that "the process" can perform some internal computation, and thereby reach a state where *exp* holds. This modality seems necessary in addition to the other one mentioned, which we shall see in a later example.

Recursion

A more appropriate specification of the structure `Square` may indicate the recursive nature of the function `square`. For this purpose we finally introduce a maximal fixpoint construct corresponding the the $\nu X \cdot \psi$ construct in our refinement logic:

`max` *id match*

Recall that a lambda abstraction in ML has the form `fn` *match*. Here we have just added an identifier *id* to name the recursion. As a very simple example consider the following:

```
max X() => (<c!>_=>X())
```

This formula specifies a process that continuously can output a value (we dont care which, indicated by the wildcard pattern _) on the channel `c`. The corresponding formula in process algebra (ignoring the communicated value) would be something like $\nu X \cdot <c!> X$. Note that a maximal fixpoint is parameterised, however in the above case just with the unit value. In general we can write maximal fixpoints like for example the following:

```
max INC(i) => (<c!>j => j>i andalso INCR(j))
```

This specifies a behaviour where increasing integers are continuously output on the channel `c`. The `andalso` construct is logical conjunction defined as a primitive in ML. Note that we have, however, not indicated which integer to begin with. Therefore, such a maximal fixpoint construct must be "applied" to an initial value in order to be effective (to really obtain the maximal fixpoint). The following formula instantiates the maximal fixpoint to zero:

```
(max INC(i) => (<c!>j => j>i andalso INCR(j)))(0)
```

We will later introduce derived forms for writing these fixpoints more conveniently, just like the other constructs that are defined as derived in ML. If we return to the complete example, the signature can now be written as follows (still with the inconvenient notation of defining and then applying the maximal fixpoint construct):

```
signature SQUARE =
  sig
    val square : real chan -> unit
    axiom forall c : real chan =>
            square(c) |-
              (max REPEAT() =>
                (<c?>x => (<c!>y =>
                  (y==sqrt(x) andalso REPEAT()))))()
  end
```

The identifiers `return` and `NEW`

Two identifiers have been reserved, one (`return`) dealing with the termination of an expression, and one (`NEW`) dealing with scope extrusion: the sending of a channel that has been locally allocated, and which therefore is new in the receiving context.

Concerning termination, recall, from the Fork Calculus, that the special event π signaled the termination of the main process under observation. One could in the logic for the Fork Calculus write a formula like $<\pi>\,\psi$ meaning that the process could terminate, and what then remained as forked would then satisfy ψ. We need a similar concept in our logic for CML. The situation in CML is slightly more complicated since expressions also return values, so our π event must additionally carry a value. The idea is that when analysing an expression *exp*, we do not observe '$exp; \pi$' (as we did in the Fork Calculus), but rather, we observe '`sync`(`send`(`return`, *exp*))', where `return` is the reserved channel name. So, when we write exp_1 `|-` exp_2, we (slightly simplified) require that the process `sync`(`send`(`return`, exp_1)) satisfies exp_2. This idea of observing $r!exp$ for some channel r is used in the Facile semantics [PGM90] to define an equivalence between expressions. We can thus write axioms like:

```
1 |- <:return!:>x => x>0
```

meaning that the expression `1` can terminate, returning an integer greater than zero. Note that every expression in CML is a "process", so the above axiom makes formally sense. Of course it makes no sense to use `return` in an input modality. As a more interesting example, consider the following function definition:

```
fun f(c:int chan,x:int) = (fork(fn () => transmit(c,x));
                           x+1)
```

This function satisfies the following formula, assuming a universal quantification over the free variables at the outermost level:

```
axiom f(c,x) |- <:return!:>y => (y==x+1 andalso <:c!:>z => z==x)
```

The second reserved name is `NEW`, and it is to be thought of as a constructor of type `NEW` : $'a$ `chan` $\rightarrow$ $'a$ `chan`. Whenever an expression (for the first time) sends a channel c that has been locally declared, we may think of the value `NEW`(c) being sent instead. This means that we are able to specify scope extrusion where a process sends a new channel to the environment. Recall that in the ψ-calculus we had modalities of the form $<c!(y)>\ldots$, where y was a binding occurrence (indicated by the brackets (...)) representing a new channel being output. This in contrast to the modality $<c!k>$ ... where k was some particular channel, already known in the context. In our case we can write two similar formulae like:

```
<:c!:>NEW(y) => ...
<:c!:>y => (y==k andalso ...)
```

Note here that `NEW(y)` is a pattern, where `y` is a binding occurrence. As a final example, let us modify our requirements to the structure `Square`:

```
signature SQUARE =
  sig
    val square : unit -> real chan
    axiom square() |- <:return!:>NEW c =>
                        (max REPEAT() =>
                          ((<c?>x => (<c!>y =>
                            (y==sqrt(x)
                             andalso REPEAT())))))()
  end
```

We now require that the `square` function when applied to the unit value immediately terminates and returns a new channel. After termination, the behaviour of what has been forked is as before: repeated input and then output of the square root. A structure satisfying this requirement is the following:

```
structure Square:SQUARE =
  struct
    fun square() =
          let
            fun doit(ch) = let val r = accept ch in
                             transmit(ch,sqrt(r));
                             doit(ch)
                           end
            val c:real chan = channel()
          in
            fork(fn () => doit c);
            c
          end
  end
```

Derived forms

In this section we shall define a set of useful derived forms. The main set of derived forms are presented in figure 9.3.

1	`case` *exp* `of` *match*	`(fn` *match*`)(`*exp*`)`
2	`if` exp_1 `then` exp_2 `else` exp_3	`case` exp_1 `of` `true =>` exp_2 $\mid$ `false =>` exp_3
3	exp_1 `andalso` exp_2	`if` exp_1 `then` exp_2 `else false`
4	exp_1 `orelse` exp_2	`if` exp_1 `then true else` exp_2
5	`not` *exp*	`if` *exp* `then false else true`
6	exp_1 `implies` exp_2	`(not` exp_1`) orelse` exp_2
7	`exists` *match*	`not forall` *NEGATE*[*match*]
8	`[:`*exp dir*`:]`*match*	`not <:`*exp dir*`:>`*NEGATE*[*match*]
9	`[::]`*exp*	`not <::>not` *exp*
10	`<:@:>`*exp*	`(exists c =>` `(<c!>_ =>` *exp* `orelse <c?>_ =>` *exp*`)` `orelse` `<::>`*exp*
11	`[:@:]`*exp*	`not <:@:>not` *exp*
12	`return` *match*	`<:return!:>`*match*
13	`min` *id match*	see below
14	`always` *exp*	`(max` *X*`() =>` *exp* `andalso [:@:]`*X*`())()`
15	`eventually` *exp*	`(min X() =>` *exp* `orelse` `(<:@:>true andalso [:@:]X()))()`

Figure 9.3: Basic derived forms of expressions

The forms $1, 2, 3, 4$ are derived forms already part of ML. The rest are special for ECML. The forms $5, 6, 7$ are just the standard derivations from logic. We have used a function *NEGATE* on match terms. The effect is to negate the expressions occurring, as for example *NEGATE* applied to the match `x => (x==0)` yields `x => not (x==0)`. The forms $8, 9, 10, 11$ are derived modalities. An example of 8 is `[:c!:]x => x>0`, which states that *if* a value `x` is output on `c`, then it is greater than zero. Also, `[:c!:]_ => false` means that it is not possible to output some value on `c`. 9 means that if an internal computation is performed, then *exp* holds after. 10 means that some transition can be performed (either a single communication or internal computation) such that *exp* holds after. 11 means that if some transition is performed, then *exp* holds after. 13 defines the minimal fixpoint from the maximal fixpoint and negation, just like we have seen in our process algebra. Recall that $\mu\, x \cdot \psi[x] \stackrel{def}{=} \neg\nu\, x \cdot \neg\psi[\neg x]$. A similar definition is intended here. 14 and 15 are the temporal operators also known from temporal logic.

Even though we have ignored the question of specifying exceptions in connection with concurrency, let us repeat the derived forms in EML for specifying exceptions. These derived forms are given in figure 9.4.

exp `raises` *match*	((*exp*; `false`) `handle` *match*) `handle _ => false`
exp `raises` *pat*	*exp* `raises` *pat* `=> true`

Figure 9.4: Derived exception predicates

As examples taken from [SdST90] consider the following:

exp `raises _`	*exp* raises an exception
exp `raises` *excon*	*exp* raises the nullary exception *excon*
exp `raises` *excon* `0`	*exp* raises *excon* with the value `0`
exp `raises` *excon* `_`	*exp* raises *excon* with some value
exp `raises` *excon* `x => x >= 0`	*exp* raises *excon* with a non-negative value

EML contains two predicates for stating properties about termination of an expression. The expression *exp* `terminates` is `true` if *exp* produces a normal value or an exception, and `false` if it fails to terminate. The expression *exp* `proper` is `true` if *exp* produces a normal value, and `false` if it raises an exception or fails to terminate. We can define these as derived forms as in figure 9.5.

exp `proper`	`exists x =>` (*exp* `== x`)
exp `terminates`	(*exp* `proper`) `orelse` (*exp* `raises _`)

Figure 9.5: Derived termination predicates

Finally, we promised derived forms for writing maximal (and minimal) fixpoints. We do this in figure 9.6, by adding two alternatives to the category of specifications, *spec*, in addition to the axiom-alternative:

As an example, the *spec*:

```
max INCR(i) = (<c!>j => j>i andalso INCR(j))
```

is short for:

`max` *fvalbind*	see below
`min` *fvalbind*	see below

Figure 9.6: Derived forms of *spec*

```
val INCR : int -> bool
axiom INCR == max INCR(i) => (<c!>j => j>i andalso INCR(j))
```

So we can define a function that outputs increasing integers on a channel given as a parameter:

```
signature INCREASE =
  sig
    val f : int chan -> unit
    local
      max INCR(c,i) = (<c!>j => j>i andalso INCR(c,j))
    in
      axiom f(c) |- INCR(c,0)
    end
  end
```

9.2 Sketching a Semantics

In this section, we shall sketch a formal semantics of the presented CML logic. Note, that it is very much a *sketch*, but it should give an intuition about the intended model. The semantics of ECML does not conform strictly to the semantics of EML [KST93, SdST90], though the basic principles are followed.

First of all, we shall concentrate on the following expressions, which are the typical axioms:

$$e_1 \texttt{ == } e_2$$
$$\texttt{forall}\ x : ty \texttt{ => } e$$
$$e_1 \texttt{ |- } e_2$$
$$\texttt{<::>}e$$
$$\texttt{<:}c!\texttt{:>}x \texttt{ => } e$$
$$\texttt{<:}c!\texttt{:>NEW}(x) \texttt{ => } e$$
$$\texttt{<:}c?\texttt{:>}x \texttt{ => } e$$

Note that e, e_1 and e_2 range over all expressions, and not just over the ones listed here. For example in e_1 `|-` e_2, the expression e_1 must be a pure CML expression.

So how are these axioms interpretated? To answer this question, we must take a look at the module level. A pair of matching signature and structure declarations of the form:

`signature` *SIG* `= sig` *spec* `end`
`structure` *Str* : *SIG* `= struct` *dec* `end`

gives rise to the proof obligation *Str* : *SIG*. Let the environment E be the denotation of *dec*. The proof obligation then corresponds to verifying that the environment *satisfies* the specification: $E \models spec$. This involves for each axiom `axiom` e in *spec*, to verify that the expression e evaluates to `true` in (some modification E' of) the environment E, which we could write as $E' : e \Longrightarrow$ `true`. The evaluation relation $\Longrightarrow$ is different from the one used for CML programs: it is only defining the meaning of axioms. This is the style applied in the semantics of EML.

Rather than play the game with environments, we shall simplify a bit, and just play a mostly syntactic game. We shall assume that all the bindings in the structure declaration *dec* are substituted into the axiom e, yielding e', and then verify that e', which is a closed term with no free variables, is valid, which we write as $\models e'$. As an example, consider the following matching signature and structure declarations:

```
signature SEND =
  sig
    val v : int
    val f : int chan -> unit
    axiom forall c:int chan =>
            f(c) |- <:c!:>x => x==v
  end

structure Send:SEND =
  struct
    val v = 5
    fun f(c) = transmit(c,v)
  end
```

Verifying `Send:SEND` reduces to verifying the validity of the closed expression:

```
forall c:int chan =>
  transmit(c,5) |- <:c!:>x => x==5
```

The validity $\models e$ of an expression e is generally checked in the context of a configuration, which initially is empty. We shall thus rather check statements of the form $C \models e$ where C is a configuration $(W, K) \triangleright P$ as this was introduced in section 8.2: K is the set of already allocated channels, W is the window containing all the channels that are visible to the context, and P is the multiset of expressions running in parallel. Further, we shall assume that $\models e$ iff $I \models e$, where I is the initially empty configuration $I \stackrel{def}{=} (\{\}, \{\}) \triangleright \{\!|\,|\!\}$. The need for configurations arises since, as we shall see, the formula e_1 `|-` e_2 "forks" e_1 into the configuration, where after e_2 is then checked in the extended configuration.

We need some notation. Let C be the configuration $(W, K) \triangleright P$, and let e be an expression. Then we define:

$$C[r!e] \stackrel{def}{=} (W \cup \{r\}, K \cup \{r\}) \triangleright P \cup \{\!|\texttt{sync}(\texttt{send}(r, e))|\!\}$$

Further, we use $e[v/x]$ for traditional substitution of all free occurrences of x with v. $e\{r/\texttt{return}\}$ is as $e[r/\texttt{return}]$ except that no substitution is performed in a sub-expression of the form: e_1 `|-` e_2. This is because each occurrence of `|-` "introduces" a new return channel. That is, the following for example holds:

$$\begin{array}{rcl}
\texttt{return}\{r/\texttt{return}\} & \stackrel{def}{=} & r \\
(e_1 \texttt{ == } e_2)\{r/\texttt{return}\} & \stackrel{def}{=} & e_1\{r/\texttt{return}\} \texttt{ == } e_2\{r/\texttt{return}\} \\
(e_1 \texttt{ |- } e_2)\{r/\texttt{return}\} & \stackrel{def}{=} & e_1 \texttt{ |- } e_2 \\
(\texttt{<:}c!\texttt{:>}x \texttt{ => } e)\{r/\texttt{return}\} & \stackrel{def}{=} & \texttt{<:}c[r/return]!\texttt{:>}x \texttt{ => } e\{r/\texttt{return}\}
\end{array}$$

The relation $\models \subseteq Con \times \mathcal{L}$ satisfies the following seven properties. Let C be the configuration $(W, K) \triangleright P$, and let r, k range over $Chan$ and v range over $\mathcal{V}$. Then:

$$\begin{array}{llll}
C & \models & e_1 \texttt{ == } e_2 & \text{iff} \quad e_1 \text{ and } e_2 \text{ are equivalent in the context of } C. \\
C & \models & \texttt{forall } x : ty \texttt{ => } e & \text{iff} \quad \forall v \in [\![ty]\!] \cdot C \models e[v/x] \\
C & \models & e_1 \texttt{ |- } e_2 & \text{iff} \quad C[r!e_1] \models e_2\{r/\texttt{return}\} \text{ for some } r \notin K \\
C & \models & \texttt{<::>}e & \text{iff} \quad C \stackrel{\tau^*}{\longrightarrow} C' \text{ for some } C', \text{ and } C' \models e \\
C & \models & \texttt{<:}c!\texttt{:>}x \texttt{ => } e & \text{iff} \quad C \stackrel{\tau^* c!v\, \tau^*}{\longrightarrow} C' \text{ for some } C' \text{ and } v \notin Chan - W, \\
 & & & \phantom{\text{iff}} \quad \text{and } C' \models e[v/x] \\
C & \models & \texttt{<:}c!\texttt{:>NEW}(x) \texttt{ => } e & \text{iff} \quad C \stackrel{\tau^* c!k\, \tau^*}{\longrightarrow} C' \text{ for some } C' \text{ and } k \notin W, \\
 & & & \phantom{\text{iff}} \quad \text{and } C' \models e[k/x] \\
C & \models & \texttt{<:}c?\texttt{:>}x \texttt{ => } e & \text{iff} \quad C \stackrel{\tau^* c?v\, \tau^*}{\longrightarrow} C' \text{ for some } C' \text{ and } v, \\
 & & & \phantom{\text{iff}} \quad \text{and } C' \models e[v/x]
\end{array}$$

The rule for e_1 `==` e_2 is not formal, since we have not defined what equivalence means for CML expressions. We can say, however, that the two expressions must be equivalent in the context of C. The rule for `forall` $x : ty$ `=>` e assumes that each type denotes a set of values. Note that we are a bit informal here since substituting a semantic value into a syntactic term does not yield a syntactic term. The rule for e_1 `|-`e_2 defines how the configuration is extended with $r!e_1$; and in e_2, every corresponding reference to `return` will instead refer to r (see above definition of `return`-substitution). The rule for `<::>`e says that the configuration can perform zero or more τ transitions, thereby ending in a state satisfying e. The rule for `<:`$c!$`:>`x `=>` e says that the configuration can output a value v on c (in between zero or more τ transitions), where v is not a local channel. The rule for `<:`$c!$`:>NEW(`x`)` `=>` e says that the output channel k must be local (not in W).

Let us extend the situation to deal with environments: with the semantics just outlined, checking that environment E satisfies an axiom e corresponds to verify that $(E, I) \models e$, with a corresponding definition of $\models$ to deal with environments. As we said earlier, a more correct approach is to say that "the expression e evaluates to `true` in (some modification E' of) the environment E, which we could write as $E' : e \implies$ `true`". The modification E' is basically (E, I), so in a full semantics, one would need to work with statements such as $(E, C) : e \implies$ `true` for configurations C. This is obligatory since axioms in general can be any boolean expression, and thus for example an expression of the form $e_1\, e_2$, which cannot be handled with our sketch of a semantics above. What would $C \models e_1\, e_2$ mean? We have avoided rules of the form $(E, C) : e \implies v$ since this seems to involve rules with negative statements about expression evaluation (caused by universal quantification and modalities), and in that case special care must be shown to get system of meaningful rules.

9.3 Example Applications

In this section we shall illustrate the logic through a number of examples, each of which demonstrates a subset of the specification situations identified below:

- Applicative specification of datatypes as we find them in EML. That is, no use of modalities.
- Specification of communication of respectively:
  - simple values (integers, pure functions, etc.)
  - channels
  - processes

- Specification of termination and return of respectively:
  - simple values (integers, pure functions, etc.)
  - channels
  - processes
- Specification using maximal fixpoints parameterised with a state. Note that this makes it possible to specify processes, that have an internal state. Think for example of a process that can do *up* and *down* actions, but it can never do more *down* actions than the number of preceding *up* actions. This system can easily be specified by a maximal fixpoint parameterised with a state which is a natural number always equal to the number of *up*'s minus the number of *down*'s.
- Specification using what we call *interface functions*: instead of interacting with a process, say p, explicitly via channels, interaction may happen via function calls, which then in turn perform the channel communication with p. This style of programming is frequent in CML. The problem is of course that if we want not to expose the channel interface of the process p, then how do we specify its behaviour? Traditional modal logic does not apply, since this is based on observing the channel communication. Some of the examples below illustrate our solution to this problem.

9.3.1 Natural Number Streams

The example illustrates communication of simple values. A channel `s` and a function `natstream` is required, where the function sends the stream of natural numbers on `s`.

```
signature NATSTREAM =
  sig
    val s : int chan
    val natstream : unit -> unit
    axiom natstream() |-
            (<:s!:>0 => true)
              andalso
            (always [:s!:]x => <:s!:>y => y==x+1)
  end
```

The axiom is read as follows: the value `0` can be sent on `s`, and also, it always holds that if an `x` is sent on `s`, then a `y` can thereafter be sent, such that `y==x+1`. The following structure satisfies the signature:

```
structure Natstream:NATSTREAM =
  struct
    val s : int chan = channel()
    fun natstream() =
          let
            fun count i = (transmit(s,i); count(i+1))
          in
            count 0
          end
  end
```

9.3.2 Several Natural Number Streams

The next example illustrates return of channels and communication of simple values and channels. It is a modification of the previous, where we now require a function that can generate several streams of natural numbers. That is, when calling the function (which is done once), we want it to immediately terminate and return a channel, on which we can request several streams.

```
signature NATSTREAMS =
  sig
    val natstreams : unit -> (int chan) chan
    axiom natstreams() |-
            return NEW(ss) =>
              always <ss!>NEW(s) =>
                ((<s!>0 => true)
                   andalso
                 (always [s!]x => <s!>y => y==x+1))
  end
```

As we see, `natstreams` returns a channel upon which integer typed channels can be communicated. In the axiom this channel is named `ss`, and we see that it is new. The axiom further says that on this channel a new channel `s` can always be sent, upon which a stream can finally be sent. The following structure satisfies the signature.

```
structure Natstreams:NATSTREAMS =
  struct
    fun natstreams() =
          let
            val ss : (int chan) chan = channel()
```

```
        fun count(c,i) = (transmit(c,i); count(c,i+1))
        fun main() = let val c : int chan = channel() in
                       transmit(ss,c);
                       fork(fn () => count(c,0));
                       main()
                     end
      in
        fork(main);
        ss
      end
  end
```

9.3.3 Sorting Lists

This example illustrates applicative specification and communication of simple values. It is an illustration of how a typical EML style can be mixed with a modal style of specification. A function `sort` is required, that can sort integer lists. The lists are to be communicated on a channel given as parameter to the function. First, we give a purely applicative specification of some auxiliary functions.

```
signature PREDICATES =
  sig
    val ordered : int list -> bool
    val permutation : int list * int list -> bool
    val count : int list -> int
    axiom ordered l ==
            (forall (a,b,l1,l2,l3) =>
               (l == l1@[a]@l2@[b]@l3) implies a <= b)
    axiom permutation(l,l') ==
            (forall x => count(x,l) == count(x,l'))
  end
```

Then we give the main signature, where we include the previous auxiliary signature in a `local` construct in order to keep it local.

```
signature SORT =
  sig
    val sort : (int list) chan -> unit
    local
      include PREDICATES
    in
```

```
      axiom sort(c) |- always [:c?:]l => <:c!:>l' =>
                         (ordered(l') andalso
                          permutation(l,l'))
    end
  end
```

The following structure satisfies the signature.

```
structure Sort:SORT =
  struct
    fun insert(i:int,[]) = [i]
       |insert(i:int,j::t) =
          if i <= j
            then i::j::t
            else j::(insert(i,t))
    fun dosort []     = []
       |dosort (i::t) = insert(i,dosort(t))
    fun sort(c) =
          let val l = accept c in
            transmit(c,dosort(l));
            sort(c)
          end
  end
```

9.3.4 Futures

This example illustrates return and use of interface functions (in this case: events). A function `future` is required that as arguments takes a function `f` and its argument `x`, and then immediately returns an event `ev` such that `sync(ev)` returns the value `f(x)`. The event is an example of a "process" that is returned, and then used as an interface to the future instead of an explicit channel interface.

```
signature FUTURE =
  sig
    val future : (int -> int) -> int -> int event
    axiom (f x) proper implies
            future f x |-
              return ev =>
                always sync ev |- return y => y == f x
  end
```

Note that `f x` must not have any side effects, it must be `proper`. Note also that the axiom contains two occurrences of `|-`. It reads as follows: assuming that `f x` is `proper`, then `future f x` satisfies the following – it immediately returns an event `ev` where after it always holds that `sync ev` can return `f x`.

The way to understand the axiom is to imagine the configuration into which first the process `future f x` is inserted, and into which then `sync ev` is inserted repeatedly (due to the `always`). Ignoring the `always` construct, it is the configuration containing `future f x` and `sync ev`, where `return` is linked to the latter, that must satisfy the formula: `return y => y == f x`. Of course this must happen via a communication inside the configuration between `future f x` and `sync ev`. The following structure satisfies the signature.

```
structure Future:FUTURE =
  struct
    fun future f x =
          let
            val resCh = channel()
            fun repeater x = (transmit(resCh, x); repeater x)
          in
            fork (fn () => repeater(f x));
            receive resCh
          end
  end
```

A use of the function is illustrated below:

```
let val ev1 = future f x
    val ev2 = future g y
in
  ... (sync ev1)*(sync ev1) + (sync ev2)
end
```

Note that the following axiom is not sufficient:

```
axiom (f x) proper implies
        future f x |-
          return ev =>
            always sync ev == f x
```

With the semantics outlined, the `always sync ev == f x` has namely the same meaning as `sync ev == f x`, so we loose the requirement that the event returns the value *each* time touched.

9.3.5 Resource Manager

This example illustrates return of channels and the use of maximal fixpoints. We require a function `permission` that when called, immediately returns two channels `free` and `alloc` upon which we can free and allocate a resource (it is the *up* and *down* counter mentioned earlier). We can never do more allocations than there are free resources. We use a maximal fixpoint with a counter as state to model this.

```
signature PERMISSION =
  sig
    val permission : unit -> (unit chan * unit chan)
    local
      max Count(f,a,0) = [:a!:]_ => false
                              andalso
                         <:f?:>_ => Count(f,a,1)
         |Count(f,a,n) = <:a!:>_ => Count(f,a,n-1)
                              andalso
                         <:f?:>_ => Count(f,a,n+1)
    in
      axiom permission() |-
              return (NEW free,NEW alloc) => Count(free,alloc,0)
    end
  end
```

Note that when the counter is `0`, no allocation is possible, but a free is possible yielding a counter of `1`. The following structure satisfies the signature.

```
structure Permission:PERMISSION =
  struct
    fun permission() =
          let val fr : unit chan = channel()
              val al : unit chan = channel()
              fun count(0) = (accept fr;count(1))
                 |count(n) = select[
                               wrap(receive fr,
                                    fn _ => count(n+1)),
                               wrap(send(al,()),
                                    fn _ => count(n-1))]
          in
            fork(fn () => count 0);
            (fr,al)
```

```
        end
    end
```

9.3.6 Variables

This example illustrates return of simple values and the use of maximal fixpoints and interface functions. We require a set of functions that give us the possibility of working with variables and assignment. Recall that in ML there are reference types. One can create a reference to a cell in the store, containing some initial value given. One can read the contents of such a cell, and one can change it. As shown in [BMT92] variables can be coded up using the concurrency primitives of CML. This gives us in fact a logic for CML including variables; that is: in a theoretic sense, since the notation might not be practical for specifying programs using variables. The signature below puts requirements on the implementation of variables.

```
signature VARIABLE =
  sig
    type 'a variable
    val declare  : '1a -> '1a variable
    val assign   : 'a variable * 'a -> unit
    val contents : 'a variable -> 'a
    local
      max Cell(i,d) = (contents(i) |-
                         return y => y==d andalso Cell(i,d))
                                 andalso
                      (assign(i,z) |-
                         return _ => Cell(i,z))
    in
      axiom declare x |- return v => Cell(v,x)
    end
  end
```

We have used a maximal fixpoint with the identification of the variable (the reference) and the current contents as parameters. If we apply `contents(i)` then we get the current contents, and the cell is unchanged. `assign` changes the contents. The following structure satisfies the signature.

```
structure Variable : VARIABLE =
  struct
    datatype
```

```
    'a variable = Var of {read : 'a chan, write : 'a chan}
  fun declare(i) =
    let
      val readCh  = channel()
      val writeCh = channel()
      fun var(x) =
            select[
              wrap(send(readCh,x),fn () => var(x)),
              wrap(receive writeCh   ,fn y  => var(y))
            ]
    in
      fork (fn () => var(i));
      Var{read=readCh,write=writeCh}
    end
  fun assign(Var{write,...},x) =
        transmit(write,x)
  fun contents(Var{read,...}) =
        accept read
end
```

The ... notation in the record patterns occurring in the definition of `assign` and `contents` is an "ignore-the-rest" primitive in ML.

9.3.7 Buffered Channels

This example illustrates applicative specification, return of simple values, and the use of maximal fixpoints and interface functions. The example does not show new concepts; it is rather the "big" example. Buffered channels provide a means of asynchronous communication (see section 8.1.3). For this, we first need to specify the queue datatype:

```
signature QUEUE_TYPE =
  sig
    type 'a queue
    val is_empty : 'a queue -> bool
    val empty    : 'a queue
    val put      : 'a * 'a queue -> 'a queue
    val next     : 'a queue -> 'a
    val remove   : 'a queue -> 'a queue
    axiom is_empty(empty) == true
    axiom next(put(x,empty)) == x
    axiom not is_empty(q) implies next(put(x,q)) == next(q)
```

```
    axiom remove(put(x,empty)) == empty
    axiom not is_empty(q) implies
            remove(put(x,q)) == put(x,remove(q))
  end
```

Then we use this datatype to specify buffered channels:

```
signature BUFFERED_CHAN =
  sig
    type 'a buffer_chan
    val buffer         : unit -> '1a buffer_chan
    val bufferTransmit : ('a buffer_chan * 'a) -> unit
    val bufferAccept   : 'a buffer_chan -> 'a
    local
      include QUEUE_TYPE
      max Buf(bc,q) =
            (bufferTransmit(bc,x) |- return _ =>
                                       Buf(bc,put(x,q)))
                 andalso
            (is_empty(q) implies
               bufferAccept(bc) |- [return!]_ => false)
                 andalso
            (not is_empty(q) implies
               bufferAccept(bc) |-
                 return y => (y==next(q) andalso
                              Buf(bc,remove(q))))
    in
      axiom buffer() |- return bc => Buf(bc,empty)
    end
  end
```

The axiom is defined in terms of a maximal fixpoint parameterised with the identification of the buffered channel and its current contents in terms of a queue. The axiom `buffer() |- ...` says that `buffer()` immediately terminates, returning the reference `bc` to a buffered channel, whereupon this buffered channel behaves like `Buf(bc,empty)`, which in turn behaves as follows: (1) after a call (and termination) of `bufferTransmit(bc,x)`, the buffer behaves like `Buf(bc,put(x,q))`. (2) if the buffer is empty, `bufferAccept(bc)` cannot terminate: it blocks. (3) if the buffer is non-empty, `bufferAccept(bc)` returns the next value in the queue, and then behaves like `Buf(bc,remove(q))`. The following structure satisfies the signature.

```
structure BufferChan:BUFFERED_CHAN =
  struct
    datatype 'a buffer_chan =
               BC of {inch : 'a chan, outch : 'a chan}
    fun buffer() =
          let val inCh  = channel()
              and outCh = channel()
              fun loop([],[]) = loop([accept inCh],[])
                | loop(front as (x::r),rear) =
                    select[wrap(receive inCh,
                             fn y  => loop(front,y::rear)),
                           wrap(send(outCh, x),
                             fn () => loop(r,rear))]
                | loop([],rear) = loop(List.rev rear, [])
          in
            fork(fn () => loop([],[]));
            BC{inch=inCh, outch=outCh}
          end
    fun bufferTransmit(BC{inch,...},x) = transmit(inch,x)
    fun bufferAccept(BC{outch,...}) = accept outch
  end
```

9.4 Discussion

One-level versus Two-level Logic

The logic presented is designed as an extension of EML. EML is obtained by basically adding some boolean valued expressions to ML, and then use boolean expressions in general as axioms. Applying this idea to ECML means that we add some more boolean valued expressions, basically exp_1 `|-` exp_2 and `<:`$exp\,dir$`:>`$match$. We call this a *one-level logic* since there is only one syntactic category of formulae: boolean expressions. Although simple, this idea may give rise to some strange specifications, as for example the following where a modality occurs at the outer level:

```
signature WEIRD =
  sig
    val c : int chan
    axiom [:c!:]_ => false
  end
```

The axiom requires that "the process" cannot communicate on `c`. But there is no process indicated, so the axiom gives no intuitive meaning. Our semantics, however, gives this a meaning since every model that is checked against the signature has an initially empty configuration, and this configuration can surely not perform a communication on `c`, so the axiom is equivalent to `true`. One could of course try to rule out such cases.

Another situation, that in the beginning of the design gave rise to some worry were axioms with nested occurrences of `|-` as in:

$$\texttt{axiom}\ exp_1\ \texttt{|-}\ (exp_2\ \texttt{|-}\ exp_3)$$

It turned out though that this perfectly gives meaning, as we have seen in some examples which use interface functions: the variable example and the buffered channel example. In these examples the second `|-` is, however, put into a fixpoint construct. A more direct example is the following, which is a property of the `VARIABLE` signature from earlier:

```
axiom declare(x)  |- return v =>
        contents(v) |- return y => y==x
```

It says that if we declare a variable with the initial value `x`, getting a reference `v` back, and then take the contents of `v`, then we get `x`. That is, each time we write exp_1 `|-` exp_2, the expression exp_1 is put into the multiset of the configuration, and exp_2 is then evaluated in the context of this extended configuration. This can of course be repeated, thus extending the configuration each time.

A *two-level logic* would deal with an *outer-level* category $\mathcal{F}$ of formulae, amongst which would be traditional predicate logic formulae like for example exp_1 `==` exp_2 and `forall` *match*, and then: *exp* `|-` $\mathcal{P}$ where $\mathcal{P}$ is the *inner-level* category of process formula. $\mathcal{P}$ would then contain the modalities. In such a two-level logic, the outermost $\mathcal{F}$ predicates would have a semantics of type $E \rightarrow bool$, where E is an environment corresponding to a structure. This corresponds to the semantics of axioms in EML. The $\mathcal{P}$ predicates on the other hand would have a different semantics with type $(E \times Con) \rightarrow bool$, where the configuration component is built from the *exp*'s occurring within formulae of the form: *exp* `|-` $\mathcal{P}$.

This *two-headed* semantics could also have been applied in our *one-level* approach, thus having two semantics for each formulae expression, depending on whether it occurs at the outermost level of a signature or to the right of `|-`. Certainly this would yield a bigger semantics. The one-level approach (with a one-headed semantics) is chosen since it seems simpler from a design point of view: fewer concepts, just add some further expressions.

Why is it not a Refinement Logic

The logic is not a refinement logic in the sense this term has been used earlier in this work. That is, we do not allow formulae (boolean expressions occurring as axioms) to contain an arbitrary mixture of programming constructs and specification constructs. To illustrate this, we consider an example of a formula where programming constructs are mixed with specification constructs.

```
signature INCREASE =
  sig
    val increase : int chan -> bool
    axiom increase(c) |- let val v = accept c in
                              <:c!:>y => y>v
                          end
  end
```

The signature specifies a function that inputs a value `v` on its argument channel `c` and then behaves as specified by the formula `<:c!:>y => y>v`. The program-part of this axiom is: `let val v = accept c`, defining exactly what action to begin with. The following formula-part says that the process thereafter *can* (amongst other actions) output a value `y` on `c` where `y>v`.

The semantics of this should be somewhat similar to the semantics of the refinement calculus in which for some programming construct Op: $p \models Op(\psi_1, \ldots, \psi_n)$ iff. there exist processes $p_1, \ldots, p_n$ such that for each $i \in \{1, \ldots, n\}$ it holds that $p_i \models \psi_i$ and further $p \equiv Op(p_1, \ldots, p_n)$. In the above case, we would in particular need to find a process (expression) that satisfies `<:c!:>y => y>v`. The problem arises with the expression `y>v` since the comparison operator `>` is a programming construct in CML, and we therefore get to choose between two interpretations of `y>v`: either as a predicate, which is satisfied by any process in case it holds, or as a program, in which case it is satisfied by any program `p` where `p == (y>v)`.

The problem arises since boolean expressions are used as formulae, and therefore in this way get a double role as 'formulae' on one hand, and boolean valued 'programs' on the other hand. Put differently in terms of another example: the formula *exp* `|-1` should mean that *exp*`==1`, but *exp* `|-true` should *not* mean that *exp* `== true`; but rather that *exp* can behave in any possible way.

Of course this problem is a result of our decision of using boolean expressions as axioms, a decision that might be rejected on this basis. It seems, though, to be a rather "stupid" problem in not being very deep.

One can, however, express the combination of programming and specification in another, though more detailed, way. The following signature illustrates this for the above example.

```
signature INCREASE =
  sig
    val increase : int chan -> bool
    local
      val rest : int chan * int -> bool
      axiom rest(c,v) |- <:c!:>y => y>v
    in
      axiom increase(c) == let val v = accept c in
                               rest(c,v)
                           end
    end
  end
```

Fixpoints versus Temporal Operators

We have chosen use fixpoint operators for describing infinite behaviour. An alternative would have been to use temporal logic operators like *always*, *eventually*, *until*, etc. Fixpoints are typically used in logics based on Hennessy-Milner logic [Lar90], which again is the typical logic for CCS-like programs, where transitions are labelled with actions. So one reason for choosing fixpoints is "tradition". A second reason is, that with fixpoints, all of the normal temporal logic operators can be programmed as derived forms. Hence, there is no need for choosing the "right" set of temporal operators. A last convincing reason is that parameterised fixpoints give us a way of specifying processes with a behaviour that in a complicated way depends on the past: counters, stacks, queues, databases, etc. If we only had temporal operators, then we were forced to introduce constructs with similar expressive power to capture the notion of state. A possible solution in that case is to introduce state variables, which are changed by actions, and which can be examined by formulae in the logic [Cha87, AM93], [Eme90].

The solution in [Cha87] is based on storing the communication history of a channel. To each channel is associated a 'temporal variable' that contains the list of messages passed on that channel. In the temporal formulae, one can then refer to these variables. As a generalisation the paper suggests that a temporal variable is not necessarily associated with one particular channel, and that it can hold any kind of information (not just the communication history): it simply contains the state of the system. One then specifies the effects of channel communication on the variable, and one can refer to the contents of such a variable in specifications.

The latter solution is close to one that I have studied in particular, and which in [Eme90] page 1022 is provided by an *existential formulae* of the form $\exists\Sigma \cdot \psi$, where Σ is a signature, and ψ is a temporal formula. A labelled transition system satisfies this formula, if there can be assigned a Σ-model to each state, such that

ψ holds. The formula ψ can then access the model of each state.

Variables

We have seen how variables can be programmed and specified. This could be carried further with the objective to obtain a logic for variables. The approach should be theoretically to regard ML reference primitives as coded up in terms of CML concurrency primitives, just as is suggested in [BMT92].

Part V

Conclusion

Chapter 10

Conclusion and Future Work

10.1 Conclusion

Our initial motivation behind this work has been to develop a specification framework for CML. Experiments with a solution to this problem lead to the conclusion, that the fork operator in CML appeared to have properties that were significantly different from the properties of the parallel composition operator in CCS. Consequently, the Hennessy-Milner logic for CCS could not directly be the basis for a CML specification logic. As the other side of the same coin, the obvious notion of bisimulation equivalence seemed not to be a congruence.

This lead to the design of the Fork Calculus, FC, where the problems with the fork operator were studied in detail, detached from the other technicalities of CML.

In chapter 3, we provided FC with an operational semantics, and we defined an equivalence relation between FC processes, which was shown to be a congruence. In chapter 4, we provided two ways of reasoning about processes: an axiomatisation and a logic. The axiomatisation was shown to be sound and complete with respect to the congruence defined in chapter 3. The logic was a variation of Hennessy-Milner logic, modified to deal with the properties of the fork operator. The logic was shown to be adequate with respect to the congruence defined in chapter 3.

In chapters 5 and 6 we extended the calculus with a channel-restriction operator, defined a congruence relation, and we provided also for this extension a sound and complete axiomatisation. Further, we defined a refinement logic in which the operators of the Fork Calculus were also operators in the logic. The logic were shown compositional in the sense that one can infer properties of a composite process in terms of properties of its constituents. It was also shown to be adequate with respect to the congruence defined. A set of proof rules were established, and a major example using the refinement logic were carried out.

In chapter 7 we defined the ψ-calculus, where channels may be communicated between processes. Hence, the calculus is comparable to the π-calculus, but with a fork operator instead of a parallel composition operator. We defined a notion of equivalence, and finally a logic, which was mainly inspired by the logic for FC and from the corresponding logic for the π-calculus. That is, the logic provides means for specifying forking as well as scope extrusion: the fact that a process communicates a local channel to the context. This logic is an important departure for defining a logic for CML, since two major problems in the context of CML are exactly forking and scope extrusion.

In chapter 8 we defined a new semantics for CML in terms of an action-labelled transition system. Other existing semantics of CML are in principle defined in terms of *unlabelled* transition systems, and this is unsatisfactory when defining an action-based Hennessy-Milner like logic.

Finally, in chapter 9 we defined the CML extension ECML, by basically adding the notion of axiom to signatures, where axioms are boolean CML expressions extended with a few powerful specification constructs such as universal quantification and modalities. We sketched a semantics for ECML, and further demonstrated its use in a number of examples.

10.2 Future Work

There are three branches of work. First of all, one can consider the various versions of the Fork Calculus that have been defined here, and perform a more detailed comparison with CCS-based calculi. An important difference between FC and CCS is for example that in CCS, the states of the transition system are CCS processes; that is: terms in the source language. This is not the case for FC, where global transitions are between multisets of processes. The theory of FC is consequently more complicated than the theory of CCS. This extra complication reflects the extra power of the fork operator in comparison with the parallel composition operator in CCS. Parallel composition is a very explicit construct in the sense that the arguments to be put in parallel are both given together. Forking, on the other hand, is implicit in the sense that only one of the arguments (to be put in parallel) is given, while the other is "the rest of the program", which is possibly unknown in the context where the forking occurs.

It has not been our intention to provide an alternative to CCS, but rather to provide a setting for examining the properties of the fork operator, and for this purpose FC has without any question been of invaluable help. It has surely been felt as a "practical" tool for understanding semantic properties of CML.

Another branch of work is to elaborate on the semantics of CML defined in chapter 8. Recall, that an equivalence between CML expressions were not defined, and a natural continuation will therefore be to define such an equivalence, and to show various congruence properties. Although a complete axiomatisation is not possible for CML, one can provide a useful axiomatisation.

The third branch includes further elaboration of the specification language for CML. We consider the work presented in this thesis of major importance for such future work. It should be shown that the designed specification language is adequate with respect to a suitable expression–equivalence; i.e. two expressions enjoy the same properties of the specification language precisely when they are behaviourally equivalent. In order to provide a useful specification logic, many examples have to be written, possibly suggesting modifications of the logic proposed here. Also, a proof theory needs to be defined including proof rules. Finally, proofs cannot be done solely by hand, so a proof theory must be supported by software tools. One cannot hope for a fully automatic theorem prover for a language like ECML, but one can hope for a user-driven proof-support involving editing, proof checking, and automated theorem proving in particular simple cases.

Appendix A

Proofs Concerning Standard Results

A.1 Strong Equivalence

We define the converse R^{-1} of a binary relation, and the composition R_1R_2 of two binary relations as follows:

$$
\begin{array}{rcl}
R^{-1} & \stackrel{def}{=} & \{(y,x) \mid (x,y) \in R\} \\
R_1R_2 & \stackrel{def}{=} & \{(x,z) \mid \exists y \cdot (x,y) \in R_1 \wedge (y,z) \in R_2\}
\end{array}
$$

We can then state the following lemma, which is later used to prove that $\sim$ is an equivalence.

Lemma A.1 *Assume that each $S_i \subseteq St \times St$ ($i = 1, 2, \ldots$) is a bisimulation, and that I is a subset of $\boldsymbol{N}$, and that $Id = \{(p,p) \mid p \in St\}$ is the identity relation. Then the following are all bisimulations:*

$$
\begin{array}{ll}
(1) & Id \\
(2) & S_i^{-1} \\
(3) & S_1S_2 \\
(4) & \bigcup\{S_i \mid i \in I\}
\end{array}
$$

Proof

(1) Obvious.

(2) Assume that $(p, q) \in S^{-1}$ and that $p \xrightarrow{l} p'$. We know that $(q, p) \in S$ and since S is a bisimulation, $q \xrightarrow{l} q'$ where $(q', p') \in S$. Then $(p', q') \in S^{-1}$.

A symmetric argument where q starts with a move ends the proof.

(3) Assume that $(p, r) \in S_1 S_2$ and that $p \xrightarrow{l} p'$. Now, for some q, $(p, q) \in S_1$ and $(q, r) \in S_2$. Since S_1 is a bisimulation we have that $q \xrightarrow{l} q'$ where $(p', q') \in S_1$. Since similarly S_2 is a bisimulation, we then also have that $r \xrightarrow{l} r'$ where $(q', r') \in S_2$. Finally $(p', r') \in S_1 S_2$.

A symmetric argument where r starts with a move ends the proof.

(4) Assume that $(p, q) \in \bigcup\{S_i \mid i \in I\}$ and that $p \xrightarrow{l} p'$. Obviously $(p, q) \in S_j$ for some j in I, and since S_j is a bisimulation, $q \xrightarrow{l} q'$ where $(p', q') \in S_j$. Therefore $(p', q') \in \bigcup\{S_i \mid i \in I\}$.

∎

Lemma A.2 $\sim$ *is the largest bisimulation.*

Proof

Definition 2.3 defines $\sim$ to be the union of all bisimulations. By lemma A.1 (4), it then follows that $\sim$ itself is a bisimulation. Finally, from the definition it follows that it includes any other strong bisimulation, and thus is the largest.

∎

Theorem A.3 ($\sim$ is an equivalence) *For any programs p, q and r it holds that:*

(1) $p \sim p$
(2) $p \sim q$ *implies* $q \sim p$
(3) $p \sim q$ *and* $q \sim r$ *implies* $p \sim r$

Proof

(1) Due to lemma A.1 (1) the identity relation Id is a bisimulation, so since $(p, p) \in Id$ it follows that $p \sim p$.

(2) Assume that $p \sim q$. Then $(p, q) \in S$ for some bisimulation S. But then $(q, p) \in S^{-1}$ which is also a bisimulation due to lemma A.1 (2). Consequently $q \sim p$.

(3) Assume that $p \sim q$ and that $q \sim r$. Then $(p,q) \in S_1$ for some bisimulation S_1 and $(q,r) \in S_2$ for some bisimulation S_2. Thus $(p,r) \in S_1S_2$ which is also a bisimulation due to lemma A.1 (3). Consequently $p \sim r$.

■

Proposition A.4 (Behaviour proposition) *For all programs p, q: $p \sim q$ iff. for all $l \in Lb$,*

1. *Whenever $p \xrightarrow{l} p'$ for some p' then $q \xrightarrow{l} q'$ for some q' and $p' \sim q'$*
2. *Whenever $q \xrightarrow{l} q'$ for some q' then $p \xrightarrow{l} p'$ for some p' and $p' \sim q'$*

Proof

We define a new relation $\sim'$ in terms of $\sim$, and we then show that $\sim'$ equals $\sim$. From this we shall be able to conclude the proposition. The relation $\sim'$ is defined as follows:

For all programs p, q, $p \sim' q$ iff, for all $l \in Lb$,

1. Whenever $p \xrightarrow{l} p'$ for some p' then $q \xrightarrow{l} q'$ for some q' and $p' \sim q'$
2. Whenever $q \xrightarrow{l} q'$ for some q' then $p \xrightarrow{l} p'$ for some p' and $p' \sim q'$

We show that $\sim'$ and $\sim$ are one and the same relation,

1. $\sim \subseteq \sim'$:

   Assume that $(p,q) \in \sim$. Since $\sim$ is a bisimulation (lemma A.2) we have that condition (1) and (2) in the definition of $\sim'$ are satisfied. Therefore $(p,q) \in \sim'$.

2. $\sim' \subseteq \sim$:

   We show that $\sim'$ is a bisimulation. This is enough, since by definition 2.3, $\sim$ is the union of all bisimulations.

   Assume that $p \sim' q$ and that $p \xrightarrow{l} p'$. Then by definition of $\sim'$, it follows that $q \xrightarrow{l} q'$ where $p' \sim q'$. But we proved in (1) that $\sim \subseteq \sim'$, so $p' \sim' q'$.

   A symmetric argument where q starts with a move ends the proof.

Since $\sim'$ equals $\sim$ the proposition follows immediately from the definition of $\sim'$.

■

A.2 Weak Equivalence

First, we need the following lemma:

Lemma A.5 *If S is a weak bisimulation, and if $(p, q) \in S$, then for all $l \in Lb$,*

1. *Whenever $p \stackrel{l}{\Longrightarrow} p'$ for some p' then $q \stackrel{l}{\Longrightarrow} q'$ for some q' and $(p', q') \in S$.*
2. *Whenever $q \stackrel{l}{\Longrightarrow} q'$ for some q' then $p \stackrel{l}{\Longrightarrow} p'$ for some p' and $(p', q') \in S$.*

Proof

This is straightforward and should be easy to see intuitively. Try with a small example. A formal proof may be done by induction on the number of τ transitions.

■

We can then state the following lemma, which is later used to prove that $\approx$ is an equivalence.

Lemma A.6 *Assume that each $S_i \subseteq St \times St$ $(i = 1, 2, \ldots)$ is a weak bisimulation, and that I is a subset of $\boldsymbol{N}$, and that $Id = \{(p, p) \,|\, p \in St\}$ is the identity relation. Then the following are all weak bisimulations:*

$$\begin{array}{ll} (1) & Id \\ (2) & S_i^{-1} \\ (3) & S_1 S_2 \\ (4) & \bigcup\{S_i \,|\, i \in I\} \end{array}$$

Proof

(1) Obvious.

(2) Assume that $(p, q) \in S^{-1}$ and that $p \stackrel{l}{\longrightarrow} p'$. We know that $(q, p) \in S$ and since S is a weak bisimulation, $q \stackrel{l}{\Longrightarrow} q'$ where $(q', p') \in S$. Then $(p', q') \in S^{-1}$.

A symmetric argument where q starts with a move ends the proof.

(3) Assume that $(p, r) \in S_1 S_2$ and that $p \stackrel{l}{\longrightarrow} p'$. Now, for some q, $(p, q) \in S_1$ and $(q, r) \in S_2$. Since S_1 is a bisimulation we have that $q \stackrel{l}{\Longrightarrow} q'$ where $(p', q') \in S_1$. Since similarly S_2 is a bisimulation, lemma A.5 then yields that $r \stackrel{l}{\Longrightarrow} r'$ where $(q', r') \in S_2$. Finally $(p', r') \in S_1 S_2$.

A symmetric argument where r starts with a move ends the proof.

(4) Assume that $(p, q) \in \bigcup\{S_i \mid i \in I\}$ and that $p \xrightarrow{l} p'$. Obviously $(p, q) \in S_j$ for some j in I, and since S_j is a bisimulation, $q \stackrel{l}{\Longrightarrow} q'$ where $(p', q') \in S_j$. Therefore $(p', q') \in \bigcup\{S_i \mid i \in I\}$.

■

Lemma A.7 $\approx$ *is the largest weak bisimulation.*

Proof

Definition 2.9 defines $\approx$ to be the union of all weak bisimulations. By lemma A.6 (4), it then follows that $\approx$ itself is a bisimulation. Finally, from the definition it follows that it includes any other weak bisimulation, and thus is the largest.

■

Theorem A.8 ($\approx$ is an equivalence) *For any programs p, q and r it holds that:*

(1) $p \approx p$
(2) $p \approx q$ *implies* $q \approx p$
(3) $p \approx q$ *and* $q \approx r$ *implies* $p \approx r$

Proof

(1) Due to lemma A.6 (1) the identity relation Id is a weak bisimulation, so since $(p, p) \in Id$ it follows that $p \approx p$.

(2) Assume that $p \approx q$. Then $(p, q) \in S$ for some weak bisimulation S. But then $(q, p) \in S^{-1}$ which is also a weak bisimulation due to lemma A.6 (2). Consequently $q \approx p$.

(3) Assume that $p \approx q$ and that $q \approx r$. Then $(p, q) \in S_1$ for some weak bisimulation S_1 and $(q, r) \in S_2$ for some weak bisimulation S_2. Thus $(p, r) \in S_1S_2$ which is also a weak bisimulation due to lemma A.6 (3). Consequently $p \approx r$.

■

Proposition A.9 (Weak behaviour proposition) *For all programs p, q: $p \approx q$ iff. for all $l \in Lb$,*

1. *Whenever $p \xrightarrow{l} p'$ for some p' then $q \overset{l}{\Longrightarrow} q'$ for some q' and $p' \approx q'$*
2. *Whenever $q \xrightarrow{l} q'$ for some q' then $p \overset{l}{\Longrightarrow} p'$ for some p' and $p' \approx q'$*

Proof

We define a new relation $\approx'$ in terms of $\approx$, and we the show that $\approx'$ equals $\approx$. From this we shall be able to conclude the proposition. The relation $\approx'$ is defined as follows:

For all programs p, q, $p \approx' q$ iff, for all $l \in Lb$,

1. Whenever $p \xrightarrow{l} p'$ for some p' then $q \overset{l}{\Longrightarrow} q'$ for some q' and $p' \approx q'$
2. Whenever $q \xrightarrow{l} q'$ for some q' then $p \overset{l}{\Longrightarrow} p'$ for some p' and $p' \approx q'$

We show that $\approx'$ and $\approx$ are one and the same relation,

1. $\approx \subseteq \approx'$:

   Assume that $(p, q) \in \approx$. Since $\approx$ is a weak bisimulation (lemma A.7) we have that condition (1) and (2) in the definition of $\approx'$ are satisfied. Therefore $(p, q) \in \approx'$.

2. $\approx' \subseteq \approx$:

   We show that $\approx'$ is a weak bisimulation. This is enough, since by definition 2.9, $\approx$ is the union of all weak bisimulations.

   Assume that $p \approx' q$ and that $p \xrightarrow{l} p'$. Then by definition of $\approx'$, it follows that $q \overset{l}{\Longrightarrow} q'$ where $p' \approx q'$. But we proved in (1) that $\approx \subseteq \approx'$, so $p' \approx' q'$.

   A symmetric argument where q starts with a move ends the proof.

Since $\approx'$ equals $\approx$ the proposition follows immediately from the definition of $\approx'$.

∎

Appendix B

Proofs Concerning Strong Axiomatisation of FC

The appendix contains proofs of properties used in the main text.

Lemma B.1

$$\{\!|\boldsymbol{forked}(p); q|\!\} \smile \{\!|p, q|\!\} \tag{B.1}$$
$$\{\!|\boldsymbol{forked}(p)|\!\} \smile \{\!|p|\!\} \tag{B.2}$$
$$P \cup Q \smile Q \cup P \tag{B.3}$$
$$P \cup (Q \cup R) \smile (P \cup Q) \cup R \tag{B.4}$$
$$P \cup \{\!|\boldsymbol{nil}|\!\} \smile P \tag{B.5}$$

The lemma also holds for the relation $\asymp$ defined in section 3.3.1.

Proof

Only the proof of (B.1) is presented here; the proofs of the other equations are left for the reader.

We show that $\{\!|\boldsymbol{forked}(p); q|\!\} \smile \{\!|p, q|\!\}$ by showing that if one of the programs can perform an action, then the other program can perform the same action, and the two resulting programs are equivalent.

Assume that $\{\!|\boldsymbol{forked}(p); q|\!\} \xrightarrow{\alpha} S$. Since $\boldsymbol{forked}(p); q \overset{\Phi(p)}{\hookrightarrow} \boldsymbol{nil}; q$ we can conclude from the global semantics of the $\boldsymbol{forked}$-operator that $\{\!|p, \boldsymbol{nil}; q|\!\} \xrightarrow{\alpha} S$. From this follows easily that $\{\!|p, q|\!\} \xrightarrow{\alpha} S'$ where $S' \smile S$.

If we let the right hand side $\{\!|p, q|\!\}$ move first, we can use the above argument in the reverse direction.

■

Proposition B.2 **($\mathcal{A}X$ is sound)** *The axioms $\mathcal{A}X$ are sound with respect to strong process congruence.*

Proof

For each equation $t_1 = t_2$ we must prove that $t_1 \equiv t_2$, which due to the definition 3.7 of strong process congruence amounts to proving that $\{\!| t_1; \pi |\!\} \smile \{\!| t_2; \pi |\!\}$. We prove a selection of cases and leave the rest of the proof to the reader.

(4.8) $p; \boldsymbol{nil} = p$

We shall use proposition 2.6 to show that $\{\!| p; \boldsymbol{nil}; \pi |\!\} \smile \{\!| p; \pi |\!\}$. We can use induction on the number of possible $\hookrightarrow$ transitions of p. If $Stop(p)$, then both programs can perform a π action and become $\{\!| \boldsymbol{nil} |\!\}$. If on the other hand p can perform an action, then $\{\!| p; \boldsymbol{nil}; \pi |\!\} \xrightarrow{\alpha} \{\!| p'; \boldsymbol{nil}; \pi |\!\} \cup R$ and $\{\!| p; \pi |\!\} \xrightarrow{\alpha} \{\!| p'; \pi |\!\} \cup R$. The induction hypothesis yields that $\{\!| p'; \boldsymbol{nil}; \pi |\!\} \smile \{\!| p'; \pi |\!\}$, and the congruence property of $\cup$ then yields that $\{\!| p'; \boldsymbol{nil}; \pi |\!\} \cup R \smile \{\!| p'; \pi |\!\} \cup R$.

(4.9) $\boldsymbol{nil}; p = p$

We shall use proposition 2.6 to show that $\{\!| \boldsymbol{nil}; p; \pi |\!\} \smile \{\!| p; \pi |\!\}$. If $Stop(p)$, then both programs can perform a π action and become $\{\!| \boldsymbol{nil} |\!\}$. If on the other hand p can perform an action, then $\{\!| \boldsymbol{nil}; p; \pi |\!\} \xrightarrow{\alpha} \{\!| p'; \pi |\!\} \cup R$ as well as $\{\!| p; \pi |\!\} \xrightarrow{\alpha} \{\!| p'; \pi |\!\} \cup R$.

(4.16) $\boldsymbol{fork}(p) = \tau; \boldsymbol{forked}(p)$

We will show, that the following relation S is a bisimulation:

$$\begin{aligned} S \stackrel{def}{=} \; & \{(\{\!| \boldsymbol{fork}(p); \pi |\!\}, \{\!| \tau; \boldsymbol{forked}(p); \pi |\!\}), \\ & \;(\{\!| \boldsymbol{nil}; \pi, p |\!\}, \{\!| \boldsymbol{nil}; \boldsymbol{forked}(p); \pi |\!\})\} \\ & \cup Id_{\mathcal{M}^{\Phi}} \end{aligned}$$

Here $Id_{\mathcal{M}^\Phi} = \{(p,p) \mid p \in \mathcal{M}^\Phi\}$. We don't need to consider the pairs in $Id_{\mathcal{M}^\Phi}$ since this is obviously itself a bisimulation. So we consider the other pairs in turn.

The pair $(\{\!|\boldsymbol{fork}(p); \pi|\!\}, \{\!|\tau; \boldsymbol{forked}(p); \pi|\!\})$**:** Assume that
$\{\!|\boldsymbol{fork}(p); \pi|\!\} \xrightarrow{\tau} \{\!|\boldsymbol{nil}; \pi, p|\!\}$.
But also
$\{\!|\tau; \boldsymbol{forked}(p); \pi|\!\} \xrightarrow{\tau} \{\!|\boldsymbol{nil}; \boldsymbol{forked}(p); \pi|\!\}$
and
$(\{\!|\boldsymbol{nil}; \pi, p|\!\}, \{\!|\boldsymbol{nil}; \boldsymbol{forked}(p); \pi|\!\}) \in S$.

The case where we start with $\{\!|\tau; \boldsymbol{forked}(p); \pi|\!\}$ is similar.

The pair $(\{\!|\boldsymbol{nil}; \pi, p|\!\}, \{\!|\boldsymbol{nil}; \boldsymbol{forked}(p); \pi|\!\})$**:** Assume that $\{\!|\boldsymbol{nil}; \pi, p|\!\} \xrightarrow{\alpha} P$. We know that $\boldsymbol{nil}; \boldsymbol{forked}(p); \pi \overset{\Phi(p)}{\hookrightarrow} \boldsymbol{nil}; \pi$ and in combination with the assumption, the semantic rule **Forked**$^{\{\!|\,|\!\}}$ yields that $\{\!|\boldsymbol{nil}; \boldsymbol{forked}(p); \pi|\!\} \xrightarrow{\alpha} P$. Since $Id_{\mathcal{M}^\Phi} \subseteq S$, $(P, P) \in S$.

The case where we start with $\{\!|\boldsymbol{nil}; \boldsymbol{forked}(p); \pi|\!\}$ is similar.

(4.17) $\boldsymbol{forked}((\alpha; \boldsymbol{forked}(p)) + q) = \boldsymbol{forked}((\alpha; p) + q)$

We first prove that $\{\!|(\alpha; \boldsymbol{forked}(p)) + q|\!\} \smile \{\!|(\alpha; p) + q|\!\}$. We obviously only need to examine the α-transition. Suppose $\{\!|(\alpha; \boldsymbol{forked}(p)) + q|\!\} \xrightarrow{\alpha} \{\!|\boldsymbol{nil}; \boldsymbol{forked}(p)|\!\}$. Then also $\{\!|(\alpha; p) + q|\!\} \xrightarrow{\alpha} \{\!|\boldsymbol{nil}; p|\!\}$. Since $\boldsymbol{nil}; \boldsymbol{forked}(p) \equiv \boldsymbol{forked}(p)$ and $\boldsymbol{nil}; p \equiv p$ (axiom 4.9 which was proven sound), we have that $\{\!|\boldsymbol{nil}; \boldsymbol{forked}(p)|\!\} \smile \{\!|\boldsymbol{forked}(p)|\!\}$ and $\{\!|p|\!\} \smile \{\!|\boldsymbol{nil}; p|\!\}$ due to lemma 3.10. But lemma B.1 (B.2) says that $\{\!|\boldsymbol{forked}(p)|\!\} \smile \{\!|p|\!\}$, so consequently $\{\!|\boldsymbol{nil}; \boldsymbol{forked}(p)|\!\} \smile \{\!|\boldsymbol{nil}; p|\!\}$. A similar argument in the reverse direction ends the proof.

Now that we have established that $\{\!|(\alpha; \boldsymbol{forked}(p)) + q|\!\} \smile \{\!|(\alpha; p) + q|\!\}$ we conclude that $\{\!|\pi, ((\alpha; \boldsymbol{forked}(p)) + q)|\!\} \smile \{\!|\pi, ((\alpha; p) + q)|\!\}$ due to the congruence property of $\smile$. But then lemma B.1 (B.1) yields that $\{\!|\boldsymbol{forked}((\alpha; \boldsymbol{forked}(p)) + q); \pi|\!\} \smile \{\!|\boldsymbol{forked}((\alpha; p) + q); \pi|\!\}$ so $\boldsymbol{forked}((\alpha; \boldsymbol{forked}(p)) + q) \equiv \boldsymbol{forked}((\alpha; p) + q)$.

$\blacksquare$

Proposition B.3 ($\mathcal{E}_1$ is sound) *The expansion law $\mathcal{E}_1$ is sound with respect to strong process congruence.*

Proof

We want to prove that $\textbf{\textit{forked}}(A); M \equiv A' + B' + C' + D' + E'$ which due to definition 3.7 follows if we can prove $\{\!|\textbf{\textit{forked}}(A); M; \pi|\!\} \smile \{\!|(A' + B' + C' + D' + E'); \pi|\!\}$. Using property (B.1) in lemma B.1, we can bring the left hand side of this equation into a more convenient form: $\{\!|\textbf{\textit{forked}}(A); M; \pi|\!\} \smile \{\!|A, M; \pi|\!\}$. So it remains to show $\{\!|A, M; \pi|\!\} \smile \{\!|(A' + B' + C' + D' + E'); \pi|\!\}$.

We show this by showing that if one of the programs can perform an action, then the other program can perform the same action, and the two resulting programs are equivalent. First, we let the left hand side $\{\!|A, M; \pi|\!\}$ perform an action. There are three cases.

Case 1 $\{\!|A|\!\} \xrightarrow{\alpha_i} \{\!|A_i|\!\}$ and $\{\!|A, M; \pi|\!\} \xrightarrow{\alpha_i} \{\!|A_i, M; \pi|\!\}$. We also know that $A' \overset{\alpha_i}{\hookrightarrow} \textbf{\textit{forked}}(A_i); M$ and therefore that $\{\!|(A' + B' + C' + D' + E'); \pi|\!\} \xrightarrow{\alpha_i} \{\!|\textbf{\textit{forked}}(A_i); M; \pi|\!\}$. Finally, property (B.1) yields that $\{\!|\textbf{\textit{forked}}(A_i); M; \pi|\!\} \smile \{\!|A_i, M; \pi|\!\}$.

Case 2 $M; \pi$ makes a move. In this case, since M is not $\textbf{\textit{nil}}$, and since $M = N + L$, either N or L makes the move, thus giving two cases.

Case 2.1 $\{\!|M; \pi|\!\} \xrightarrow{\beta_j} \{\!|N_j; \pi|\!\}$ and $\{\!|A, M; \pi|\!\} \xrightarrow{\beta_j} \{\!|A, N_j; \pi|\!\}$. We also know that $B' \overset{\beta_j}{\hookrightarrow} \textbf{\textit{forked}}(A); N_j$ and therefore that $\{\!|(A' + B' + C' + D' + E'); \pi|\!\} \xrightarrow{\beta_j} \{\!|\textbf{\textit{forked}}(A); N_j; \pi|\!\}$. Finally, property (B.1) yields that $\{\!|\textbf{\textit{forked}}(A); N_j; \pi|\!\} \smile \{\!|A, N_j; \pi|\!\}$.

Case 2.2 $\{\!|M; \pi|\!\} \xrightarrow{\gamma_k} \{\!|\textbf{\textit{forked}}(C_k); \pi|\!\}$ and $\{\!|A, M; \pi|\!\} \xrightarrow{\gamma_k} \{\!|A, \textbf{\textit{forked}}(C_k); \pi|\!\}$. We also know that $C' \overset{\gamma_k}{\hookrightarrow} \textbf{\textit{forked}}(A); \textbf{\textit{forked}}(C_k)$ and therefore that $\{\!|(A'+B'+C'+D'+E'); \pi|\!\} \xrightarrow{\gamma_k} \{\!|\textbf{\textit{forked}}(A); \textbf{\textit{forked}}(C_k); \pi|\!\}$. Finally, property (B.1) yields that $\{\!|\textbf{\textit{forked}}(A); \textbf{\textit{forked}}(C_k); \pi|\!\} \smile \{\!|A, \textbf{\textit{forked}}(C_k); \pi|\!\}$.

Case 3 A and $M; \pi$ communicates. In this case, since $M = N + L$, either N or L communicates with A, thus giving two cases.

Case 3.1 $\{\!|A|\!\} \xrightarrow{\alpha_i} \{\!|A_i|\!\}$, $\{\!|M; \pi|\!\} \xrightarrow{\beta_j} \{\!|N_j; \pi|\!\}$ where $\alpha_i = rev(\beta_j)$ and $\{\!|A, M; \pi|\!\} \xrightarrow{\tau} \{\!|A_i, N_j; \pi|\!\}$. we also know that $D' \overset{\tau}{\hookrightarrow} \textbf{\textit{forked}}(A_i); N_j$ and therefore that $\{\!|(A'+B'+C'+D'+E'); \pi|\!\} \xrightarrow{\tau} \{\!|\textbf{\textit{forked}}(A_i); N_j; \pi|\!\}$. Finally, property (B.1) yields that $\{\!|\textbf{\textit{forked}}(A_i); N_j; \pi|\!\} \smile \{\!|A_i, N_j; \pi|\!\}$.

Case 3.2 $\{\!|A|\!\} \xrightarrow{\alpha_i} \{\!|A_i|\!\}$, $\{\!|M; \pi|\!\} \xrightarrow{\gamma_k} \{\!|\textbf{\textit{forked}}(C_k); \pi|\!\}$ where $\alpha_i = rev(\gamma_k)$ and $\{\!|A, M; \pi|\!\} \xrightarrow{\tau} \{\!|A_i, \textbf{\textit{forked}}(C_k); \pi|\!\}$. We also know that $E' \overset{\tau}{\hookrightarrow} \textbf{\textit{forked}}(A_i); \textbf{\textit{forked}}(C_k)$ and therefore that $\{\!|(A' + B' + C' + D' +$

$E'); \pi \}\!\} \xrightarrow{\tau} \{\!\{ \boldsymbol{forked}(A_i); \boldsymbol{forked}(C_k); \pi \}\!\}$. Finally, property (B.1) yields that $\{\!\{ \boldsymbol{forked}(A_i); \boldsymbol{forked}(C_k); \pi \}\!\} \smile \{\!\{ A_i, \boldsymbol{forked}(C_k); \pi \}\!\}$.

If we let the right hand side $\{\!\{ (A' + B' + C' + D' + E'); \pi \}\!\}$ move first, we can use the above argument in the reverse direction. The conclusion is that $\{\!\{ A, M; \pi \}\!\} \smile \{\!\{ (A' + B' + C' + D' + E'); \pi \}\!\}$, and we are done.

∎

Proposition B.4 ($\mathcal{E}_2$ is sound) *The expansion law $\mathcal{E}_2$ is sound with respect to strong process congruence.*

Proof

We want to prove that $\boldsymbol{forked}(A); \boldsymbol{forked}(B) \equiv \boldsymbol{forked}(A' + B' + C')$ which due to definition 3.7 follows if we can prove $\{\!\{ \boldsymbol{forked}(A); \boldsymbol{forked}(B); \pi \}\!\} \smile \{\!\{ \boldsymbol{forked}(A' + B' + C'); \pi \}\!\}$. Using the property (B.1) in lemma B.1 and the congruence property of $\smile$, we can bring the left hand side and right hand side of this equation into more convenient forms. Left hand side:

$$\begin{aligned}
&\{\!\{ \boldsymbol{forked}(A); \boldsymbol{forked}(B); \pi \}\!\} \\
&\smile \{\!\{ A, \boldsymbol{forked}(B); \pi \}\!\} \\
&\smile \{\!\{ A, B, \pi \}\!\}
\end{aligned}$$

and right hand side:

$$\begin{aligned}
&\{\!\{ \boldsymbol{forked}(A' + B' + C'); \pi \}\!\} \\
&\smile \{\!\{ (A' + B' + C'), \pi \}\!\}
\end{aligned}$$

So the remaining job is to show $\{\!\{ A, B, \pi \}\!\} \smile \{\!\{ (A' + B' + C'), \pi \}\!\}$. But, since $\smile$ is a congruence, it will be enough to show that $\{\!\{ A, B \}\!\} \smile \{\!\{ A' + B' + C' \}\!\}$ (Note how close this is to the CCS expansion law in [Mil89] page 96).

We show this by showing that if one of the programs can perform an action, then the other program can perform the same action, and the two resulting programs are equivalent. First, we let the left hand side $\{\!\{ A, B \}\!\}$ perform an action. There are three cases.

Case 1 $\{\!\{ A \}\!\} \xrightarrow{\alpha_i} \{\!\{ A_i \}\!\}$ and $\{\!\{ A, B \}\!\} \xrightarrow{\alpha_i} \{\!\{ A_i, B \}\!\}$. We also know that $A' \xhookrightarrow{\alpha_i} \boldsymbol{forked}(A_i); \boldsymbol{forked}(B)$ and therefore that $\{\!\{ A' + B' + C' \}\!\} \xrightarrow{\alpha_i} \{\!\{ \boldsymbol{forked}(A_i); \boldsymbol{forked}(B) \}\!\}$. Property (B.1) yields that

$\{|\boldsymbol{forked}(A_i); \boldsymbol{forked}(B)|\} \smile \{|A_i, \boldsymbol{forked}(B)|\}$ and property (B.2) further yields that
$\{|A_i, \boldsymbol{forked}(B)|\} \smile \{|A_i, B|\}$.
So transitivity yields that $\{|\boldsymbol{forked}(A_i); \boldsymbol{forked}(B)|\} \smile \{|A_i, B|\}$.

Case 2 $\{|B|\} \xrightarrow{\beta_j} \{|B_j|\}$ and $\{|A, B|\} \xrightarrow{\beta_j} \{|A, B_j|\}$. We also know that $B' \overset{\beta_j}{\hookrightarrow} \boldsymbol{forked}(A); \boldsymbol{forked}(B_j)$ and therefore that
$\{|A' + B' + C'|\} \xrightarrow{\beta_j} \{|\boldsymbol{forked}(A); \boldsymbol{forked}(B_j)|\}$. Property (B.1) yields that $\{|\boldsymbol{forked}(A); \boldsymbol{forked}(B_j)|\} \smile \{|A, \boldsymbol{forked}(B_j)|\}$ and property (B.2) further yields that
$\{|A, \boldsymbol{forked}(B_j)|\} \smile \{|A, B_j|\}$.
So transitivity yields that $\{|\boldsymbol{forked}(A); \boldsymbol{forked}(B_j)|\} \smile \{|A, B_j|\}$.

Case 3 $\{|A|\} \xrightarrow{\alpha_i} \{|A_i|\}$, $\{|B|\} \xrightarrow{\beta_j} \{|B_j|\}$ where $\alpha_i = rev(\beta_j)$ and $\{|A, B|\} \xrightarrow{\tau} \{|A_i, B_j|\}$. We also know that $C' \overset{\tau}{\hookrightarrow} \boldsymbol{forked}(A_i); \boldsymbol{forked}(B_j)$ and therefore that $\{|A' + B' + C'|\} \xrightarrow{\tau} \{|\boldsymbol{forked}(A_i); \boldsymbol{forked}(B_j)|\}$. Property (B.1) yields that $\{|\boldsymbol{forked}(A_i); \boldsymbol{forked}(B_j)|\} \smile \{|A_i, \boldsymbol{forked}(B_j)|\}$ and property (B.2) further yields that $\{|A_i, \boldsymbol{forked}(B_j)|\} \smile \{|A_i, B_j|\}$. So transitivity yields that
$\{|\boldsymbol{forked}(A_i); \boldsymbol{forked}(B_j)|\} \smile \{|A_i, B_j|\}$.

If we let the right hand side $\{|A' + B' + C'|\}$ move first, we can use the above argument in the reverse direction. The conclusion is that $\{|A, B|\} \smile \{|A' + B' + C'|\}$, and we are done.

■

Lemma B.5 (Nested *forked*s) *For any normal form N, there exists a simple normal form A such that:*

$$\vdash \boldsymbol{forked}(N) = \boldsymbol{forked}(A)$$

Proof

We use induction on the depth n of N.

<u>**Base case $n = 0$:**</u> In this case $N = \boldsymbol{nil}$, and since $\boldsymbol{nil}$ is already on simple normal form, we have our $A = \boldsymbol{nil}$, and surely $\vdash \boldsymbol{forked}(\boldsymbol{nil}) = \boldsymbol{forked}(\boldsymbol{nil})$.

<u>**Induction step $n > 0$:**</u> Assume that N has the form:

$$N \;=\; \sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j)$$

Then we can perform the following deductions:

$$\begin{array}{l}
\boldsymbol{forked}(N) \\
= \\
\boldsymbol{forked}(\sum_i \alpha_i; N_i + \sum_j \beta_j; \boldsymbol{forked}(B_j)) \\
\textit{– unfolding } N \\
= \\
\boldsymbol{forked}(\sum_i \alpha_i; N_i + \sum_j \beta_j; B_j) \\
\textit{– axiom (4.17)} \\
= \\
\boldsymbol{forked}(\sum_i \alpha_i; \boldsymbol{forked}(N_i) + \sum_j \beta_j; B_j) \\
\textit{– axiom (4.17)} \\
= \\
\boldsymbol{forked}(\sum_i \alpha_i; \boldsymbol{forked}(A_i) + \sum_j \beta_j; B_j) \\
\textit{– induction hypothesis applied to} \\
\textit{– } \boldsymbol{forked}(N_i).\ A_i \textit{ is on} \\
\textit{– simple normal form} \\
= \\
\boldsymbol{forked}(\sum_i \alpha_i; A_i + \sum_j \beta_j; B_j) \\
\quad \textit{– axiom (4.17)}
\end{array}$$

The latter argument to ***forked*** is on simple normal form, and so we are done.

∎

Lemma B.6 (Sequential *forked*s) *For any simple normal forms A and B, there exists a simple normal form C such that:*

$$\vdash \boldsymbol{forked}(A); \boldsymbol{forked}(B) = \boldsymbol{forked}(C)$$

Proof

Use induction on the sum n of depths of A and B. The base cases are those where the depth of either A or B is zero (A or B is ***nil***).

Base case $n = depth(A)$ **or** $n = depth(B)$**.** In case one of A or B is ***nil*** one can use axioms (4.18), (4.8) and (4.9) to obtain the result. For example if A is ***nil***:

$$
\begin{array}{ll}
\boldsymbol{forked}(A); \boldsymbol{forked}(B) & \\
= & \\
\boldsymbol{forked}(\boldsymbol{nil}); \boldsymbol{forked}(B) & \textit{– unfolding of } A \\
= & \\
\boldsymbol{nil}; \boldsymbol{forked}(B) & \textit{– axiom (4.18)} \\
= & \\
\boldsymbol{forked}(B) & \textit{– axiom (4.9)}
\end{array}
$$

Induction step $n > depth(A)$ **and** $n > depth(B)$**.** In this case, neither A or B is ***nil***. Use expansion law $\mathcal{E}_2$ (definition 4.9).

Let:

$$
\begin{array}{rcl}
A & = & \sum_i \alpha_i; A_i \\
B & = & \sum_j \beta_j; B_j
\end{array}
$$

Then:

$$
\begin{array}{l}
\boldsymbol{forked}(A); \boldsymbol{forked}(B) \\
= \\
\boldsymbol{forked}(\sum_i \alpha_i; A_i); \boldsymbol{forked}(\sum_j \beta_j; B_j) \\
\textit{– unfolding of } A \textit{ and } B \\
= \\
\boldsymbol{forked}(\\
\quad \sum_i \alpha_i; \boldsymbol{forked}(A_i); \boldsymbol{forked}(B) + \\
\quad \sum_j \beta_j; \boldsymbol{forked}(A); \boldsymbol{forked}(B_j) + \\
\quad \sum_{\alpha_i = rev(\beta_j)} \tau; \boldsymbol{forked}(A_i); \boldsymbol{forked}(B_j)) \\
\textit{– expansion law } \mathcal{E}_2 \\
= \\
\boldsymbol{forked}(\\
\quad \sum_i \alpha_i; \boldsymbol{forked}(C_i) + \\
\quad \sum_j \beta_j; \boldsymbol{forked}(D_j) + \\
\quad \sum_{\alpha_i = rev(\beta_j)} \tau; \boldsymbol{forked}(E_{i,j})) \\
\textit{– induction hypothesis yielding new simple normal forms} \\
\textit{– } C_i,\ D_j \textit{ and } E_{i,j} \\
= \\
\boldsymbol{forked}(\sum_i \alpha_i; C_i + \sum_j \beta_j; D_j + \sum_{\alpha_i = rev(\beta_j)} \tau; E_{i,j}) \\
\textit{– axiom (4.17) applied repeatedly}
\end{array}
$$

■

Lemma B.7 (Normal forms for a subcategory) *For any simple normal form A and normal form M (M different from* ***nil****), there exists a normal form N such that:*

$$\vdash \boldsymbol{forked}(A); M = N$$

Proof

Use induction on the sum n of the depth of A and M with the base cases where the depth of A is zero (recall that M is different from ***nil*** when applying this lemma, also in the inductive step).

Base case $n = depth(M)$ $(depth(A) = 0)$:

$$\begin{array}{ll} \boldsymbol{forked}(\boldsymbol{nil}); M & \\ = & \\ \boldsymbol{nil}; M & \text{– } \textit{axiom (4.18)} \\ = & \\ M & \text{– } \textit{axiom (4.9)} \end{array}$$

Induction step $n > depth(M)$ $(depth(A) > 0)$:

Let:

$$\begin{array}{rcl} A & = & \sum_i \alpha_i; A_i \\ M & = & \sum_j \beta_j; M_j + \sum_k \gamma_k; \boldsymbol{forked}(C_k) \end{array}$$

and:

$$\begin{array}{rcl} A' & = & \sum_i \alpha_i; A'' \\ & & \textit{where} \quad A'' = \boldsymbol{forked}(A_i); M \\ B' & = & \sum_j \beta_j; B'' \\ & & \textit{where} \quad B'' = \boldsymbol{forked}(A); M_j \\ C' & = & \sum_k \gamma_k; C'' \\ & & \textit{where} \quad C'' = \boldsymbol{forked}(A); \boldsymbol{forked}(C_k) \\ D' & = & \sum_{\alpha_i = rev(\beta_j)} \tau; D'' \\ & & \textit{where} \quad D'' = \boldsymbol{forked}(A_i); M_j \\ E' & = & \sum_{\alpha_i = rev(\gamma_k)} \tau; E'' \\ & & \textit{where} \quad E'' = \boldsymbol{forked}(A_i); \boldsymbol{forked}(C_k) \end{array}$$

Then using the expansion law $\mathcal{E}_1$ (definition 4.7) we get:

$$\boldsymbol{forked}(A); M \quad = \quad A' + B' + C' + D' + E'$$

Now, each of the summands S' is on the form $\sum_n act_n; S''$. Due to the induction hypothesis, the summand A'' reduces to a term on normal form. This is also the case for B'' and D'' in case M_j is different from ***nil***. If M_j on the other hand is ***nil***, B'' and D'' reduce to respectively $\boldsymbol{forked}(A)$ and $\boldsymbol{forked}(A_i)$ due to axiom (4.8).

Finally, due to lemma 4.13, the summands C'' and E'' each reduce to a term of the form $\boldsymbol{forked}(G)$ where G is on simple normal form.

To conclude, each S'' reduces to a term in normal form or to a term on the form $\boldsymbol{forked}(G)$ where G is on simple normal form. Therefore the summation $A' + B' + C' + D' + E'$ reduces to a normal form.

■

Lemma B.8 (Normal forms for a subcategory) *For any two normal forms M, N, there exists a normal form K such that:*

$$\vdash M; N = K$$

Proof We use induction on the depth n (length of longest path $\alpha_1; \alpha_2; \ldots$) of M.

Base case $n = 0$:

$M; N$
$=$
$\boldsymbol{nil}; N$ – *unfolding M*
$=$
N – *axiom (4.9)*

Induction step $n > 0$: Use the normal form lemma B.7

$M; N$

$=$

$(\sum_i \alpha_i; M_i + \sum_k \gamma_k; \boldsymbol{forked}(C_k)); N$

– *unfolding* M *which is on normal form*

$=$

$\sum_i \alpha_i; M_i; N + \sum_k \gamma_k; \boldsymbol{forked}(C_k); N$

– *axiom (4.11)*

$=$

$\sum_i \alpha_i; K_i + \sum_k \gamma_k; \boldsymbol{forked}(C_k); N$

– *induction hypothesis applied to* $M_i; N$

– *yielding normal form* K_i

$=$

$\sum_i \alpha_i; K_i + \sum_k \gamma_k; L_k$

– *normal form lemma applied to* $\boldsymbol{forked}(C_k); N$

– *yielding normal form* L_k

∎

Appendix C

Proofs Concerning Refinement Logic

First of all, we need a simple compositionality result.

Lemma C.1 *For any processes* $p_1, \ldots, p_n \in \mathcal{L}$*, and formulae* $\psi_1, \ldots, \psi_n \in \Psi_o$*, for any environment* $\rho \in Env$*, and for any operator* $Op \in \{\sum_i \alpha_i;\_, \_;\_, \boldsymbol{fork}(\_), \boldsymbol{forked}(\_), (a)\_, \{a\}\_\}$ *with arity* n*,*

$$\begin{array}{ll} \textit{Whenever} & p_i \in [\![\psi_i]\!]\rho, \textit{ for } i \in \{1, \ldots, n\} \\ \textit{then} & Op(p_1, \ldots, p_n) \in [\![Op(\psi_1, \ldots, \psi_n)]\!]\rho \end{array}$$

Proof

The result follows easily from the fact that $Op(p_1, \ldots, p_n) \equiv Op(p_1, \ldots, p_n)$ and then by applying definition (6.15).

∎

We also need the following lemma about substitution.

Lemma C.2 *For any formulae* $\psi, \psi' \in \Psi_o$ *and for any environment* $\rho \in Env$*,*

$$[\![\psi]\!]\rho[[\![\psi']\!]\rho/x] = [\![\psi[\psi'/x]]\!]\rho$$

Proof

Left for the reader.

∎

We also need the following lemma about the evaluation of recursive processes. Recall that we only consider well-guarded processes where the recursion variable is guarded.

Lemma C.3 *$Config[\boldsymbol{fix}\, x \cdot p] \xrightarrow{\alpha} C_{fix}$ for some configuration C_{fix} if and only if $Config[p] \xrightarrow{\alpha} C_p$ and $C_{fix} = C_p[\boldsymbol{fix}\, x \cdot p/x]$ for some configuration C_p.*

Proof

Left for the reader.

■

We need some lemmas concerned with window manipulations. We shall use the notation $C\backslash a$ for restricting the window of the configuration C with the channel a. Likewise we use the notation $C \oplus a$ for extending the window. That is, Let C be the configuration $(W, K) \triangleright P$. Then $C\backslash a = ((W, K) \triangleright P)\backslash a \stackrel{def}{=} (W - \{a\}, K) \triangleright P$ and $C \oplus a = ((W, K) \triangleright P) \oplus a \stackrel{def}{=} (W \cup \{a\}, K) \triangleright P$. We shall then need the following lemma.

Lemma C.4 *For any process $p \in \mathcal{L}$ and channel $a \in Chan$,*

$$Config[\{a\}p] \smile Config[p]\backslash a$$

Proof Let $(W_{ap}, K_{ap}) \stackrel{def}{=} Env[\{a\}p]$ and $(W_p, K_p) \stackrel{def}{=} Env[p]$. We assume that $a \in K_p$. Then

$$
\begin{array}{ll}
Config[\{a\}p] & \\
= & \\
(W_{ap}, K_{ap}) \triangleright \{\!|\{a\}p; \pi|\!\} & \\
\smile & \\
(W_{ap}, K_{ap} \cup \{a\}) \triangleright \{\!|p; \pi|\!\} & \textit{– easy to show} \\
= & \\
(W_p - \{a\}, K_p) \triangleright \{\!|p; \pi|\!\} & \textit{– since } W_{ap} = W_p - \{a\} \textit{ and} \\
 & \textit{– since } K_{ap} \cup \{a\} = K_p \\
= & \\
Config[p]\backslash a &
\end{array}
$$

■

Lemma C.5 *For any configurations $C, C' \in$ Con, action $\alpha \in \mathcal{A}$ and channel $a \in$ Chan,*

$$\begin{array}{lll} \textit{Whenever} & C \xrightarrow{\alpha} C' & \\ \textit{then} & C \oplus a \xrightarrow{\alpha} C' \oplus a & \\ \textit{and} & C \backslash a \xrightarrow{\alpha} C' \backslash a & \text{provided } \alpha \neq a \end{array}$$

Proof

Left for the reader.

∎

Lemma C.6 *For any configurations $C_1, C_2 \in$ Con, and channel $a \in$ Chan,*

$$\begin{array}{ll} \textit{Whenever} & C_1 \in_\rho C_2 \\ \textit{then} & C_1 \backslash a \in_\rho C_2 \backslash a \end{array}$$

Proof

Assume that $C_1 \in_\rho C_2$. Then $C_1 \smile \mathit{config}[p;\pi]$ and $C_2 \overset{\circ}{\smile} \mathit{config}[q;\pi]$ for some $p \in \mathcal{L}$ and $q \in \mathcal{L}_w$ such that $p \in [\![\varphi[q]]\!]\rho$ (or $C_1 \smile \mathit{config}[p]$ and $C_2 \overset{\circ}{\smile} \mathit{config}[q]$. We leave this case to the reader since it is similar). Let $(W_{ap}, K_{ap}) \overset{def}{=} \mathit{Env}[\{a\}p]$ and $(W_p, K_p) \overset{def}{=} \mathit{Env}[p]$. Now, we can perform the following deduction:

$$\begin{array}{ll} C_1 \backslash a & \\ \smile & \\ \mathit{config}[p;\pi] \backslash a & \text{– \textit{easy to show}} \\ = & \\ (W_p - \{a\}, K_p) \triangleright \{\!|p;\pi|\!\} & \\ = & \\ (W_{ap}, K_{ap} \cup \{a\}) \triangleright \{\!|p;\pi|\!\} & \text{– \textit{since} } W_p - \{a\} = W_{ap} \text{ \textit{and}} \\ & \text{– \textit{since} } K_p = K_{ap} \cup \{a\} \\ \smile & \\ (W_{ap}, K_{ap}) \triangleright \{\!|\{a\}p;\pi|\!\} & \text{– \textit{lemma (C.4)}} \\ = & \\ \mathit{config}[\{a\}p;\pi] & \end{array}$$

In a similar way we can show that $C_2 \backslash a \overset{\circ}{\smile} \mathit{config}[\{a\}q;\pi]$. Finally, since $p \in [\![\varphi[q]]\!]\rho$, lemma (C.1) yields that $\{a\}p \in [\![\varphi[\{a\}q]]\!]\rho$.

∎

We can now prove the core lemma.

Lemma C.7 *For any processes $p \in \mathcal{L}$ and $q \in \mathcal{L}_g$, and for any environment $\rho \in Env$, such that $p \in \llbracket\varphi[q]\rrbracket\rho$ it holds that for all $\alpha \in Act$,*

1. *Whenever $Config[p] \xrightarrow{\alpha} C_1$ for some C_1 then $Config[q] \xrightarrow{\alpha} C_2$ for some C_2 and $C_1 \in_\rho C_2$*
2. *Whenever $Config[q] \xrightarrow{\alpha} C_2$ for some C_2 then $Config[p] \xrightarrow{\alpha} C_1$ for some C_1 and $C_1 \in_\rho C_2$*

Proof

We use induction over the structure of q.

Case $q = \sum_{i \in I} \alpha_i; q_i$**.** We assume that $p \in \llbracket\varphi[\sum_i \alpha_i; q_i]\rrbracket\rho = \llbracket\sum_i \alpha_i; \varphi[q_i]\rrbracket\rho$. This means that $p \equiv \sum_i \alpha_i; r_i$ for some processes $\{r_i \mid i \in I\}$ such that $r_i \in \llbracket\varphi[q_i]\rrbracket\rho$. Let $\rho_q \stackrel{def}{=} Env[\sum_i \alpha_i; q_i]$ and $\rho_r \stackrel{def}{=} Env[\sum_i \alpha_i; r_i]$. Assume now that $Config[p] \xrightarrow{\alpha} C$. Then since $p \equiv \sum_i \alpha_i; r_i$, also $Config[\sum_i \alpha_i; r_i] = \rho_r \triangleright \{\!|\sum_i \alpha_i; r_i; \pi|\!\} \xrightarrow{\alpha} \rho_r \triangleright \{\!|r_i; \pi|\!\}$, where $C \smile \rho_r \triangleright \{\!|r_i; \pi|\!\}$. Note that $\alpha = \alpha_i$ for some $i \in I$. Due to lemma (5.10) we further have that $\rho_r \triangleright \{\!|r_i; \pi|\!\} \smile Config[r_i]$, so $C \smile Config[r_i]$. On the other hand, we have that $Config[\sum_i \alpha_i; q_i] = \rho_q \triangleright \{\!|\sum_i \alpha_i; q_i; \pi|\!\} \xrightarrow{\alpha} \rho_q \triangleright \{\!|q_i; \pi|\!\}$. Due to lemma (5.10) we further have that $\rho_q \triangleright \{\!|q_i; \pi|\!\} \stackrel{\circ}{\smile} Config[q_i]$. Finally, our assumption was that $r_i \in \llbracket\varphi[q_i]\rrbracket\rho$.

A symmetric argument starting with q ends this case.

Case $q = q_1; q_2$**.** We assume that $p \in \llbracket\varphi[q_1; q_2]\rrbracket\rho = \llbracket\varphi[q_1]; \varphi[q_2]\rrbracket\rho$. This means that $p \equiv r_1; r_2$ for some processes r_1, r_2 such that $r_i \in \llbracket\varphi[q_i]\rrbracket\rho$.

Let $\rho_q \stackrel{def}{=} Env[q_1; q_2]$ and $\rho_r \stackrel{def}{=} Env[r_1; r_2]$. Assume now that $Config[p] \xrightarrow{\alpha} C_p$. Then since $p \equiv r_1; r_2$, also $Config[r_1; r_2] \xrightarrow{\alpha} C_r$, where $C_p \smile C_r$.

Now there are two cases to analyse, depending on whether r_1 during the evaluation reaches a state r_1' where $Stop(r_1')$ (r_1 gets exhausted) or not.

Case 1: r_1 gets exhausted. In this case, we know that $Config[r_1] \xrightarrow{\pi} \ldots$, and since $r_1 \in \llbracket\varphi[q_1]\rrbracket\rho$ the induction hypothesis yields that $Config[q_1] \xrightarrow{\pi} \ldots$. We can therefore easily show that $Config[r_1; r_2] \smile Config[\boldsymbol{forked}(r_1); r_2]$ and that $Config[q_1; q_2] \smile Config[\boldsymbol{forked}(q_1); q_2]$. Since $Config[r_1; r_2] \xrightarrow{\alpha} C_r$ we then conclude that $Config[\boldsymbol{forked}(r_1); r_2] = \rho_r \triangleright \{\!|\boldsymbol{forked}(r_1); r_2; \pi|\!\} \Longrightarrow \rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi, r_1|\!\} \xrightarrow{\alpha} C_{Fr}$ where $C_r \smile C_{Fr}$ and thereby $C_p \smile C_{Fr}$. Now suppose we can show that $Config[\boldsymbol{forked}(q_1); q_2] \xrightarrow{\alpha} C_{Fq}$ where $C_{Fr} \in_\rho C_{Fq}$. Then the result namely follows since $\mathit{Config}[\boldsymbol{forked}(q_1); q_2] \smile \mathit{Config}[q_1; q_2]$ and consequently $\mathit{Config}[q_1; q_2] \xrightarrow{\alpha} C_q$ where $C_{Fq} \smile C_q$ and consequently using lemma (6.20) (since $C_p \smile C_{Fr}$) $C_p \in_\rho C_q$.

So we shall show that $\mathit{Config}[\boldsymbol{forked}(q_1); q_2] \xrightarrow{\alpha} C_{Fq}$ where $C_{Fr} \in_\rho C_{Fq}$. There are three cases to consider, based on whether r_1 acts alone, r_2 acts alone or r_1 and r_2 communicates in the transition: $\rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi, r_1|\!\} \xrightarrow{\alpha} C_{Fr}$.

r_1 acts alone. In this case we know that $\rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi, r_1|\!\} \xrightarrow{\alpha} \rho'_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi|\!\} \cup \{\!|r'_1|\!\} \cup R = C_{Fr}$ for some R. But then of course $\rho_r \triangleright \{\!|r_1|\!\} \xrightarrow{\alpha} \rho'_r \triangleright \{\!|r'_1|\!\} \cup R$ and consequently $\rho_r \triangleright \{\!|r_1; \pi|\!\} \xrightarrow{\alpha} \rho'_r \triangleright \{\!|r'_1; \pi|\!\} \cup R$. Due to lemma (5.10) we further have that $\rho_r \triangleright \{\!|r_1; \pi|\!\} \smile \mathit{Config}[r_1]$ so $\mathit{Config}[r_1] \xrightarrow{\alpha} C_{r1}$ where $\rho'_r \triangleright \{\!|r'_1; \pi|\!\} \cup R \smile C_{r1}$. Since $r_1 \in [\![\varphi[q_1]]\!]\rho$ the induction hypothesis yields that $\mathit{Config}[q_1] \xrightarrow{\alpha} C_{q1}$ where $C_{r1} \in_\rho C_{q1}$. Due to lemma (5.10) we further have that $\mathit{Config}[q_1] \overset{\circ}{\smile} \rho_q \triangleright \{\!|q_1; \pi|\!\}$, so $\rho_q \triangleright \{\!|q_1; \pi|\!\} \xrightarrow{\alpha} \rho'_q \triangleright \{\!|q'_1; \pi|\!\} \cup Q$ for some Q where $C_{q1} \overset{\circ}{\smile} \rho'_q \triangleright \{\!|q'_1; \pi|\!\} \cup Q$. We can from this transition obtain that $\rho_q \triangleright \{\!|q_1|\!\} \xrightarrow{\alpha} \rho'_q \triangleright \{\!|q'_1|\!\} \cup Q$. We can extend this to obtain $\mathit{Config}[\boldsymbol{forked}(q_1); q_2] = \rho_q \triangleright \{\!|\boldsymbol{forked}(q_1); q_2; \pi|\!\} \Longrightarrow \rho_q \triangleright \{\!|\boldsymbol{nil}; q_2; \pi, q_1|\!\} \xrightarrow{\alpha} \rho'_q \triangleright \{\!|\boldsymbol{nil}; q_2; \pi|\!\} \cup \{\!|q'_1|\!\} \cup Q = C_{Fq}$. Since now $\rho'_r \triangleright \{\!|r'_1; \pi|\!\} \cup R \smile C_{r1}$ and $\rho'_q \triangleright \{\!|q'_1; \pi|\!\} \cup Q \overset{\circ}{\smile} C_{q1}$ and $C_{r1} \in_\rho C_{q1}$ we have that $\rho'_r \triangleright \{\!|r'_1; \pi|\!\} \cup R \smile \mathit{config}[s; \pi]$ and that $\rho'_q \triangleright \{\!|q'_1; \pi|\!\} \cup Q \overset{\circ}{\smile} \mathit{config}[t; \pi]$ such that $s \in [\![\varphi[t]]\!]\rho$. We can consequently easily show that $C_{Fr} = \rho'_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi|\!\} \cup \{\!|r'_1|\!\} \cup R \smile \mathit{config}[\boldsymbol{forked}(s); r_2; \pi]$ and that $C_{Fq} = \rho'_q \triangleright \{\!|\boldsymbol{nil}; q_2; \pi|\!\} \cup \{\!|q'_1|\!\} \cup Q \overset{\circ}{\smile} \mathit{config}[\boldsymbol{forked}(t); q_2; \pi]$, where due to lemma (C.1): $\boldsymbol{forked}(s); r_2 \in [\![\varphi[\boldsymbol{forked}(t); q_2]]\!]\rho$.

r_2 acts alone. In this case we know that $\rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi, r_1|\!\} \xrightarrow{\alpha} \rho'_r \triangleright A \cup \{\!|r_1|\!\} = C_{Fr}$ for some A. But then of course $\rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi|\!\} \xrightarrow{\alpha} \rho'_r \triangleright A$. Due to lemma (5.10) we further have that $\rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi|\!\} \smile \mathit{Config}[r_2]$ so $\mathit{Config}[r_2] \xrightarrow{\alpha} C_{r2}$ where $\rho'_r \triangleright A \smile C_{r2}$. Since $r_2 \in [\![\varphi[q_2]]\!]\rho$ the induction hypothesis yields that $\mathit{Config}[q_2] \xrightarrow{\alpha} C_{q2}$ where $C_{r2} \in_\rho C_{q2}$. Due to lemma (5.10) we further have that $\mathit{Config}[q_2] \overset{\circ}{\smile} \rho_q \triangleright \{\!|\boldsymbol{nil}; q_2; \pi|\!\}$, so $\rho_q \triangleright \{\!|\boldsymbol{nil}; q_2; \pi|\!\} \xrightarrow{\alpha} \rho'_q \triangleright B$ for some B where $C_{q2} \overset{\circ}{\smile} \rho'_q \triangleright B$. We can extend this to obtain that $\mathit{Config}[\boldsymbol{forked}(q_1); q_2] = \rho_q \triangleright \{\!|\boldsymbol{forked}(q_1); q_2; \pi|\!\} \Longrightarrow \rho_q \triangleright \{\!|\boldsymbol{nil}; q_2; \pi, q_1|\!\} \xrightarrow{\alpha} \rho'_q \triangleright B \cup \{\!|q_1|\!\} = C_{Fq}$. Since now $\rho'_r \triangleright A \smile C_{r2}$ and $\rho'_q \triangleright B \overset{\circ}{\smile} C_{q2}$ and $C_{r2} \in_\rho C_{q2}$ we have that $\rho'_r \triangleright A \smile \mathit{config}[s]$ and that $\rho'_q \triangleright B \overset{\circ}{\smile} \mathit{config}[t]$ (or $\rho'_r \triangleright A \smile \mathit{config}[s; \pi]$ and $\rho'_q \triangleright B \overset{\circ}{\smile} \mathit{config}[t; \pi]$) such that $s \in [\![\varphi[t]]\!]\rho$. We can consequently easily show that $C_{Fr} = \rho'_r \triangleright A \cup \{\!|r_1|\!\} \smile \mathit{config}[\boldsymbol{forked}(r_1); s]$ and that $C_{Fq} = \rho'_q \triangleright B \cup \{\!|q_1|\!\} \overset{\circ}{\smile} \mathit{config}[\boldsymbol{forked}(q_1); t]$ (or $C_{Fr} \smile$

$\mathit{config}[\boldsymbol{forked}(r_1); s; \pi]$ and $C_{Fq} \stackrel{\circ}{\smile} \mathit{config}[\boldsymbol{forked}(q_1); t; \pi]$), where due to lemma (C.1): $\boldsymbol{forked}(r_1); s \in [\![\varphi[\boldsymbol{forked}(q_1); t]]\!]\rho$.

r_1 and r_2 communicates. In this case we know that $\rho_r \triangleright \{\!|r_1|\!\} \xrightarrow{c} \rho_r^1 \triangleright \{\!|r_1'|\!\} \cup R_1$ and that $\rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi|\!\} \xrightarrow{rev(c)} \rho_r^2 \triangleright \{\!|r_2'; \pi|\!\} \cup R_2$ and that consequently $\rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi, r_1|\!\} \xrightarrow{\tau} \rho_r^1 \cup \rho_r^2 \triangleright \{\!|r_2'; \pi, r_1'|\!\} \cup R_1 \cup R_2 = C_{Fr}$ for some R_1 and R_2.
We can thus investigate individually the c-transition and the $rev(c)$-transition.
We have that $\rho_r \triangleright \{\!|r_1|\!\} \xrightarrow{c} \rho_r^1 \triangleright \{\!|r_1'|\!\} \cup R_1$, and consequently $\rho_r \triangleright \{\!|r_1; \pi|\!\} \xrightarrow{c} \rho_r^1 \triangleright \{\!|r_1'; \pi|\!\} \cup R_1$. Due to lemma (5.10) we further have that $\rho_r \triangleright \{\!|r_1; \pi|\!\} \smile \mathit{Config}[r_1]$ so $\mathit{Config}[r_1] \xrightarrow{c} C_{r1}$ where $\rho_r^1 \triangleright \{\!|r_1'; \pi|\!\} \cup R_1 \smile C_{r1}$. Since $r_1 \in [\![\varphi[q_1]]\!]\rho$ the induction hypothesis yields that $\mathit{Config}[q_1] \xrightarrow{c} C_{q1}$ where $C_{r1} \in_\rho C_{q1}$. Due to lemma (5.10) we further have that $\mathit{Config}[q_1] \stackrel{\circ}{\smile} \rho_q \triangleright \{\!|q_1; \pi|\!\}$, so $\rho_q \triangleright \{\!|q_1; \pi|\!\} \xrightarrow{c} \rho_q^1 \triangleright \{\!|q_1'; \pi|\!\} \cup Q_1$ for some Q_1 where $C_{q1} \stackrel{\circ}{\smile} \rho_q^1 \triangleright \{\!|q_1'; \pi|\!\} \cup Q_1$. We can from this transition obtain that $\rho_q \triangleright \{\!|q_1|\!\} \xrightarrow{c} \rho_q^1 \triangleright \{\!|q_1'|\!\} \cup Q_1$. Since now $\rho_r^1 \triangleright \{\!|r_1'; \pi|\!\} \cup R_1 \smile C_{r1}$ and $\rho_q^1 \triangleright \{\!|q_1'; \pi|\!\} \cup Q_1 \stackrel{\circ}{\smile} C_{q1}$ and $C_{r1} \in_\rho C_{q1}$ we have that $\rho_r^1 \triangleright \{\!|r_1'; \pi|\!\} \cup R_1 \smile \mathit{config}[s_1; \pi]$ and that $\rho_q^1 \triangleright \{\!|q_1'; \pi|\!\} \cup Q_1 \stackrel{\circ}{\smile} \mathit{config}[t_1; \pi]$ such that $s_1 \in [\![\varphi[t_1]]\!]\rho$. We can consequently easily show that $\rho_r^1 \triangleright \{\!|r_1'|\!\} \cup R_1 \smile \mathit{config}[s_1]$ and that $\rho_q^1 \triangleright \{\!|q_1'|\!\} \cup Q_1 \stackrel{\circ}{\smile} \mathit{config}[t_1]$ such that $s_1 \in [\![\varphi[t_1]]\!]\rho$.
Concerning the $rev(c)$-transition we have that $\rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi|\!\} \xrightarrow{rev(c)} \rho_r^2 \triangleright \{\!|r_2'; \pi|\!\} \cup R_2$. Due to lemma (5.10) we further have that $\rho_r \triangleright \{\!|\boldsymbol{nil}; r_2; \pi|\!\} \smile \mathit{Config}[r_2]$ so $\mathit{Config}[r_2] \xrightarrow{rev(c)} C_{r2}$ where $\rho_r^2 \triangleright \{\!|r_2'; \pi|\!\} \cup R_2 \smile C_{r2}$. Since $r_2 \in [\![\varphi[q_2]]\!]\rho$ the induction hypothesis yields that $\mathit{Config}[q_2] \xrightarrow{rev(c)} C_{q2}$ where $C_{r2} \in_\rho C_{q2}$. Due to lemma (5.10) we further have that $\mathit{Config}[q_2] \stackrel{\circ}{\smile} \rho_q \triangleright \{\!|\boldsymbol{nil}; q_2; \pi|\!\}$, so $\rho_q \triangleright \{\!|\boldsymbol{nil}; q_2; \pi|\!\} \xrightarrow{rev(c)} \rho_q^2 \triangleright \{\!|q_2'; \pi|\!\} \cup Q_2$ for some Q_2 where $C_{q2} \stackrel{\circ}{\smile} \rho_q^2 \triangleright \{\!|q_2'; \pi|\!\} \cup Q_2$. Since now $\rho_r^2 \triangleright \{\!|r_2'; \pi|\!\} \cup R_2 \smile C_{r2}$ and $\rho_q^2 \triangleright \{\!|q_2'; \pi|\!\} \cup Q_2 \stackrel{\circ}{\smile} C_{q2}$ and $C_{r2} \in_\rho C_{q2}$ we have that $\rho_r^2 \triangleright \{\!|r_2'; \pi|\!\} \cup R_2 \smile \mathit{config}[s_2; \pi]$ and that $\rho_q^2 \triangleright \{\!|q_2'; \pi|\!\} \cup Q_2 \stackrel{\circ}{\smile} \mathit{config}[t_2; \pi]$ such that $s_2 \in [\![\varphi[t_2]]\!]\rho$.
We finally put these c and $rev(c)$ transitions together:
$\mathit{Config}[\boldsymbol{forked}(q_1); q_2] = \rho_q \triangleright \{\!|\boldsymbol{forked}(q_1); q_2; \pi|\!\} \Longrightarrow$
$\rho_q \triangleright \{\!|\boldsymbol{nil}; q_2; \pi, q_1|\!\} \xrightarrow{\tau} \rho_q^1 \cup \rho_q^2 \triangleright \{\!|q_2'; \pi|\!\} \cup Q_2 \cup \{\!|q_1'|\!\} \cup Q_1 = C_{Fq}$.
We can then show that $C_{Fr} = \rho_r^1 \cup \rho_r^2 \triangleright \{\!|r_2'; \pi, r_1'|\!\} \cup R_1 \cup R_2 \smile \mathit{config}[\boldsymbol{forked}(s_1); s_2; \pi]$ and that $C_{Fq} = \rho_q^1 \cup \rho_q^2 \triangleright \{\!|q_2'; \pi, q_1'|\!\} \cup Q_1 \cup Q_2 \stackrel{\circ}{\smile} \mathit{config}[\boldsymbol{forked}(t_1); t_2; \pi]$, where due to lemma (C.1): $\boldsymbol{forked}(s_1); s_2 \in [\![\varphi[\boldsymbol{forked}(t_1); t_2]]\!]\rho$.

Case 2: r_1 does not get exhausted. In this case we know that $Config[r_1;r_2] = \rho_r \triangleright \{|r_1;r_2;\pi|\} \xrightarrow{\alpha} \rho_r' \triangleright \{|r_1';r_2;\pi|\} \cup R = C_r \smile C_p$. But then $\rho_r \triangleright \{|r_1;\pi|\} \xrightarrow{\alpha} \rho_r' \triangleright \{|r_1';\pi|\} \cup R$. Due to lemma (5.10) we further have that $\rho_r \triangleright \{|r_1;\pi|\} \smile Config[r_1]$, so $Config[r_1] \xrightarrow{\alpha} C_{r1}$ where $\rho_r' \triangleright \{|r_1';\pi|\} \cup R \smile C_{r1}$. Since $r_1 \in [\![\varphi[q_1]]\!]\rho$ the induction hypothesis yields that $Config[q_1] \xrightarrow{\alpha} C_{q1}$ where $C_{r1} \in_\rho C_{q1}$. Due to lemma (5.10) we further have that $Config[q_1] \overset{\circ}{\smile} \rho_q \triangleright \{|q_1;\pi|\}$, so $\rho_q \triangleright \{|q_1;\pi|\} \xrightarrow{\alpha} \rho_q' \triangleright \{|q_1';\pi|\} \cup Q$ for some Q where $C_{q1} \overset{\circ}{\smile} \rho_q' \triangleright \{|q_1';\pi|\} \cup Q$. We can extend this to obtain $Config[q_1;q_2] = \rho_q \triangleright \{|q_1;q_2;\pi|\} \xrightarrow{\alpha} \rho_q' \triangleright \{|q_1';q_2;\pi|\} \cup Q = C_q$. Since now $\rho_r' \triangleright \{|r_1';\pi|\} \cup R \smile C_{r1}$ and $\rho_q' \triangleright \{|q_1';\pi|\} \cup Q \overset{\circ}{\smile} C_{q1}$ and $C_{r1} \in_\rho C_{q1}$ we have that $\rho_r' \triangleright \{|r_1';\pi|\} \cup R \smile config[s;\pi]$ and that $\rho_q' \triangleright \{|q_1';\pi|\} \cup Q \overset{\circ}{\smile} config[t;\pi]$ such that $s \in [\![\varphi[t]]\!]\rho$. We can consequently easily show that $C_r = \rho_r' \triangleright \{|r_1';r_2;\pi|\} \cup R \smile config[s;r_2;\pi]$ and that $C_q = \rho_q' \triangleright \{|q_1';q_2;\pi|\} \cup Q \overset{\circ}{\smile} config[t;q_2;\pi]$, where due to lemma (C.1): $s;r_2 \in [\![\varphi[t;q_2]]\!]\rho$.

A symmetric argument starting with q ends this case.

Case $q = \boldsymbol{fork}(s)$. We assume that $p \in [\![\varphi[\boldsymbol{fork}(s)]]\!]\rho = [\![\boldsymbol{fork}(\varphi[s])]\!]\rho$. This means that $p \equiv \boldsymbol{fork}(r)$ for some process r such that $r \in [\![\varphi[s]]\!]\rho$. Let $\rho_q \overset{def}{=} Env[\boldsymbol{fork}(s)]$ and $\rho_r \overset{def}{=} Env[\boldsymbol{fork}(r)]$. Assume now that $Config[p] \xrightarrow{\alpha} C$. Then since $p \equiv \boldsymbol{fork}(r)$, also $Config[\boldsymbol{fork}(r)] = \rho_r \triangleright \{|\boldsymbol{fork}(r);\pi|\} \xrightarrow{\alpha} \rho_r \triangleright \{|\boldsymbol{nil};\pi,r|\}$, where $C \smile \rho_r \triangleright \{|\boldsymbol{nil};\pi,r|\}$. Note that $\alpha = \tau$. It is easy to see that $\rho_r \triangleright \{|\boldsymbol{nil};\pi,r|\} \smile \rho_r \triangleright \{|\boldsymbol{forked}(r);\pi|\} = Config[\boldsymbol{forked}(r)]$, so $C \smile Config[\boldsymbol{forked}(r)]$. On the other hand, we have that $Config[\boldsymbol{fork}(s)] = \rho_q \triangleright \{|\boldsymbol{fork}(s);\pi|\} \xrightarrow{\tau} \rho_q \triangleright \{|\boldsymbol{nil};\pi,s|\}$. We further have that $\rho_q \triangleright \{|\boldsymbol{nil};\pi,s|\} \overset{\circ}{\smile} Config[\boldsymbol{forked}(s)]$. Finally, our assumption was that $r \in [\![\varphi[s]]\!]\rho$. So using lemma (C.1) then gives $\boldsymbol{forked}(r) \in [\![\boldsymbol{forked}(\varphi[s])]\!]\rho = [\![\varphi[\boldsymbol{forked}(s)]]\!]\rho$.

A symmetric argument starting with q ends this case.

Case $q = \boldsymbol{forked}(s)$. We assume that $p \in [\![\varphi[\boldsymbol{forked}(s)]]\!]\rho = [\![\boldsymbol{forked}(\varphi[s])]\!]\rho$. This means that $p \equiv \boldsymbol{forked}(r)$ for some process r such that $r \in [\![\varphi[s]]\!]\rho$. Let $\rho_q \overset{def}{=} Env[\boldsymbol{forked}(s)]$ and $\rho_r \overset{def}{=} Env[\boldsymbol{forked}(r)]$. Assume now that $Config[p] \xrightarrow{\alpha} C_p$. Then since $p \equiv \boldsymbol{forked}(r)$, also $Config[\boldsymbol{forked}(r)] \xrightarrow{\alpha} C_{Fr}$ where $C_p \smile C_{Fr}$. Now, there are two cases to consider, depending on whether $\alpha = \pi$ or not.

Case 1: $\alpha = \pi$. In this case $Config[\boldsymbol{forked}(r)] = \rho_r \triangleright \{|\boldsymbol{forked}(r);\pi|\} \Longrightarrow \rho_r \triangleright \{|\boldsymbol{nil};\pi,r|\} \xrightarrow{\pi} \rho_r \triangleright \{|\boldsymbol{nil},r|\} \smile config[r]$. But also $Config[\boldsymbol{forked}(s)] = \rho_q \triangleright \{|\boldsymbol{forked}(s);\pi|\} \Longrightarrow \rho_q \triangleright \{|\boldsymbol{nil};\pi,s|\} \xrightarrow{\pi} \rho_q \triangleright \{|\boldsymbol{nil},s|\} \smile config[s]$. Finally, our assumption was that $r \in [\![\varphi[s]]\!]\rho$.

Case 2: $\alpha \neq \pi$**.** In this case $Config[\boldsymbol{forked}(r)] = \rho_r \triangleright \{|\boldsymbol{forked}(r); \pi|\} \Longrightarrow \rho_r \triangleright \{|\boldsymbol{nil}; \pi|\} \cup \{|r|\} \xrightarrow{\alpha} \rho'_r \triangleright \{|\boldsymbol{nil}; \pi|\} \cup \{|r'|\} \cup R = C_{Fr}$, where $C_p \smile C_{Fr}$. But then we easily see that $Config[r] = \rho_r \triangleright \{|r; \pi|\} \xrightarrow{\alpha} \rho'_r \triangleright \{|r'; \pi|\} \cup R = C_r$. Since $r \in [\![\varphi[s]]\!]\rho$ the induction hypothesis yields that $Config[s] = \rho_q \triangleright \{|s; \pi|\} \xrightarrow{\alpha} \rho'_q \triangleright \{|s'; \pi|\} \cup Q = C_q$ where $C_r \in_\rho C_q$. We can extend this to obtain $Config[\boldsymbol{forked}(s)] = \rho_q \triangleright \{|\boldsymbol{forked}(s); \pi|\} \Longrightarrow \rho_q \triangleright \{|\boldsymbol{nil}; \pi|\} \cup \{|s|\} \xrightarrow{\alpha} \rho'_q \triangleright \{|\boldsymbol{nil}; \pi|\} \cup \{|s'|\} \cup Q = C_{Fq}$. Since now $C_r \in_\rho C_q$ where $C_r = \rho'_r \triangleright \{|r'; \pi|\} \cup R$ and $C_q = \rho'_q \triangleright \{|s'; \pi|\} \cup Q$ we get that $\rho'_r \triangleright \{|r'; \pi|\} \cup R \smile config[v; \pi]$ and that $\rho'_q \triangleright \{|s'; \pi|\} \cup Q \stackrel{\circ}{\smile} config[w; \pi]$ such that $v \in [\![\varphi[w]]\!]\rho$. We can consequently easily show that $C_{Fr} = \rho'_r \triangleright \{|\boldsymbol{nil}; \pi|\} \cup \{|r'|\} \cup R \smile config[\boldsymbol{forked}(v); \pi]$ and that $C_{Fq} = \rho'_q \triangleright \{|\boldsymbol{nil}; \pi|\} \cup \{|s'|\} \cup Q \stackrel{\circ}{\smile} config[\boldsymbol{forked}(w); \pi]$, where due to lemma (C.1): $\boldsymbol{forked}(v) \in [\![\varphi[\boldsymbol{forked}(w)]]\!]\rho$.

A symmetric argument starting with q ends this case.

Case $q = (a)s$**.** We assume that $p \in [\![\varphi[(a)s]]\!]\rho = [\![(a)\varphi[s]]\!]\rho$. This means that $p \equiv (a)r$ for some process r such that $r \in [\![\varphi[s]]\!]\rho$. Let $(W_q, K_q) \stackrel{def}{=} Env[(a)s]$ and $(W_r, K_r) \stackrel{def}{=} Env[(a)r]$. Assume now that $Config[p] \xrightarrow{\alpha} C_p$. Then since $p \equiv (a)r$, also $Config[(a)r] = (W_r, K_r) \triangleright \{|(a)r; \pi|\} \xrightarrow{\tau} (W_r, K_r \cup \{k\}) \triangleright \{|r[k/a]; \pi|\} = C_r$, where $C_p \smile C_r$. Note that $\alpha = \tau$. Of course k does not belong to K_r. On the other hand, $Config[(a)s] = (W_q, K_q) \triangleright \{|(a)s; \pi|\} \xrightarrow{\tau} (W_q, K_q \cup \{k'\}) \triangleright \{|s[k'/a]; \pi|\} = C_q$, for some k' not in K_q. Now due to lemma (C.4) $C_r = (W_r, K_r \cup \{k\}) \triangleright \{|r[k/a]; \pi|\} \smile (W_r, K_r) \triangleright \{|\{k\}r[k/a]; \pi|\} = config[\{k\}r[k/a]; \pi]$. The last equality holds since $(W_r, K_r) = Env[(a)r] = Env[\{k\}r[k/a]]$. Likewise, we get that $C_q = (W_q, K_q \cup \{k'\}) \triangleright \{|s[k'/a]; \pi|\} \stackrel{\circ}{\smile} config[\{k'\}s[k'/a]; \pi]$. We can finally show that $\{k\}r[k/a] \in [\![\varphi[\{k'\}s[k'/a]]]\!]\rho$: First, note that since alpha-conversion (change of bound names) results in equivalent terms (we shall not prove this), $\{k\}r[k/a] \equiv \{a\}r$ where we had that $r \in [\![\varphi[s]]\!]\rho$. Definition (6.15) then gives that $\{k\}r[k/a] \in [\![\varphi[\{a\}s]]\!]\rho = [\![\varphi[\{k'\}s[k'/a]]]\!]\rho$. The last equality holds since alpha-conversion does not change the denotation (we shall not prove this).

A symmetric argument starting with q ends this case.

Case $q = \{a\}s$**.** We assume that $p \in [\![\varphi[\{a\}s]]\!]\rho = [\![\{a\}\varphi[s]]\!]\rho$. This means that $p \equiv \{a\}r$ for some process r such that $r \in [\![\varphi[s]]\!]\rho$. Assume now that $Config[p] \xrightarrow{\alpha} C_p$. Then since $p \equiv \{a\}r$, also $Config[\{a\}r] \xrightarrow{\alpha} C_{ar}$, where $C_p \smile C_{ar}$. Note that $\alpha \neq a$. Due to lemma (C.4), $Config[\{a\}r] \smile Config[r] \setminus a$, so $Config[r] \setminus a \xrightarrow{\alpha} C_{r \setminus a} \smile C_{ar}$. Now due to lemma (C.5), consequently $Config[r] \xrightarrow{\alpha} C_{r \setminus a} \oplus a$. Since $r \in [\![\varphi[s]]\!]\rho$ the induction hypothesis yields that $Config[s] \xrightarrow{\alpha} C_q$ where $C_{r \setminus a} \oplus a \in_\rho C_q$. But then lemma (C.5) gives that $\mathit{Config}[s]\backslash a \xrightarrow{\alpha} C_q\backslash a$. Lemma (C.4) implies that $\mathit{Config}[s]\backslash a \stackrel{\circ}{\smile} \mathit{Config}[\{a\}s]$, so $\mathit{Config}[\{a\}s] \xrightarrow{\alpha} C_{aq} \stackrel{\circ}{\smile} C_q\backslash a$. Now, since $C_{r\backslash a}\oplus a \in_\rho C_q$ lemma (C.6) yields that $C_{r\backslash a} \in_\rho C_q\backslash a$. Finally, since $C_p \smile C_{r\backslash a}$ (transitivity) and since $C_{aq} \stackrel{\circ}{\smile} C_q\backslash a$ we can conclude that $C_p \in_\rho C_{aq}$ due to lemma (6.20).

Note that lemmas C.4 and 6.20 are used beyond their scope since the involved configurations are open with free variables, and the lemmas only hold for closed terms. The lemmas do, however, extend to the open case also.

A symmetric argument starting with q ends this case.

Case $q = x$**.** This case violates the assumption that q is guarded, and it needs therefore not be considered.

Case $q = \boldsymbol{\mathit{fix}}\, x \cdot s$**.** We assume that $p \in [\![\varphi[\boldsymbol{\mathit{fix}}\, x \cdot s]]\!]\rho = [\![\nu\, x \cdot \varphi[s]]\!]\rho$. Due to definition (6.15), and Tarski's theorem, $[\![\nu x \cdot \varphi[s]]\!]\rho$ is a (maximal) fixpoint to the function $\lambda S \cdot [\![\varphi[s]]\!]\rho[S/x]$. So $p \in [\![\nu\, x \cdot \varphi[s]]\!]\rho = [\![\varphi[s]]\!]\rho[([\![\nu\, x \cdot \varphi[s]]\!]\rho)/x]$. Let $\rho' \stackrel{def}{=} \rho[([\![\nu\, x \cdot \varphi[s]]\!]\rho)/x]$. Since $\boldsymbol{\mathit{fix}}\, x \cdot s$ is guarded, s is guarded, and we can therefore apply the induction hypothesis. We consider the two cases where respectively p and q begins with a move.

p **moves.** Assume that $\mathit{Config}[p] \xrightarrow{\alpha} C_p$. Due to the induction hypothesis then also $\mathit{Config}[s] \xrightarrow{\alpha} C_s$ where $C_p \in_{\rho'} C_s$. The latter means that $C_p \smile \mathit{config}[p']$ and $C_s \stackrel{\circ}{\smile} \mathit{config}[s']$ (or $C_p \smile \mathit{config}[p';\pi]$ and $C_s \stackrel{\circ}{\smile} \mathit{config}[s';\pi]$) for some processes p' and s' such that $p' \in [\![\varphi[s']]\!]\rho' = [\![\varphi[s'][\nu\, x \cdot \varphi[s]/x]]\!]\rho = [\![\varphi[s'[\boldsymbol{\mathit{fix}}\, x \cdot s/x]]]\!]\rho$. The second last equality is due to lemma (C.2). Now since $\mathit{Config}[s] \xrightarrow{\alpha} C_s$ we have due to lemma (C.3) that $\mathit{Config}[\boldsymbol{\mathit{fix}}\, x \cdot s] \xrightarrow{\alpha} C_s[\boldsymbol{\mathit{fix}}\, x \cdot s/x]$. We can finally show that $C_p \in_\rho C_s[\boldsymbol{\mathit{fix}}\, x \cdot s/x]$ as follows: First recall that $C_p \smile \mathit{config}[p']$ (or $C_p \smile \mathit{config}[p';\pi]$). Second, since $C_s \stackrel{\circ}{\smile} \mathit{config}[s']$ (or $C_s \stackrel{\circ}{\smile} \mathit{config}[s';\pi]$) we have that $C_s[\boldsymbol{\mathit{fix}} x \cdot s/x] \stackrel{\circ}{\smile} \mathit{config}[s'[\boldsymbol{\mathit{fix}} x \cdot s/x]]$ (or $C_s[\boldsymbol{\mathit{fix}}\, x \cdot s/x] \stackrel{\circ}{\smile} \mathit{config}[s'[\boldsymbol{\mathit{fix}}\, x \cdot s/x];\pi]$). Finally recall that $p' \in [\![\varphi[s'[\boldsymbol{\mathit{fix}}\, x \cdot s/x]]]\!]\rho$.

q **moves.** Assume that $\mathit{Config}[\boldsymbol{\mathit{fix}} x \cdot s] \xrightarrow{\alpha} C_{fix}$. Due to lemma (C.3) we get that $\mathit{Config}[s] \xrightarrow{\alpha} C_s$ where $C_{fix} = C_s[\boldsymbol{\mathit{fix}}\, x \cdot s/x]$. By the induction hypothesis, $\mathit{Config}[p] \xrightarrow{\alpha} C_p$ where $C_p \in_{\rho'} C_s$. But then we can show (ending this case) that $C_p \in_\rho C_{fix}$. It follows from the fact that $C_p \in_{\rho'} C_s$. This namely means that $C_p \smile \mathit{config}[p']$ and $C_s \stackrel{\circ}{\smile} \mathit{config}[s']$ (or $C_p \smile \mathit{config}[p';\pi]$ and $C_s \stackrel{\circ}{\smile} \mathit{config}[s';\pi]$) for some processes p' and s' such that $p' \in [\![\varphi[s']]\!]\rho' = [\![\varphi[s'][\nu\, x \cdot \varphi[s]/x]]\!]\rho = [\![\varphi[s'[\boldsymbol{\mathit{fix}} x \cdot s/x]]]\!]\rho$. The second last equality is due to lemma (C.2). Now, since $C_s \stackrel{\circ}{\smile} \mathit{config}[s']$ (or $C_s \stackrel{\circ}{\smile} \mathit{config}[s';\pi]$) we have that $C_{fix} = C_s[\boldsymbol{\mathit{fix}}\, x \cdot s/x] \stackrel{\circ}{\smile}$

$\mathit{config}[s'[\boldsymbol{fix} x \cdot s/x]]$ (or $C_{fix} = C_s[\boldsymbol{fix} x \cdot s/x] \overset{\circ}{\smile} \mathit{config}[s'[\boldsymbol{fix} x \cdot s/x]; \pi]$). Finally recall that $p' \in [\![\varphi[s'[\boldsymbol{fix}\, x \cdot s/x]]]\!]\rho$.

∎

Appendix D

Proofs of Protocol Transitions

This appendix contains partial proofs of the two lemmas 6.30 and 6.32.

D.1 Proof of Protocol Transition Lemma

This section is concerned with the proof of lemma 6.30 with focus on states S_0, S_1 and S_4.

Before proving properties about the individual states of the protocol design, we record some properties about the medium *Tau_Inmed?*. First, we reformulate its definition (just naming subformulae):

$$\begin{aligned}
\textit{Tau_Inmed?} &\stackrel{def}{=} \circ\{inmed?\} \wedge \textit{Always} \\
\textit{Always} &\stackrel{def}{=} \textit{In_Out} \wedge \textit{Out_In} \\
\textit{In_Out} &\stackrel{def}{=} \Box([inmed?] \circ \{outmed!, err!\}) \\
\textit{Out_In} &\stackrel{def}{=} \Box([outmed!, err!] \circ \{inmed?\})
\end{aligned}$$

We then state some properties used in later proofs.

Lemma D.1 *Let* $F \stackrel{def}{=} \lambda X.\ \ll\tau, inmed?\gg \wedge [\tau]X$. *Then:*

1. $\circ\{inmed?\} \Longleftrightarrow \bigvee_{n \in \boldsymbol{N}} F^n(\textit{ff})$
2. $\textit{Always} \Longrightarrow [\tau]\textit{Always}$
3. $\textit{Tau\_Inmed?} \Longleftrightarrow \bigvee_{n \in \boldsymbol{N}}(F^n(\textit{ff}) \wedge \textit{Always})$
4. $F^{n+1}(\textit{ff}) \wedge \textit{Always} \Longleftrightarrow \ll\tau, inmed?\gg \wedge [\tau](F^n(\textit{ff}) \wedge \textit{Always}) \wedge \textit{Always}$
5. $\ll\tau, inmed?\gg \wedge [\tau](F^n(\textit{ff}) \wedge \textit{Always}) \wedge \textit{Always} \Longrightarrow \textit{Tau\_Inmed?}$

Proof

Proof of 1:

The formula $\circ\{inmed?\}$ stands for the minimal fixpoint $\mu X \cdot F(X)$. But since $F(X)$ is continuous this minimal fixpoint equals $\bigvee_{n \in \mathbf{N}} F^n(\mathit{ff})$.

Proof of 2:

$$
\begin{array}{lll}
 & Always & \\
= & In\_Out \wedge Out\_In & \\
= & \Box([inmed?] \circ \{outmed!, err!\}) \wedge \Box([outmed!, err!] \circ \{inmed?\}) & \\
= & \nu X \cdot [inmed?] \circ \{outmed!, err!\} \wedge [ACT]X & \\
 & \quad \wedge & \\
 & \nu X \cdot [outmed!, err!] \circ \{inmed?\} \wedge [ACT]X & \\
\Longrightarrow & [inmed?] \circ \{outmed!, err!\} \wedge [ACT] In\_Out & (\mathbf{UnfoldMax}) \\
 & \quad \wedge & \\
 & [outmed!, err!] \circ \{inmed?\} \wedge [ACT] Out\_In & \\
\Longrightarrow & [\tau](In\_Out \wedge Out\_In) & (\mathbf{And}[\alpha]) \\
= & [\tau] Always &
\end{array}
$$

Proof of 3:

$$
\begin{array}{lll}
 & Tau\_Inmed? & \\
= & \circ\{inmed?\} \wedge Always & \\
\Longleftrightarrow & \bigvee_{n \in \mathbf{N}} F^n(\mathit{ff}) \wedge Always & \text{(lemma (D.1)-1)} \\
\Longleftrightarrow & \bigvee_{n \in \mathbf{N}} (F^n(\mathit{ff}) \wedge Always) &
\end{array}
$$

Proof of 4:

$$
\begin{array}{lll}
 & F^{n+1}(\mathit{ff}) \wedge Always & \\
\Longleftrightarrow & F^{n+1}(\mathit{ff}) \wedge Always \wedge Always & \text{(straightforward)} \\
\Longleftrightarrow & (\ll\tau, inmed?\gg \wedge [\tau] F^n(\mathit{ff})) \wedge [\tau] Always \wedge Always & \text{(lemma (D.1)-2)} \\
\Longleftrightarrow & \ll\tau, inmed?\gg \wedge [\tau](F^n(\mathit{ff}) \wedge Always) \wedge Always & (\mathbf{And}[\alpha])
\end{array}
$$

Proof of 5:

$$
\begin{array}{lll}
 & \ll\tau, inmed?\gg \wedge [\tau](F^n(\mathit{ff}) \wedge Always) \wedge Always & \\
\Longrightarrow & F^{n+1}(\mathit{ff}) \wedge Always & \text{(lemma (D.1)-4)} \\
\Longrightarrow & \mu X \cdot F(X) \wedge Always & \text{(follows from lemma (D.1)-1)} \\
= & Tau\_Inmed? &
\end{array}
$$

∎

$\boxed{S_0 \Longrightarrow [accept?]S_1}$

(1)	$Accept?$	
$=$	$accept?; Inmed!$	
$\Longrightarrow$	$[accept?]Inmed!$	$(\mathbf{Seq}_\alpha[\alpha])$
(2)	$\boldsymbol{forked}(Accept?)$	
$\Longrightarrow$	$[accept?]\boldsymbol{forked}(Inmed!)$	(1), $(\mathbf{Forked}[\alpha])$
(3)	$Tau\_Inmed?$	
$\Longrightarrow$	$\circ\{inmed?\}$	
$\Longrightarrow$	$\ll\tau, inmed?\gg \ \wedge [\tau] \circ \{inmed?\}$	$(\mathbf{UnfoldMin})$
$\Longrightarrow$	$[accept?]\boldsymbol{ff}$	
(4)	$\boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Accept?)$	
$\Longrightarrow$	$[accept?]($	(2,3), $(\mathbf{Seq}_\Phi[c])$
	$\quad(\boldsymbol{forked}(\boldsymbol{ff}); \boldsymbol{forked}(Accept?))$	
	$\qquad\vee$	
	$\quad(\boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Inmed!)))$	
$\Longrightarrow$	$[accept?]\boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Inmed!)$	$(\mathbf{Strict})$
(5)	$Outmed?$	
$=$	$outmed?; Deliver!$	
$\Longrightarrow$	$[accept?]\boldsymbol{ff}$	$(\mathbf{Seq}_\alpha[\beta])$
(6)	$\boldsymbol{forked}(Outmed?); \boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Accept?)$	
$\Longrightarrow$	$[accept?]($	(4,5), $(\mathbf{Seq}_\Phi[c])$
	$\quad(\boldsymbol{forked}(\boldsymbol{ff}); \boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Accept?))$	
	$\qquad\vee$	
	$\quad(\boldsymbol{forked}(Outmed?); \boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Inmed!)))$	
$\Longrightarrow$	$[accept?]$	$(\mathbf{Strict})$
	$\quad\boldsymbol{forked}(Outmed?); \boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Inmed!)$	
(7)	S_0	
$=$	$\{I\}\boldsymbol{forked}(Outmed?); \boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Accept?)$	
$\Longrightarrow$	$[accept?]$	(6), $(\mathbf{Allocated}[\alpha]_1)$
	$\quad\{I\}\boldsymbol{forked}(Outmed?); \boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Inmed!)$	
$=$	$[accept?]S_1$	

$\boxed{S_0 \Longrightarrow \circ\{accept?\}}$

Since S_0 has a minimal fixpoint (in terms of *Tau_Inmed?*) embedded within programming constructs, we use an induction technique based on the fact that the minimal fixpoint (in this case) is equivalent to an infinite disjunction. That is, lemma (D.1)-1 implies that: $\mathit{Tau\_Inmed?} \Longrightarrow \circ\{inmed?\} \Longrightarrow \bigvee_{n \in \mathbf{N}} F^n(\mathit{ff})$. So we can proceed as follows:

$$
\begin{array}{lll}
 & S_0 & \\
= & \{I\}\boldsymbol{forked}(\mathit{Outmed?});\boldsymbol{forked}(\mathit{Tau\_Inmed?});\boldsymbol{forked}(\mathit{Accept?}) & \\
\Longrightarrow & \{I\}\boldsymbol{forked}(\mathit{Outmed?});\boldsymbol{forked}(\bigvee_{n \in \mathbf{N}} F^n(\mathit{ff}));\boldsymbol{forked}(\mathit{Accept?}) & \text{(see above)} \\
\Longrightarrow & \bigvee_{n \in \mathbf{N}}\{I\}\boldsymbol{forked}(\mathit{Outmed?});\boldsymbol{forked}(F^n(\mathit{ff}));\boldsymbol{forked}(\mathit{Accept?}) & (\vee\textbf{-Distr})
\end{array}
$$

We denote the n'th disjunct with Disj_n. To show that S_0 satisfies $\circ\{accept?\}$, we then show that each disjunct does so. That is, we shall for any natural number n show, that:

$$
\begin{array}{lll}
\mathit{Disj}_n & = & \{I\}\boldsymbol{forked}(\mathit{Outmed?});\boldsymbol{forked}(F^n(\mathit{ff}));\boldsymbol{forked}(\mathit{Accept?}) \\
 & \Longrightarrow & \circ\{accept?\}
\end{array}
$$

This we can show by induction on n as follows.

Base case:

$$
\begin{array}{lll}
 & \mathit{Disj}_0 & \\
= & \{I\}\boldsymbol{forked}(\mathit{Outmed?});\boldsymbol{forked}(\mathit{ff});\boldsymbol{forked}(\mathit{Accept?}) & \\
\Longrightarrow & \mathit{ff} & (\textbf{Strict}) \\
\Longrightarrow & \circ\{accept?\} & (\textbf{Empty})
\end{array}
$$

Induction step:

We assume that for any $k \leq n$ that $\mathit{Disj}_k \Longrightarrow \circ\{accept?\}$. Then we proceed as follows:

$$
\begin{array}{lll}
 & \mathit{Disj}_{n+1} & \\
= & \{I\}\boldsymbol{forked}(\mathit{Outmed?});\boldsymbol{forked}(F^{n+1}(\mathit{ff}));\boldsymbol{forked}(\mathit{Accept?}) & \\
= & \{I\}\boldsymbol{forked}(\mathit{Outmed?}); & \\
 & \quad \boldsymbol{forked}(\ll\tau, inmed?\gg\ \wedge [\tau]F^n(\mathit{ff})); & \\
 & \quad \boldsymbol{forked}(\mathit{Accept?}) & \\
\Longrightarrow & \ll\tau, accept?\gg\ \wedge [\tau]\mathit{Disj}_n & \text{(easy to show)} \\
\Longrightarrow & \ll\tau, accept?\gg\ \wedge [\tau] \circ \{accept?\} & \text{(induction hypothesis)} \\
\Longrightarrow & \circ\{accept?\} & (\textbf{UnfoldMin})
\end{array}
$$

$\boxed{S_1 \Longrightarrow \ll\tau\gg}$

Note that:

$$\ll\tau\gg \;=\; <\tau>\, \boldsymbol{tt} \wedge [ACT - \{\tau\}]\boldsymbol{ff} \;=\; <\tau>\, \boldsymbol{tt} \wedge [accept?]\boldsymbol{ff} \wedge [deliver!]\boldsymbol{ff}$$

We just prove $<\tau>\, \boldsymbol{tt}$ and $[accept?]\boldsymbol{ff}$.

$\underline{S_1 \Longrightarrow <\tau>\, \boldsymbol{tt}}$

$$\begin{array}{lll}
(1) & Tau_Inmed? & \\
\Longrightarrow & \circ\{inmed?\} & \\
= & \mu X \cdot \ll\tau, inmed?\gg \wedge [\tau]X & \\
\Longrightarrow & \ll\tau, inmed?\gg \wedge [\tau] \circ \{inmed?\} & (\textbf{UnfoldMin}) \\
\Longrightarrow & \ll\tau, inmed?\gg & \\
= & <\tau, inmed?>\, \boldsymbol{tt} \wedge [ACT - \{\tau, inmed?\}]\boldsymbol{ff} & \\
\Longrightarrow & <\tau>\, \boldsymbol{tt} \vee <inmed?>\, \boldsymbol{tt} & \\
 & & \\
(2) & inmed! & \\
\Longrightarrow & <inmed!>\, \boldsymbol{nil} & (\textbf{Act}<\alpha>) \\
 & & \\
(3) & Inmed! & \\
= & inmed!;\, Ack?_Err? & \\
\Longrightarrow & <inmed!>\, (\boldsymbol{nil};\, Ack?_Err?) & (2),\ (\textbf{Seq}<\alpha>) \\
\Longrightarrow & <inmed!>\, \boldsymbol{tt} & \\
 & & \\
(4) & \boldsymbol{forked}(Inmed!) & \\
\Longrightarrow & <inmed!>\, \boldsymbol{forked}(\boldsymbol{tt}) & (3),\ (\textbf{Forked}<\alpha>) \\
\Longrightarrow & <inmed!>\, \boldsymbol{tt} & \\
 & & \\
(5) & \boldsymbol{forked}(Tau_Inmed?);\, \boldsymbol{forked}(Inmed!) & \\
\Longrightarrow & \boldsymbol{forked}(<\tau>\, \boldsymbol{tt} \vee <inmed?>\, \boldsymbol{tt});\, <inmed!>\, \boldsymbol{tt} & (1{,}4) \\
\Longrightarrow & \boldsymbol{forked}(<\tau>\, \boldsymbol{tt});\, <inmed!>\, \boldsymbol{tt} & (\vee\textbf{-Distr}) \\
 & \quad\vee & \\
 & \boldsymbol{forked}(<inmed?>\, \boldsymbol{tt});\, <inmed!>\, \boldsymbol{tt} & \\
\Longrightarrow & <\tau>\, \boldsymbol{forked}(\boldsymbol{tt});\, <inmed!>\, \boldsymbol{tt} & (\textbf{Seq}<\alpha>),\ (\textbf{Seq}_\Phi <\tau>) \\
 & \quad\vee & \\
 & <\tau>\, \boldsymbol{forked}(\boldsymbol{tt});\, \boldsymbol{tt} & \\
\Longrightarrow & <\tau>\, \boldsymbol{tt} &
\end{array}$$

$$
\begin{array}{lll}
(6) & \textbf{\textit{forked}}(Outmed?); \textbf{\textit{forked}}(Tau\_Inmed?); \textbf{\textit{forked}}(Inmed!) & \\
\Longrightarrow & <\tau> \textbf{\textit{forked}}(Outmed?); \textbf{\textit{tt}} & (5),(\mathbf{Seq_\Phi}\ <\alpha>) \\
\Longrightarrow & <\tau> \textbf{\textit{tt}} &
\end{array}
$$

$$
\begin{array}{lll}
(7) & S_1 & \\
= & \{I\}\textbf{\textit{forked}}(Outmed?); \textbf{\textit{forked}}(Tau\_Inmed?); \textbf{\textit{forked}}(Inmed!) & \\
\Longrightarrow & <\tau> \{I\}\textbf{\textit{tt}} & (6),(\mathbf{Allocated}<\alpha>) \\
\Longrightarrow & <\tau> \textbf{\textit{tt}} &
\end{array}
$$

$\underline{S_1 \Longrightarrow [accept?]\textbf{\textit{ff}}}$

$$
\begin{array}{lll}
(1) & Tau\_Inmed? & \\
\Longrightarrow & [accept?]\textbf{\textit{ff}} & \text{(proved earlier)}
\end{array}
$$

$$
\begin{array}{lll}
(2) & Inmed! & \\
= & inmed!; Ack?\_Err? & \\
\Longrightarrow & [accept?]\textbf{\textit{ff}} & (\mathbf{Seq}_\alpha[\beta])
\end{array}
$$

$$
\begin{array}{lll}
(3) & \textbf{\textit{forked}}(Inmed!) & \\
\Longrightarrow & [accept?]\textbf{\textit{forked}}(\textbf{\textit{ff}}) & (2),\ (\mathbf{Forked}[\alpha]) \\
\Longrightarrow & [accept?]\textbf{\textit{ff}} & (\mathbf{Strict})
\end{array}
$$

$$
\begin{array}{lll}
(4) & \textbf{\textit{forked}}(Tau\_Inmed?); \textbf{\textit{forked}}(Inmed!) & \\
\Longrightarrow & [accept?] & (1,3),\ (\mathbf{Seq_\Phi}[c]) \\
 & \quad ((\textbf{\textit{forked}}(\textbf{\textit{ff}}); \textbf{\textit{forked}}(Inmed!)) \ \vee\ (\textbf{\textit{forked}}(Tau\_Inmed?); \textbf{\textit{ff}})) & \\
\Longrightarrow & [accept?]\textbf{\textit{ff}} & (\mathbf{Strict})
\end{array}
$$

$$
\begin{array}{lll}
(5) & Outmed? & \\
= & outmed?; Deliver! & \\
\Longrightarrow & [accept?]\textbf{\textit{ff}} & (\mathbf{Seq}_\alpha[\beta])
\end{array}
$$

$$
\begin{array}{lll}
(6) & \textbf{\textit{forked}}(Outmed?); \textbf{\textit{forked}}(Tau\_Inmed?); \textbf{\textit{forked}}(Inmed!) & \\
\Longrightarrow & [accept?](& (4,5),\ (\mathbf{Seq_\Phi}[c]) \\
 & \quad (\textbf{\textit{forked}}(\textbf{\textit{ff}}); \textbf{\textit{forked}}(Tau\_Inmed?); \textbf{\textit{forked}}(Inmed!)) & \\
 & \qquad \vee & \\
 & \quad (\textbf{\textit{forked}}(Outmed?); \textbf{\textit{ff}})) & \\
\Longrightarrow & [accept?]\textbf{\textit{ff}} & (\mathbf{Strict})
\end{array}
$$

$$
\begin{array}{lll}
(7) & S1 & \\
= & \{I\}\boldsymbol{forked}(Outmed?); \boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Inmed!) & \\
\Longrightarrow & [accept?]\{I\}\boldsymbol{ff} & (6), (\mathbf{Allocated}[\alpha]_1) \\
\Longrightarrow & [accept?]\boldsymbol{ff} & (\mathbf{Strict})
\end{array}
$$

$\boxed{S_1 \Longrightarrow [\tau](S_1 \vee S_2)}$

$$
\begin{array}{lll}
(1) & \mathit{Tau\_Inmed?} & \\
= & \circ\{inmed?\} \wedge \mathit{Always} & \\
\Longrightarrow & \ll\tau, inmed?\gg \wedge [\tau] \circ \{inmed?\} \wedge \mathit{Always} & (\textbf{UnfoldMin}) \\
\Longrightarrow & [\tau] \circ \{inmed?\} \wedge [\tau]\mathit{Always} & \text{(see above)} \\
\Longrightarrow & [\tau](\circ\{inmed?\} \wedge \mathit{Always}) & (\textbf{And}[\alpha]) \\
= & [\tau]\,\mathit{Tau\_Inmed?} & \\
 & & \\
(2) & \mathit{Inmed!} & \\
= & inmed!; \mathit{Ack?\_Err?} & \\
\Longrightarrow & [\tau]\boldsymbol{\mathit{ff}} & (\textbf{Seq}_\alpha[\beta]) \\
 & & \\
(3) & \boldsymbol{\mathit{forked}}(\mathit{Inmed!}) & \\
\Longrightarrow & [\tau]\boldsymbol{\mathit{forked}}(\boldsymbol{\mathit{ff}}) & (2),\ (\textbf{Forked}[\alpha]) \\
\Longrightarrow & [\tau]\boldsymbol{\mathit{ff}} & (\textbf{Strict}) \\
 & & \\
(4) & \mathit{Tau\_Inmed?} & \\
= & \circ\{inmed?\} \wedge \mathit{In\_Out} \wedge \mathit{Out\_In} & \\
\Longrightarrow & \mathit{In\_Out} \wedge \mathit{Out\_In} & \\
= & \Box([inmed?] \circ \{outmed!, err!\}) \wedge \Box([outmed!, err!] \circ \{inmed?\}) & \\
= & \nu X \cdot [inmed?] \circ \{outmed!, err!\} \wedge [ACT]X & \\
 & \quad \wedge & \\
 & \nu X \cdot [outmed!, err!] \circ \{inmed?\} \wedge [ACT]X & \\
\Longrightarrow & [inmed?] \circ \{outmed!, err!\} \wedge [ACT]\mathit{In\_Out} & (\textbf{UnfoldMax}) \\
 & \quad \wedge & \\
 & [outmed!, err!] \circ \{inmed?\} \wedge [ACT]\mathit{Out\_In} & \\
\Longrightarrow & [inmed?](\circ\{outmed!, err!\} \wedge \mathit{In\_Out} \wedge \mathit{Out\_In}) & (\textbf{And}[\alpha]) \\
= & [inmed?]\,\mathit{Tau\_Outmed!\_Err!} & \\
 & & \\
(5) & \mathit{Inmed!} & \\
= & inmed!; \mathit{Ack?\_Err?} & \\
\Longrightarrow & [inmed!]\mathit{Ack?\_Err?} & (\textbf{Seq}_\alpha[\alpha]) \\
 & & \\
(6) & \boldsymbol{\mathit{forked}}(\mathit{Inmed!}) & \\
\Longrightarrow & [inmed!]\boldsymbol{\mathit{forked}}(\mathit{Ack?\_Err?}) & (5),\ (\textbf{Forked}[\alpha])
\end{array}
$$

$$
\begin{array}{lll}
(7) & \textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); \textbf{\textit{forked}}(\mathit{Inmed!}) & \\
\Longrightarrow & [\tau]((\textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); \textbf{\textit{forked}}(\mathit{Inmed!})) \vee & (1,3,4,6), (\mathbf{Seq}_{\Phi}[\tau]) \\
& \quad (\textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); \textbf{\textit{ff}}) \vee & \\
& \quad (\textbf{\textit{forked}}(\mathit{Tau\_Outmed!\_Err!}); \textbf{\textit{forked}}(\mathit{Ack?\_Err?}))) & \\
\Longrightarrow & [\tau]((\textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); \textbf{\textit{forked}}(\mathit{Inmed!})) \vee & (\mathbf{Strict}) \\
& \quad (\textbf{\textit{forked}}(\mathit{Tau\_Outmed!\_Err!}); \textbf{\textit{forked}}(\mathit{Ack?\_Err?}))) &
\end{array}
$$

$$
\begin{array}{lll}
(8) & \mathit{Outmed?} & \\
= & \mathit{outmed?}; \mathit{Deliver!} & \\
\Longrightarrow & [\tau]\textbf{\textit{ff}} & (\mathbf{Seq}_{\alpha}[\beta])
\end{array}
$$

$$
\begin{array}{lll}
(9) & \textbf{\textit{forked}}(\mathit{Outmed?}); \textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); \textbf{\textit{forked}}(\mathit{Inmed!}) & \\
\Longrightarrow & [\tau]((\textbf{\textit{forked}}(\textbf{\textit{ff}}); \textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); \textbf{\textit{forked}}(\mathit{Inmed!})) & (7,8), (\mathbf{Seq}_{\Phi}[\tau]) \\
& \qquad \vee & \\
& \quad (\textbf{\textit{forked}}(\mathit{Outmed?}); (& \\
& \qquad (\textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); \textbf{\textit{forked}}(\mathit{Inmed!})) & \\
& \qquad\qquad \vee & \\
& \qquad (\textbf{\textit{forked}}(\mathit{Tau\_Outmed!\_Err!}); \textbf{\textit{forked}}(\mathit{Ack?\_Err?}))))) & \\
\Longrightarrow & [\tau]((\textbf{\textit{forked}}(\mathit{Outmed?}); & (\mathbf{Strict}), (\vee\textbf{-Distr}) \\
& \quad \textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); & \\
& \quad \textbf{\textit{forked}}(\mathit{Inmed!})) & \\
& \qquad \vee & \\
& \quad (\textbf{\textit{forked}}(\mathit{Outmed?}); & \\
& \quad \textbf{\textit{forked}}(\mathit{Tau\_Outmed!\_Err!}); & \\
& \quad \textbf{\textit{forked}}(\mathit{Ack?\_Err?}))) &
\end{array}
$$

$$
\begin{array}{lll}
(10) & S_1 & \\
= & \{I\}\textbf{\textit{forked}}(\mathit{Outmed?}); \textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); \textbf{\textit{forked}}(\mathit{Inmed!}) & \\
\Longrightarrow & [\tau]\{I\}((\textbf{\textit{forked}}(\mathit{Outmed?}); & (9), (\mathbf{Allocated}[\alpha]_1) \\
& \qquad \textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); & \\
& \qquad \textbf{\textit{forked}}(\mathit{Inmed!})) & \\
& \qquad\quad \vee & \\
& \qquad (\textbf{\textit{forked}}(\mathit{Outmed?}); & \\
& \qquad \textbf{\textit{forked}}(\mathit{Tau\_Outmed!\_Err!}); & \\
& \qquad \textbf{\textit{forked}}(\mathit{Ack?\_Err?}))) & \\
\Longrightarrow & [\tau]((\{I\}\textbf{\textit{forked}}(\mathit{Outmed?}); & (\vee\textbf{-Distr}) \\
& \qquad \textbf{\textit{forked}}(\mathit{Tau\_Inmed?}); & \\
& \qquad \textbf{\textit{forked}}(\mathit{Inmed!})) & \\
& \quad \vee & \\
& \quad (\{I\}\textbf{\textit{forked}}(\mathit{Outmed?}); & \\
& \qquad \textbf{\textit{forked}}(\mathit{Tau\_Outmed!\_Err!}); & \\
& \qquad \textbf{\textit{forked}}(\mathit{Ack?\_Err?}))) & \\
= & [\tau](S_1 \vee S_2) &
\end{array}
$$

$\boxed{S_4 \Longrightarrow \odot\, S_0}$

First observe that:

$$\begin{array}{ll} & S_4 \\ = & \{I\}\boldsymbol{forked}(Ack!);\boldsymbol{forked}(Tau\_Inmed?);\boldsymbol{forked}(Ack?\_Err?) \\ \Longrightarrow & \{I\}\boldsymbol{forked}(Ack!);\boldsymbol{forked}(\bigvee_{n\,\in\,\mathbf{N}}(F^n(\boldsymbol{ff})\ \wedge\ Always));\boldsymbol{forked}(Ack?\_Err?) \quad \text{(lemma (D.1)-3)} \\ \Longrightarrow & \bigvee_{n\,\in\,\mathbf{N}}\{I\}\boldsymbol{forked}(Ack!);\boldsymbol{forked}(F^n(\boldsymbol{ff})\ \wedge\ Always);\boldsymbol{forked}(Ack?\_Err?) \quad (\vee\textbf{-Distr}) \end{array}$$

We denote the n'th disjunct with $Disj_n$. To show that S_4 satisfies $\odot\, S_0$, we show that each disjunct does so. That is, we shall for any natural number n show, that:

$$Disj_n \ = \ \{I\}\boldsymbol{forked}(Ack!);\boldsymbol{forked}(F^n(\boldsymbol{ff})\ \wedge\ Always);\boldsymbol{forked}(Ack?\_Err?) \ \Longrightarrow\ \odot\, S_0$$

This we can show by induction on n as follows.

<u>Base case:</u>

$$\begin{array}{ll} & Disj_0 \\ = & \{I\}\boldsymbol{forked}(Ack!);\boldsymbol{forked}(\boldsymbol{ff}\ \wedge\ Always);\boldsymbol{forked}(Ack?\_Err?) \\ \Longrightarrow & \boldsymbol{ff} \quad (\textbf{Strict}) \\ \Longrightarrow & \odot\, S_0 \quad (\textbf{Empty}) \end{array}$$

<u>Induction step:</u>

We assume for any $k \leq n$ that $Disj_k \Longrightarrow \odot\, S_0$. Then we proceed as follows:

$$\begin{array}{ll} & Disj_{n+1} \\ = & \{I\}\boldsymbol{forked}(Ack!);\boldsymbol{forked}(F^{n+1}(\boldsymbol{ff})\ \wedge\ Always);\boldsymbol{forked}(Ack?\_Err?) \\ \Longrightarrow & \{I\}\boldsymbol{forked}(Ack!); \quad \text{(lemma (D.1)-4)} \\ & \quad \boldsymbol{forked}(\ll\tau, inmed?\gg\ \wedge [\tau](F^n(\boldsymbol{ff})\ \wedge\ Always)\ \wedge\ Always); \\ & \quad \boldsymbol{forked}(Ack?\_Err?) \\ \Longrightarrow & \ll\tau\gg\ \wedge [\tau](T_1\ \vee\ T_2) \end{array}$$

where

$$\begin{array}{lll} T_1 & = & \{I\}\boldsymbol{forked}(Ack!);\boldsymbol{forked}(F^n(\boldsymbol{ff})\ \wedge\ Always);\boldsymbol{forked}(Ack?\_Err?) \\ T_2 & = & \{I\}\boldsymbol{forked}(Outmed?); \\ & & \quad \boldsymbol{forked}(\ll\tau, inmed?\gg\ \wedge [\tau](F^n(\boldsymbol{ff})\ \wedge\ Always)\ \wedge\ Always); \\ & & \quad \boldsymbol{forked}(Accept?) \end{array}$$

This is easy to show. Now provided we can show that $T_1 \Longrightarrow \odot\, S_0$ and that $T_2 \Longrightarrow \odot\, S_0$ and thereby that $T_1\ \vee\ T_2 \Longrightarrow \odot\, S_0$ we can conclude that:

$$
\begin{array}{lll}
 & Disj_{n+1} & \\
\Longrightarrow & \ll\tau\gg \wedge [\tau](T_1 \vee T_2) & \text{(from above)} \\
\Longrightarrow & \ll\tau\gg \wedge [\tau] \odot S_0 & \text{(from above)} \\
\Longrightarrow & S_0 \vee (\ll\tau\gg \wedge [\tau] \odot S_0) & \text{(obvious)} \\
\Longrightarrow & \odot S_0 & (\textbf{UnfoldMin})
\end{array}
$$

Now $T_1 \Longrightarrow \odot S_0$ by our assumption, so it remains to show that $T_2 \Longrightarrow \odot S_0$:

$$
\begin{array}{lll}
 & T_2 & \\
= & \{I\}\boldsymbol{forked}(Outmed?); & \\
 & \quad \boldsymbol{forked}(\ll\tau, inmed?\gg \wedge [\tau](F^n(\boldsymbol{ff}) \wedge Always) \wedge Always); & \\
 & \quad \boldsymbol{forked}(Accept?) & \\
\Longrightarrow & \{I\}\boldsymbol{forked}(Outmed?); \boldsymbol{forked}(Tau\_Inmed?); \boldsymbol{forked}(Accept?) & \text{(lemma (D.1)-5)} \\
= & S_0 & \\
\Longrightarrow & S_0 \vee (\ll\tau\gg \wedge [\tau] \odot S_0) & \text{(obvious)} \\
\Longrightarrow & \odot S_0 & (\textbf{UnfoldMin})
\end{array}
$$

D.2 Proof of Medium Transition Lemma

This section is concerned with the proof of lemma 6.32 with focus on state M_1.

$\boxed{M_1 \Longrightarrow \ll outmed!, err!, \tau \gg}$

Note that:

$$\begin{aligned}
&\ll outmed!, err!, \tau \gg \\
&= <outmed!, err!, \tau> \boldsymbol{tt} \wedge \ [ACT - \{outmed!, err!, \tau\}]\boldsymbol{ff} \\
&= <outmed!, err!, \tau> \boldsymbol{tt} \wedge \ [inmed?]\boldsymbol{ff}
\end{aligned}$$

We prove each of the conjuncts.

$\underline{M_1 \Longrightarrow <outmed!, err!, \tau> \boldsymbol{tt}}$

A proof of for example $M_1 \Longrightarrow <outmed!> \boldsymbol{tt}$ is enough. We shall, however, for illustration prove that also $err!$ and τ are possible actions.

(1)	*Outmed!_Err!_Crash!*	
$=$	$outmed!; Unstable + err!; Unstable + crash!; \boldsymbol{nil}$	
$\Longrightarrow$	$<outmed!> Unstable \wedge \ <err!> Unstable \wedge \ <crash!> \boldsymbol{nil}$	(**Sum**$<\alpha>$)
(2)	*Crash?*	
$=$	$crash?; Err!$	
$\Longrightarrow$	$<crash?> Err!$	(**Sum**$<\alpha>$)
(3)	$\boldsymbol{forked}(Outmed!\_Err!\_Crash!)$	
$\Longrightarrow$	$<outmed!> \boldsymbol{forked}(Unstable) \wedge \ <err!> \boldsymbol{forked}(Unstable)$	(1), (**Forked**$<\alpha>$)
(4)	$\boldsymbol{forked}(Outmed!\_Err!\_Crash!); Crash?$	
$\Longrightarrow$	$<outmed!> \boldsymbol{forked}(Unstable); Crash? \wedge$	(1,2,3), (**Seq**$<\alpha>$), ($\mathbf{Seq}_\Phi <\tau>$)
	$<err!> \boldsymbol{forked}(Unstable); Crash? \wedge$	
	$<\tau> \boldsymbol{forked}(\boldsymbol{nil}); Err!$	
$\Longrightarrow$	$<outmed!> \boldsymbol{tt} \wedge \ <err!> \boldsymbol{tt} \wedge \ <\tau> \boldsymbol{tt}$	(**All**)
(5)	M_1	
$=$	$\{crash\}\boldsymbol{forked}(Outmed!\_Err!\_Crash!); Crash?$	
$\Longrightarrow$	$<outmed!> \{crash\}\boldsymbol{tt} \wedge \ <err!> \{crash\}\boldsymbol{tt} \wedge \ <\tau> \{crash\}\boldsymbol{tt}$	(4), (**Allocated**$<\alpha>$)
$\Longrightarrow$	$<outmed!, err!, \tau> \boldsymbol{tt}$	(**All**)

$\underline{M_1 \Longrightarrow [inmed?]\boldsymbol{ff}}$

(1)	$Outmed!\_Err!\_Crash!$	
$=$	$outmed!;\, Unstable + err!;\, Unstable + crash!;\, \boldsymbol{nil}$	
$\Longrightarrow$	$[inmed?]\boldsymbol{ff}$	$(\mathbf{Sum}[\alpha])$
(2)	$Crash?$	
$=$	$crash?;\, Err!$	
$\Longrightarrow$	$[inmed?]\boldsymbol{ff}$	$(\mathbf{Sum}[\alpha])$
(3)	$\boldsymbol{forked}(Outmed!\_Err!\_Crash!);\, Crash?$	
$\Longrightarrow$	$[inmed?]$	(1,2), $(\mathbf{Seq}_{\Phi}[c])$
	$(\boldsymbol{forked}(\boldsymbol{ff});\, Crash? \vee \boldsymbol{forked}(Outmed!\_Err!\_Crash!);\, \boldsymbol{ff})$	
$\Longrightarrow$	$[inmed?]\boldsymbol{ff}$	$(\mathbf{Strict})$
(4)	M_1	
$=$	$\{crash\}\boldsymbol{forked}(Outmed!\_Err!\_Crash!);\, Crash?$	
$\Longrightarrow$	$[inmed?]\{crash\}\boldsymbol{ff}$	(3), $(\mathbf{Allocated}[\alpha]_1)$
$\Longrightarrow$	$[inmed?]\boldsymbol{ff}$	$(\mathbf{Strict})$

$\boxed{M_1 \Longrightarrow [\tau]M_2}$

(1)	$Outmed!\_Err!\_Crash!$	
$=$	$outmed!; Unstable + err!; Unstable + crash!; \boldsymbol{nil}$	
$\Longrightarrow$	$[crash!]\boldsymbol{nil} \wedge [\tau]\boldsymbol{ff}$	($\mathbf{Sum}[\alpha]$)
(2)	$Crash?$	
$=$	$crash?; Err!$	
$\Longrightarrow$	$[crash?]Err! \wedge [ACT - \{crash?\}]\boldsymbol{ff}$	$\mathbf{Sum}[\alpha]$
(3)	$\boldsymbol{forked}(Outmed!\_Err!\_Crash!); Crash?$	
$\Longrightarrow$	$[\tau]((\boldsymbol{forked}(\boldsymbol{ff}); Crash?)$ $\vee$ $(\boldsymbol{forked}(Outmed!\_Err!\_Crash!); \boldsymbol{ff})$ $\vee$ $(\boldsymbol{forked}(\boldsymbol{nil}); Err!))$	(1,2), ($\mathbf{Seq}_\Phi[\tau]$)
$\Longrightarrow$	$[\tau](\boldsymbol{forked}(\boldsymbol{nil}); Err!)$	(**Strict**)
$\Longrightarrow$	$[\tau]Err!$	($\mathbf{EqFork}_3$), ($\mathbf{EqSequence}_2$)
(4)	M_1	
$=$	$\{crash\}\boldsymbol{forked}(Outmed!\_Err!\_Crash!); Crash?$	
$\Longrightarrow$	$[\tau]\{crash\}Err!$	(3), ($\mathbf{Allocated}[\alpha]_1$)
$=$	$[\tau]M_2$	

$\boxed{M_1 \Longrightarrow [outmed!, err!]M_0}$

$$
\begin{array}{lll}
(1) & \mathit{Outmed!\_Err!\_Crash!} & \\
= & outmed!;\, \mathit{Unstable} + err!;\, \mathit{Unstable} + crash!;\, \boldsymbol{nil} & \\
\Longrightarrow & [outmed!]\,\mathit{Unstable} \wedge [err!]\,\mathit{Unstable} & (\mathbf{Sum}[\alpha]) \\
\\
(2) & \mathit{Crash?} & \\
= & crash?;\, \mathit{Err!} & \\
\Longrightarrow & [outmed!, err!]\boldsymbol{ff} & (\mathbf{Sum}[\alpha])
\end{array}
$$

We now investigate the two cases $outmed!$ respectively $err!$. We treat only $outmed!$ since the $err!$-case is similar:

$$
\begin{array}{lll}
(3) & \boldsymbol{forked}(\mathit{Outmed!\_Err!\_Crash!});\, \mathit{Crash?} & \\
\Longrightarrow & [outmed!](& (1,2),\ (\mathbf{Seq}_{\Phi}[c]) \\
 & \quad (\boldsymbol{forked}(\mathit{Unstable});\, \mathit{Crash?}) & \\
 & \qquad \vee & \\
 & \quad (\boldsymbol{forked}(\mathit{Outmed!\_Err!\_Crash!});\, \boldsymbol{ff})) & \\
\Longrightarrow & [outmed!]\boldsymbol{forked}(\mathit{Unstable});\, \mathit{Crash?} & (\mathbf{Strict}) \\
\Longrightarrow & [outmed!]\boldsymbol{forked}(\mathit{Inmed?});\, \mathit{Crash?} & (\mathbf{UnfoldMax}) \\
\\
(4) & M_1 & \\
= & \{crash\}\boldsymbol{forked}(\mathit{Outmed!\_Err!\_Crash!});\, \mathit{Crash?} & \\
\Longrightarrow & [outmed!]\{crash\}\boldsymbol{forked}(\mathit{Inmed?});\, \mathit{Crash?} & (3),\ (\mathbf{Allocated}[\alpha]_1) \\
= & [outmed!]M_0 &
\end{array}
$$

Bibliography

[Abr85] J. R. Abrial. Programming as a Mathematical Exercise. In C. A. R Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 113–139. Prentice/Hall International, 1985.

[AGR92] E. Astesiano, A. Giovini, and G. Reggio. Observational Structures and their Logic. *Theoretical Computer Science, North-Holland*, 96:249–283, 1992.

[AM93] L. M. Alonso and R. P. Mari. Using State Variables for the Specification and Verification of TCSP Processes. In A. Bode et al., editor, *PARLE'93, Parallel Architectures and Languages Europe, LNCS 694*, pages 541–552, 1993.

[AMST92] G. Agha, I. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. Preliminary version submitted to CONCUR'92, 1992.

[AR87a] E. Astesiano and G. Reggio. An Outline of the SMoLCS Approach. In M. Venturini Zilli, editor, *Advanced School on Mathematical Models of Parallelism, LNCS 280*, pages 81–113, 1987.

[AR87b] E. Astesiano and G. Reggio. SMoLCS-Driven Concurrent Calculi. In H. Ehrig et al., editor, *TAPSOFT'87, LNCS 249*, pages 169–201, 1987.

[AR87c] E. Astesiano and G. Reggio. The SMoLCS Approach to the Formal Semantics of Programming Languages – A Tutorial Introduction. In *CRAI Spring International Conference: Innovative Software Factories and Ada, LNCS 275*, pages 81–116, 1987.

[AR92] E. Astesiano and G. Reggio. A Structural Approach to the Formal Modelization and Specification of Concurrent Systems. University of Genova, Italy, 1992.

[Arn] A. Arnold. Verification and Comparison of Transition Systems.

[AW91] H. R. Andersen and G. Winskel. Compositional Checking of Satisfaction. In K. G. Larsen and A. Skou, editors, *CAV'91, Computer Aided Verification, LNCS 575*, pages 24–36, 1991.

[Bar84] J. G. P. Barnes. *Programming in Ada.* International Computer Science Series, 2 edition, 1984.

[BB87] B. Banieqbal and H. Barringer. Temporal Logic with Fixed Points. In B. Banieqbal et al., editor, *Temporal Logic in Specification, LNCS 398*, pages 62–74, 1987.

[BB89] T. Bolognesi and E. Brinksma. *The Formal Description Technique LOTOS: Introduction to the ISO Specification Language LOTOS*, pages 23–73. North-Holland, 1989.

[BB90] G. Berry and G. Boudol. The Chemical Abstract Machine. In *Proceedings of the 17th ACM Symphosium on Principles of Programming Languages*, 1990.

[Ber93] B. Berthomieu. Programming with Behaviours in an ML Framework, The Syntax and Semantics of LCS. Technical Report LAAS/CNRS 93133, LAAS Toulouse, France, 1993.

[BG77] R. M. Burstall and J. A. Goguen. Putting Theories Together to Make Specifications. In *Fifth International Joint Conference on Artificial Intelligence, Carnegie-Mellon University, Pittsburgh*, pages 1045–1058, 1977.

[BGM89] M. Bidoit, M. C. Gaudel, and A. Mauboussin. How to Make Algebraic Specifications More Understandable – An Experiment with the PLUSS Specification Language. *LNCS 394*, pages 31–67, 1989.

[BHH92] D. Bjørner, A. Haxthausen, and K. Havelund. Formal, Model-oriented Software Development Methods: From VDM to ProCoS and from RAISE to LaCoS. *Future Generation Computer Systems, North-Holland*, 7 (1991/1992):111–138, 1992.

[BJ78] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language, LNCS 61.* 1978.

[BJ82] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development.* Prentice-Hall International, 1982.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may Compose Temporal Logic Specifications. In *STOC'84*, pages 51–63, 1984.

[BMT92] D. Berry, R. Milner, and D.N. Turner. A Semantics for ML Concurrency Primitives. In *Proceedings of the 19th ACM Symphosium on Principles of Programming Languages*, 1992.

[Bro83] S. D. Brookes. On the Relationship of CCS and CSP. In J. Diaz, editor, *10th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 154*, pages 83–96, 1983.

[BS90] J. Bradfield and C. Stirling. Verifying Temporal Properties of Processes. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR'90, LNCS 458*, pages 115–125, 1990.

[Cha87] Z. Chaochen. Specifying Communicating Systems with Temporal Logic. In B. Banieqbal et al., editor, *Temporal Logic in Specification, LNCS 398*, pages 304–323, 1987.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CR91] G. Costa and G. Reggio. Abstract Dynamic Data Types: A Temporal Logic Approach. Preliminary version of paper appearing in proceedings of MFCS'91, LNCS, 1991.

[Dah87] O-J. Dahl. Object-Oriented Specification. *Research Directions in Object-Oriented Programming, Computer Systems Series*, pages 561–576, 1987.

[Dah90] O-J. Dahl. Object Orientation and Formal Techniques. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90, VDM and Z – Formal Methods in Software Development, LNCS 428*, pages 1–11, 1990.

[Eme90] E. A. Emerson. *Handbook of Theoretical Computer Science: Temporal and Modal Logic*, chapter 16, pages 996–1072. Elsevier Science Publishers, 1990.

[FGR92] A. Fantechi, S. Gnesi, and G. Ristori. From ACTL to μ-calculus – Extended Abstract. ERCIM Workshop on Theory and Practice in Verification, Pisa, december 1992.

[GHNW88] C. George, K. Havelund, M. Nielsen, and K. Wagner. The RAISE Language, Method and Tools. *LNCS, Springer-Verlag*, 328, 1988.

[GHNW89] C. George, K. Havelund, M. Nielsen, and K. Wagner. The RAISE Language, Method and Tools. *Formal Aspects of Computing, Springer International*, 1(1), January-March 1989. Same paper as [GHNW88].

[GLZ] J. C. Godskesen, K. G. Larsen, and M. Zeeberg. TAV (Tools for Automatic Verification) Users Manual. Aalborg University Center, Denmark.

[GM87] J. A. Goguen and J. Mesequer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. *Research Directions in Object-Oriented Programming, Computer Systems Series*, pages 417–477, 1987.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. FACILE: A Symmetric Integration of Concurrent and Functional Programming. In *TAPSOFT'89, LNCS 352*, pages 184–209, 1989.

[Gog90] J. Goguen. An Algebraic Approach to Refinement. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90, VDM and Z – Formal Methods in Software Development, LNCS 428*, pages 12–28, 1990.

[GS86] S. Graf and J. Sifakis. A Logic for the Description of Nondeterministic Programs and their Properties. *Information and Control*, 68(1-3):254–270, 1986.

[GS87] S. Graf and J. Sifakis. An Expressive Logic for a Process Algebra with Silent Actions. In B. Banieqbal et al., editor, *Temporal Logic in Specification, LNCS 398*, pages 44–61, 1987.

[GV89] J. F. Groote and F. W. Vaandrager. Structured Operational Semantics and Bisimulation as a Congruence. *LNCS 372*, 1989.

[Hav92] K. Havelund. *RSL Tutorial*, pages 9–250. The BCS Practitioner Series. Prentice Hall, 1992. The written material is part I of the book named 'The RAISE Specification Language'. Part II of the book (pages 251–369) is the 'RSL Reference Description' and is written by A. Haxthausen and K. Havelund in collaboration.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Boston, 1988.

[HL93a] K. Havelund and K. G. Larsen. The Fork Calculus. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *20th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 700*, pages 544–557, 1993.

[HL93b] K. Havelund and K. G. Larsen. The Fork Calculus. In S. Meldal and M. Haveraaen, editors, *4th Nordic Workshop on Program Correctness, Report no 78, University of Bergen*, pages 153–164. Department of Informatics, University of Bergen, Norway, 1993.

[HL93c] M. Hennessy and H. Lin. Proof Systems for Message-Passing Process Algebras. In *CONCUR'93, LNCS 715*, pages 202–216, 1993.

[HL93d] M. Hennessy and X. Liu. A Modal Logic for Message Passing Processes. Technical Report 3/93, University of Sussex, 1993.

[HM89] K. Havelund and R. Milne. The Semantics of RSL. Technical Report RAISE/DDC/KH/43, Computer Resources International (CRI), 1989.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM*, 32(1):137–161, January 85.

[Hoa85a] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[Hoa85b] C. A. R. Hoare. Programs are Predicates. In C. A. R Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 141–155. Prentice/Hall International, 1985.

[Hol89] S. Holmström. A Refinement Calculus for Specifications in Hennessy-Milner Logic with Recursion. *Formal Aspects of Computing*, 1:242–272, 1989.

[Hug89] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989.

[IT91] A. Ingólfsdóttir and B. Thomsen. Semantic Models for CCS with Values. In *Chalmars Workshop on Concurrency*, 1991. Preliminary version of the proceedings paper.

[Jen92] F. Jensen. The C-Lisp Language. Unpublished paper, Aalborg University Center, Denmark, 1992.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall International, 1980. A newer book exists: 'Systematic Software Development – Using VDM' (1989), also Prentice-Hall, but I have not read it like the first.

[Kap89] S. Kaplan. Algebraic Specification of Concurrent Systems. *Theoretical Computer Science, North-Holland*, 69:69–115, 1989.

[Kaz91] E. Kazmierczak. Modularizing the Specification of a Small Database System in Extended ML. LFCS Report Series ECS-LFCS-91-177, Department of Computer Science, University of Edinburgh, September 1991.

[KD91] S. Kaplan and G. Deutsch. Algebraic Semantics of Real-Time Process Specifications. Preliminary version of AMAST'91, 1991.

[KKN+90] D. Kato, T. Kikuchi, R. Nakajima, J. Sawada, and H. Tsuiki. Modal Logic Programming. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90, VDM and Z – Formal Methods in Software Development, LNCS 428*, pages 29–40, 1990.

[Kle93] S. Kleuker. Case Study: Stepwise Development of a Communication Processor using Trace Logic. 1993.

[Koy87] R. Koymans. Specifying Message Passing Systems Requires Extending Temporal Logic. In B. Banieqbal et al., editor, *Temporal Logic in Specification, LNCS 398*, pages 213–223, 1987.

[Koz82] D. Kozen. Results on the Propositional mu-calculus. In *9th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 140*, 1982.

[KP87] S. Kaplan and A. Pnueli. Specification and Implementation of Concurrently Accessed Data Structures: An Abstract Data Type Approach. In F.J. Brandenburg et al., editor, *4th Annual Symposium on Theoretical Aspects of Computer Science (STACS'87), LNCS 247*, pages 220–244, 1987.

[KST93] S. Kahrs, D. Sannella, and A. Tarlecki. The Semantics of Extended ML: A Gentle Introduction. Submitted for Publication, 1993.

[KT90] D. Kozen and J. Tiuryn. *Handbook of Theoretical Computer Science: Logics of Programs*, chapter 14, pages 791–840. Elsevier Science Publishers, 1990.

[Lar87] K. G. Larsen. From Modal Logic to Process Algebra. Aalborg University Center, Denmark, 1987.

[Lar90] K. G. Larsen. Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *Theoretical Computer Science*, 72:265–288, 1990.

[LO81] L. Lamport and S. Owicki. Program Logics and Program Verification. *Logics of Programs, LNCS 131*, pages 197–199, 1981.

[LT88] K. G. Larsen and B. Thomsen. A Modal Process Logic. Technical Report R 88-5, Aalborg University Center, Denmark, January 1988.

[LT92a] L. Leth and B. Thomsen. A Proposal for a Facile Specification Logic. ECRC Magic note no. 31, European Computer-Industry Research Centre, Munich, Germany, 1992.

[LT92b] L. Leth and B. Thomsen. Some Facile Chemestry. Technical Report ECRC-92-14, ECRC – European Computer-Industry Research Centre, Munich, Germany, 1992.

[Mat91] D. Matthews. A Distributed Implementation of Standard ML. LFCS Report Series ECS-LFCS-91-174, Department of Computer Science, University of Edinburgh, August 1991.

[Mil81] R. Milner. A Modal Characterisation of Observable Machine-Behaviour. In *CAAP'81, LNCS 112*, pages 25–34, 1981.

[Mil85] R. Milner. The use of Machines to Assist in Rigorous Proof. In C. A. R Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 77–88. Prentice/Hall International, 1985.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[Mis89] J. Misra. A Foundation of Parallel Programming. *NATO ASI Series: Constructive Methods in Computing Science*, F55:397–443, 1989.

[MM93] P. D. Mosses and M. Musicante. An Action Semantics for ML Concurrency Primitives. Submitted for publication, 1993.

[Mol90] F. Moller. The Importance of the Left Merge Operator in Process Algebra. *LNCS 443*, 1990.

[MP89] Z. Manna and A. Pnueli. Completing the Temporal Picture. In G. Ausiello et al., editor, *16th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 372*, pages 534–558, 1989.

[MPW89a] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, part I. Technical report, Department of Computer Science, University of Edinburgh, 1989.

[MPW89b] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, part II. Technical report, Department of Computer Science, University of Edinburgh, 1989.

[MPW91] R. Milner, J. Parrow, and D. Walker. Modal Logics for Mobile Processes. Technical Report R91:03, Swedish Institute of Computer Science (SICS), 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[NFGR92] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An Action-based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems. Appeared in: Computer Networks and ISDN Systems, 1992.

[Nie89] F. Nielson. The Typed Lambda-Calculus with First-Class Processes. In *PARLE'89, Parallel Architectures and Languages Europe, LNCS 366*, 1989.

[NN93] F. Nielson and H. R. Nielson. From CML to Process Algebras. In *CONCUR'93, LNCS 715*, 1993.

[NT92a] U. Nestmann and L. Teleki. A Chemical Abstract Machine for a Calculus of Communicating Functions. Technical Report IMMD7-14/92, Erlangen University, Germany, 1992.

[NT92b] U. Nestmann and L. Teleki. Towards a Calculus of Communicating Functions. Technical Report IMMD7-13/92, Erlangen University, Germany, 1992.

[NV90] R. De Nicola and F. Vaandrager. Action Versus State Based Logics for Transition Systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, LNCS 469*, pages 407–419, 1990.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of 5th GI Conference, LNCS 104*, 1981.

[PGM90] S. Prasad, A. Giacalone, and P. Mishra. Operational and Algebraic Semantics for Facile. In *17th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 443*, pages 765–780, 1990.

[Plo81] G. Plotkin. A Structural Approach to Operational Semantics. FN 19, DAIMI, Aarhus University, Denmark, 1981.

[PN87] H. Partsch and K. Nijmegen. Algebraic Specification – A Step towards Future Software Engineering. *Algebraic Methods: Theory, Tools and Applications, LNCS 394*, pages 7–30, 1987.

[Pra91] S. Prasad. *Towards a Symmetric Integration of Concurrent and Functional Programming.* PhD thesis, State University of New York, at Stony Brook, december 1991.

[Rep88] J. H. Reppy. Synchronous Operations as First-Class Values. In *ACM Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, June 22-24*, pages 250–259, 1988.

[Rep91a] J. H. Reppy. CML: A Higher-order Concurrent Language. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (SIGPLAN Notices 26(6))*, pages 293–305, 1991.

[Rep91b] J.H. Reppy. Concurrent Programming with Events – The Concurrent ML Manual. Technical report, Cornell University, Department of Computer Science, 1991.

[Rep91c] J.H. Reppy. An Operational Semantics of First–class Synchronous Operations. Technical Report TR 91–1232, Cornell University, Department of Computer Science, 1991.

[Rep92] J. H. Reppy. *High-Order Concurrency.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, June 1992.

[San89] D. Sannella. Formal Program Development in Extended ML for the Working Programmer. LFCS Report Series ECS-LFCS-89-102, Department of Computer Science, University of Edinburgh, December 1989.

[San93] D. Sangiorgi. A Theory of Bisimulation for the π-calculus. 1993.

[SdST90] D. Sannella, F. da Silva, and A. Tarlecki. Syntax, Static Semantics and Dynamic Semantics of Extended ML. 1990.

[SFSE87] A. Sernadas, J. Fiadeiro, C. Sernadas, and H. D. Ehrich. Abstract Object Types: A Temporal Perspective. In B. Banieqbal et al., editor, *Temporal Logic in Specification, LNCS 398*, pages 324–350, 1987.

[SGM89] T. Stroup, N. Götz, and M Mendler. Stepwise Refinement of Layered Protocols by Formal Program Development. Technical Report 2/89, Erlangen University, Germany, 1989.

[Spi89] J. M. Spivey. *Understanding Z.* Cambridge University Press, 1989. I have been reading a preliminary pre-runner of the book, before the book was printed, basically a copy of Spivey's thesis.

[ST90] D. Sannella and A. Tarlecki. Extended ML: Past, Present and Future. In H. Ehrig et al., editor, *Recent Trends in Data Type Specification, 7th Workshop on Specification of Abstract Data Types, LNCS 534*, pages 297–322, 1990.

[Sti85a] C. Stirling. A Complete Compositional Modal Proof System for a Subset of CCS. *12th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 194*, 1985.

[Sti85b] C. Stirling. A Complete Modal Proof System for a Subset of SCCS. In H. Ehrig et al., editor, *Mathematical Foundations of Software Development, LNCS 185*, pages 253–266, 1985.

[Sti85c] C. Stirling. A Proof Theoretic Characterisation of Observational Equivalence. *Theoretical Computer Science, North-Holland*, 39, 1985.

[Sti87] C. Stirling. Comparing Linear and Branching Time Temporal Logics. In B. Banieqbal et al., editor, *Temporal Logic in Specification, LNCS 398*, pages 1–20, 1987.

[Sti88] C. Stirling. Temporal Logics for CCS. In J. W. de Bakker et al., editor, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS 354*, pages 660–672, 1988.

[Tar55] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math.*, 5, 1955.

[Tho89] B. Thomsen. A Calculus of Higher Order Communicating Systems. In *Proceedings of the 16th ACM Symphosium on Principles of Programming Languages*, pages 143–154, 1989.

[Tho90] B. Thomsen. *Calculi for Higher-Order Communicating Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, Department of Computing, London, 1990.

[Tho92] B. Thomsen. Compositional Reasoning about Facile Programs. ECRC Magic note no. 21, European Computer-Industry Research Centre, Munich, Germany, 1992.

[Tur85] D. A. Turner. Functional Programs as Executable Specifications. In C. A. R Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54. Prentice/Hall International, 1985.

[Win84] G. Winskel. On the Composition and Decomposition of Assertions. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency, CMU-Pittsburgh, LNCS 197*, pages 62–75, 1984.

[Win86] G. Winskel. A Complete Proof System for SCCS with Modal Assertions. *Fundamenta Informaticae, North-Holland*, IX:401–419, 1986.

[Win89] G. Winskel. A Note on Model Checking the Modal ν-calculus. In *16th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 372*, pages 761–772, 1989.

[Wol83] P. Wolper. Temporal Logic can be More Expressive. *Information and Control*, 56:72–99, 1983.