A Symbolic Approach to Value-Passing Processes

Anna Ingólfsdóttir
Institute of Computer Science
Aalborg University

Huimin Lin
Laboratory for Computer Science
Institute of Software, Chinese Academy of Sciences

Abstract

In this chapter we give a survey of the semantic theory for value-passing processes, focusing on bisimulation equivalences. The emphasis is on the symbolic method. Both operational and proof theoretic approaches are described and their expressiveness compared.

keywords: Value-passing processes, operational semantics, late/early/strong/weak/symbolic bisimulation, axiomatization, proof system.

Contents

1 Introduction

In the pioneering work of Milner [Mil80] on *CCS* and Hoare [Hoa78] on *CSP*, exchange of data appeared as a natural feature of these languages. Subsequently, although some work was done on their semantics, see e.g. [MM79, HP80, FLP84], most of the research effort was devoted to the pure versions of these languages such as *pure CCS* [Mil80, Mil89] and *theoretical CSP*, *TCSP* [BHR84]. The focus was on the behavioural properties of processes where no data are involved and the only form of communication considered was pure synchronization. In particular, structural operational semantics by means of inference rules [Plo81] and more abstract behavioral equivalences based on these were defined and their properties studied in detail.

About a decade after the introduction of *CCS* and *CSP*, the theory of pure processes was considered as being well established and research on semantic theories for value-passing processes was back in focus [Hen91, HI93, HI93b, HL95, HL96]. Not so surprisingly, it turned out that allowing transmission of elements from a possibly infinite value domain introduced some technical complications, with the consequences that some of the already established theories for the pure calculi could not be applied directly.

In this chapter we will try to give the reader some insight into the theoretical complications that arise when such value-passing processes are described semantically, as well as solutions to those. Our main focus will be on techniques for checking semantic equivalences between processes. In particular, we are interested in investigating equivalences based on the idea of *bisimulation* due to Milner and Park [Mil83, Par81]. For this purpose *symbolic operational semantics* and the corresponding notion of *symbolic bisimulation* are introduced. These are abstractions of the standard semantics which allow us to perform case analysis on the input values and thus, in many cases, to resolve the infinity arising from an infinite value space.

We start the chapter by providing the readers with some motivation, mainly inspired by examples. Based on the acquired intuition we give an informal introduction to our example language, which is basically the original *CCS* as it was introduced by Milner in [Mil80].

1.1 Some Examples

In standard sequential programming, manipulating and exchanging values with the environment occurs naturally, as illustrated by the following example.

Example 1.1 The program `Prog` below is a simple sequential program that reads two numbers, calculates their sum and multiple, divides the sum by the multiple and outputs the result:

$$
\begin{array}{l}
\texttt{Prog}: \\
\texttt{begin} \\
\quad \texttt{input}\,(x, y); \\
\quad t := x + y; u := x \cdot y; \\
\quad \texttt{output}\,(t/u) \\
\texttt{end}.
\end{array}
$$

This program interacts or communicates with its environment, i.e. the user, by receiving the input values x and y from him and giving him back the result of the computation. $\square$

In concurrent computations, value-passing occurs just as naturally and processes cooperate with each other by exchanging messages, typically through communication channels or ports. A synchronous communication event, which is the type of communication we consider in this chapter, usually consists of two complementary actions: input and output. A process *rec*, which is ready to receive or input a message from a channel c, may be given by $rec = c?x.rec'$ where typically x occurs free in the sequel rec'; when inputting a value v from channel c this process evolves into $rec'[v/x]$, i.e. the process rec' with all free occurrences of the variable x replaced by the value v. Similarly a process *send* that prepares to send or output a value v on the same channel and by sending evolves into its sequel $send'$, may be expressed as $send = c!v.send'$. When these two processes are running in parallel, denoted by $rec \mid send$, a communication event can take place, with the consequence that the composite system evolves into a new state $rec'[v/x] \mid send'$ where the "passing" of the value v from *send* to *rec* is modeled by substitution, as explained before. Furthermore, once these processes have agreed to communicate, the communication is private and no other processes can take part in it; this kind of communication is called *handshaking*. More formally the event described above may be given by a transition relation induced by

$$rec \xrightarrow{c?v} rec'[v/x] \text{ and } send \xrightarrow{c!v} send' \text{ imply } rec \mid send \xrightarrow{\tau} rec'[v/x] \mid send'$$

Here τ is a special invisible action indicating that synchronization and exchange of values between processes is not observable and therefore it can not take part in a further synchronization with the environment.

Example 1.2 In the above example, noting that the commands $t := x+y$ and $u := x \cdot y$ may be performed independently, the program `Prog` may be rewritten in the process description language *CCS* as a concurrent process by

$$p = (in_1?x.in_2?y.(c!(x+y) \mid d!(x \cdot y) \mid d?u.c?t.out!(t/u)))\backslash\{c, d\}$$

where the restriction operator $\_\backslash\{c, d\}$ indicates that p can use c and d only for internal communication, but that it is not allowed to communicate with its environment via these channels. We note that the process calculus *CCS* is applicative in the sense that it does not allow for assignments. Thus the results of the calculations of $x + y$ and $x \cdot y$ are output directly without assigning them to a variable first. $\square$

The next example shows how processes with infinite behaviour may be given using recursive definitions parameterized over data.

Example 1.3 A *counter* is a process that outputs the natural numbers, one after the other in increasing order on a given channel, say c. To express this we let

$$C(n) \Leftarrow c!n.C(n+1)$$

meaning that the parameterized recursive process $C(n)$ behaves like the right hand side of the defining equality, i.e. it outputs the number n on channel c and afterwards behaves like $C(n{+}1)$. It is not difficult to see that $C(0)$ has the expected behaviour of a counter as described above. To obtain two distinguishable copies of the counter, instead of giving two copies of similar recursive definitions, we apply a so-called *renaming operator* on $C(n)$. Thus $C(n)[c \mapsto c']$, the process $C(n)$ with c renamed to c', behaves like $C(n)$ with all occurrences of c replaced by c', i.e. like the counter that makes use of the channel c' instead of c. □

The following example shows how parameterized recursive processes over an inductively defined data domain may be defined inductively.

Example 1.4 In what follows let $Stack(b_1 b_2 \ldots b_n)$ denote a stack over an alphabet L. We represent the content of the stack as a string $b_1 b_2 \ldots b_n \in L^*$ where b_1 is the top element of the stack. Let ε denote the empty string, $b \in L$ and $s \in L^*$. Furthermore, let $push?x$ denote the action of inputting an element from channel $push$ and assigning it to x, and $pop!b$ the action of outputting the element b on channel pop. Then we may model $Stack(s), s \in L^*$, inductively as follows:

$$\begin{aligned} Stack(\varepsilon) &\Leftarrow push?x.Stack(x) \\ Stack(bs) &\Leftarrow pop!b.Stack(s) + push?x.Stack(xbs) \end{aligned}$$

Here the operator $+$ (*choice*) indicates that the the process $Stack(bs)$ nondeterministically chooses between behaving like one of its components, i.e. either like $pop!b.Stack(s)$ or like $push?x.Stack(xbs)$. The intuitive behaviour of the process $Stack$ is that if it is empty ($Stack(\varepsilon)$) its only possible action is to push onto it an element received from the environment via the channel $push$. If the stack is not empty, i.e. it contains a string of the form bs ($Stack(bs)$) with b as its top element, then either the element b can be popped off the stack and returned to the environment via pop, or the environment can push a new element x onto the stack via $push$ whereby it evolves to the stack $Stack(xbs)$. □

1.2 Late and Early Semantics

A standard way of describing each computational step of a concurrent or reactive system is by assigning to it a so-called *Structural Operational Semantics* as described in Chapters 1.1 and 1.3 in this issue; the semantics we have introduced informally in this section is based on this approach. However it is a widespread opinion amongst researchers in the area that this kind of semantic description is in general too concrete to reason about the behaviour of processes on a more abstract level, for instance to decide when two processes are behaviourally or semantically equivalent. In [Mil83] Milner overcomes this inadequacy by describing a general idea to model observational equivalence of processes referred to as *black box testing* (see Chapter 1.1 in this issue for further motivation); a process is considered to be a black box provided with an interface containing buttons, one for each (observable) action it can perform, plus possibly some other control devices such as lights or buttons. An external observer can interact with, and thus perform experiments on, the process by pressing these buttons and observing its reactions. Two processes are said to be behaviourally equivalent if the observer is not able to distinguish between their reactions to his experiments. A variety of preorders and equivalences are defined in the literature based on the black box model, but with different notions of observers, observable actions and experiments. A standard class of equivalences, and probably the most studied one, consists of Milner's equivalences based on the idea of *bisimulation*, which may briefly be described as follows (see also Chapter 1.1 in this issue for a more detailed explanation):

> Two processes p and q are bisimilar if whenever one of them, say p, reacts when the button labelled by a is pressed by evolving to some state p' (i.e. $p \xrightarrow{a} p'$) then q also reacts when some button, labelled by a, is pressed and by doing so it evolves to some state q' where p' and q' are again bisimilar.

Originally the idea of bisimulation only concerned pure processes. Extending the idea to value–passing processes raises some questions about what we mean by an (observable) action and consequently how processes are supposed to synchronize. We will try to explain this by the following example.

Example 1.5 Let c, c_1 and c_2 be three different channel names and the processes p, p_1, p_2 and q, q_1, q_2 be defined by

$$\begin{aligned}
p_1 &= c?x.\mathtt{if}\ x \leq 5\ \mathtt{then}\, c_1!x\ \mathtt{else}\, c_2!x \\
p_2 &= c?x.\mathtt{if}\ x \leq 5\ \mathtt{then}\, c_2!x\ \mathtt{else}\, c_1!x \\
q_1 &= c?x.c_1!x \\
q_2 &= c?x.c_2!x \\
p &= p_1 + p_2 \\
q &= q_1 + q_2
\end{aligned}$$

Obviously p and q are bisimilar according to Milner's original semantics as they have isomorphic transition graphs. However we also note that different moves of p due to the same component, say p_1, have to be matched by q by moves coming from different components. Thus

$$p \xrightarrow{c?v} \mathtt{if}\ v \leq 5\ \mathtt{then}\, c_1!v\ \mathtt{else}\, c_2!v \text{ for } v = 1 \text{ and } v = 6$$

because

$$p_1 \xrightarrow{c?v} \mathtt{if}\ v \leq 5\ \mathtt{then}\, c_1!v\ \mathtt{else}\, c_2!v \text{ for } v = 1 \text{ and } v = 6$$

On the other hand, to match these moves by q we need one move from each of its components q_1 and q_2, i.e.

$$q \xrightarrow{c?1} c_1!1 \text{ because } q_1 \xrightarrow{c?1} c_1!1$$

and

$$q \xrightarrow{c?6} c_2!6 \text{ because } q_2 \xrightarrow{c?6} c_2!6$$

□

In terms of black box testing the semantic approach described in the examples above corresponds to the situation in which the buttons for the input actions are marked with $c?v$, i.e. with labels consisting of both $c?$, to indicate that the value should be received along the channel c, and also the *concrete value* v that is to be received along that channel. This semantic approach has been referred to as *early semantic*s due to the early instantiation of the value variable that is involved in the communication.

Another approach would be to consider input actions of the form $c?$ for the action of inputting *some value* on the channel c. In the black box interpretation this means that we only mark the corresponding buttons for input actions with labels of the form $c?$ and leave out the concrete value v. This entails that the result of pressing such a button or performing an action of the form $c?$ is an incomplete process as the information about the concrete value the process is about to receive from the environment is still missing. In this introduction we consider this incomplete state as a function that inputs a value and delivers a completed process. (In the main body of this chapter these incomplete states are given as process terms with possibly free value variables.) Such functions are referred to as *abstractions* in [Mil92]. Syntactically abstractions are denoted by $(x)p$ where x is a value variable and p is a process term. They can be interpreted as a shorthand for the λ-abstraction $\lambda v.p[v/x]$. Two such abstractions match each other semantically if they do so pointwise, i.e. if whenever they are both applied to the same value, the results are indistinguishable. This semantic approach has been termed *late* semantics due to the delayed instantiation of the value variables.

Example 1.6 To show that p and q from the previous example are not strongly bisimilar according to the late approach, we proceed as follows. The process p has the late transition

$$p \xrightarrow{c?} \lambda v.\mathtt{if}\ v \leq 5\ \mathtt{then}\ c_1!v\ \mathtt{else}\ c_2!v = f$$

The possible $c?$ transitions for q are

$$q \xrightarrow{c?} \lambda v.c_1!v = f_1 \quad \text{and} \quad q \xrightarrow{c?} \lambda v.c_2!v = f_2$$

For p and q to be bisimilar in the late sense it is necessary that either f and f_1 match completely pointwise or otherwise f and f_2 must do. So either we have that for all v, $f(v)$ and $f_1(v)$ are bisimilar or that for all v, $f(v)$ and $f_2(v)$ are. To see that this is not the case we observe that

$$f(6) = \mathtt{if}\ 6 \leq 5\ \mathtt{then}\ c_1!6\ \mathtt{else}\ c_2!6 \xrightarrow{c_2!6} \quad \text{while} \quad f_1(6) \not\xrightarrow{c_2!6}$$

and that

$$f(1) \xrightarrow{c_1!1} \quad \text{while} \quad f_2(1) \not\xrightarrow{c_1!1}$$

where $p \xrightarrow{a}$ means that $p \xrightarrow{a} p'$ for some p' and $p \not\xrightarrow{a}$ means that no such p' exists. □

The two semantic approaches described above lead to two different notions of bisimulation: the *early* bisimulation and the *late* one, which is strictly finer. However we want to point out here that if we look at the rules for communication, it turns out that the early and the late semantics yield exactly the same result.

Example 1.7 In the example above, if $r = c!v$ then the late semantics yields

$$q \mid r \xrightarrow{\tau} f_1(v) \mid \mathbf{0} = c_1!v \mid \mathbf{0} \text{ and } q \mid r \xrightarrow{\tau} f_2(v) \mid \mathbf{0} = c_2!v \mid \mathbf{0}$$

It is not difficult to see that the early semantics gives exactly the same transitions. $\square$

In more general terms only the intermediate states which the processes reach immediately after an input action can be distinguished by the late and early semantics but after a completed communication they are syntactically the same for these two approaches. This implies, as shown in [Ing95], that although behavioural equivalences that involve investigation of these intermediate states are different for the late and early semantics, the equivalences that do not involve observations of states that occur as a result of input actions cannot be told apart. An example of a semantics, where the late and early versions coincide is the testing based semantics due to De Nicola and Hennessy.

1.3 Infinite Branching

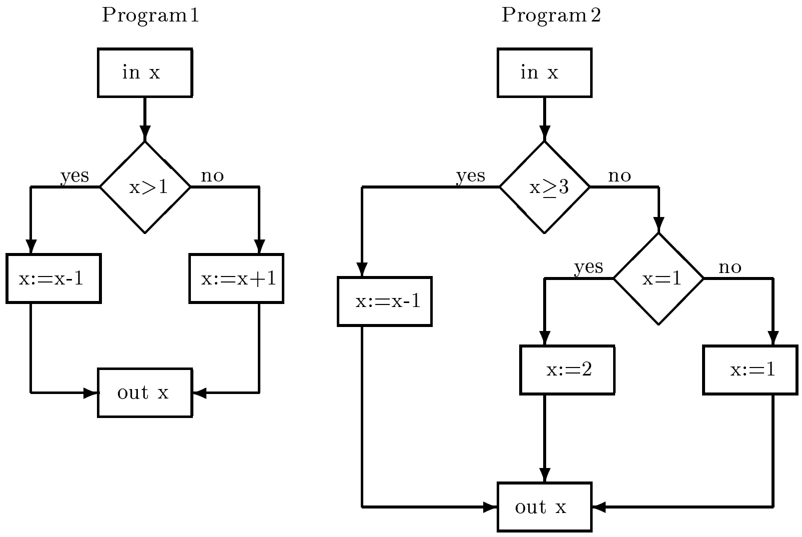
As pointed out at the beginning of this section, the main technical difficulties associated with introducing value-passing into the semantic theory of pure processes is due to the infinite branching of the input transitions. For example, the process $c?x.p$ has the transitions $c?x.p \xrightarrow{c?v} p[v/x]$ where v ranges over the value domain which in general may be assumed to be infinite. However, in *CCS* the only control structure that allows branching on input values is the construction `if _ then _ else _` and obviously each process can only contain finitely many of those. Therefore, by applying a case analysis on the input values, in many cases this infinite branching can be resolved and reduced to finite branching. To explain this point let us first have a look at the following example from sequential programming.

Example 1.8 Let us consider the following two sequential programs.

$Program1$;
`input` x;
`if` $x > 1$ `then` $x := x - 1$
`else` $x := x - 2$;
`output` x.

$Program2$;
`input` x;
`if` $x \geq 3$ `then` $x := x - 1$
`else` (`if` $x = 1$ `then` $x := 2$
`else` $x := 1$);
`output` x.

The actions of these two programs are given by the data flow diagrams below.

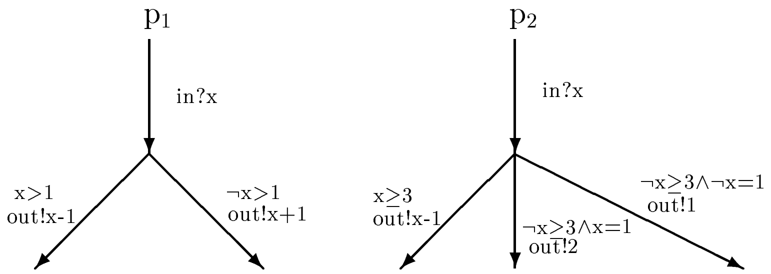


By case analysis on the input value for the variable x, considering the cases $0 \leq x < 1$, $x = 1$, $1 < x < 3$ and $x \geq 3$ the reader should have no difficulties in convincing himself that these two programs are equivalent in the sense that for a given input x they will both return exactly the same output.

As pointed out before, *CCS* does not allow for assignments. However the action of these two programs may be simulated by the following *CCS* processes.

$$p_1 = in?x.\mathtt{if}\, x > 1\, \mathtt{then}\, out!(x-1)\, \mathtt{else}\, out!(x+1)$$
$$p_2 = in?x.\mathtt{if}\, x \geq 3\, \mathtt{then}\, out!(x-1)\, \mathtt{else}\, (\mathtt{if}\, x = 1\, \mathtt{then}\, out!2\, \mathtt{else}\, out!1)$$

Based on the same idea as the data flow diagrams, an abstract or *symbolic* transition graph for these processes may be given as below. We note that these graphs have finite branching.



Again, using the same kind of case analysis, based on the same cases as before, the reader should be able to check that these two processes are bisimilar both in the early and the late sense. □

One of the main purposes of this chapter is to describe formally a *symbolic semantics* that realizes the ideas described above and thus provides a theoretical framework that allows us to formally reason about equivalences between value-passing processes by only investigating finitely branching transition graphs.

1.4 Axiomatization

Providing sound and complete axiomatizations for various equivalence relations has been one of the major research topics in the development of process theories. A complete axiomatization not only allows us to reason about process behaviours by syntactic manipulation, but also helps to understand the properties of the operators used to build complex processes from simpler components.

Traditionally, axiomatizations in process calculi are based on pure equational reasoning, namely replacing equals by equals. For value-passing processes, however, this approach is in general not adequate, because, for instance, input prefixing introduces the notion of bound variables which is beyond the realm of pure equational reasoning. To be able to deal with such "non-algebraic" constructs we generalize equational reasoning by introducing inference rules while preserving the equational laws that are at the core of the axiomatizations in pure calculi.[1]

As data play an important role in value-passing processes, it seems that any axiomatization for such processes would have to involve axiomatizations for the data domain. However, it turns out that we can factor out data reasoning from process reasoning, by employing *conditional equations* of the form

$$b \rhd t = u$$

where t and u are process terms and b is a boolean expression representing the condition on the data domain under which t and u are equal. An example of a proof rule of our inference system is

$$\text{COND} \quad \frac{b' \wedge b \rhd t = u \qquad b' \wedge \neg b \rhd \mathbf{0} = u}{b' \rhd \texttt{if } b \texttt{ then } t = u}$$

It captures the intuitive meaning of the (one-armed) conditional construct: `if` b `then` t behaves like t when b is true, and like $\mathbf{0}$ (the "empty process) otherwise. In fact this rule is all we need to know about the `if _ then _` construct when manipulating syntactical terms. Looked upon from a "goal-directed" point of view, it moves the parts involving data (b) from the process term (`if` b `then` t) to the conditional part of the judgment. Such conditions can be used to discharge constructs involving data when some other inference rules are applied. For instance, the following rule for output prefixing:

$$\text{OUTPUT} \quad \frac{b \models e = e', \qquad b \rhd t = u}{b \rhd c!e.t = c!e'.u}$$

allows us to conclude that $c!e.t$ and $c!e'.u$ are equal under b, provided that t and u are equal under b and that $e = e'$ is implied by b. As the last condition only relies on reasoning over data, it is factored out from the main proof system. This is manifested by using the notation $b \models e = e'$.

Using inference rules in this style, together with the usual equational axioms, a spectrum of proof systems for value-passing processes can be obtained which are sound and complete (relative to data domain reasoning) with respect to the various bisimulation-based equivalences.

1.5 Outline of the Chapter

The rest of this chapter is organized as follows. In Section 2 we formally introduce the syntax and operational semantics for our example language, full *CCS*, whereas Section 3 is devoted to defining the various bisimulation based equivalences we investigate.

[1]For a treatment of these matters that stays closer to equational logic see [GL98].

$$
\begin{array}{rcl}
t & ::= & \mathbf{0} \mid \alpha.t \mid t+t \mid \texttt{if}\ b\ \texttt{then}\ t \mid t \mid t \mid t\backslash L \mid t[f] \mid P(\overline{e}) \\
\alpha & ::= & \tau \mid c?x \mid c!e
\end{array}
$$

Figure 1: Syntax

Symbolic operational semantics and symbolic bisimulations are studied in Section 4. In Section 5 we formulate proof systems to reason about value-passing processes, one for each of the congruences mentioned above, and we give some soundness and completeness results. The main content of this chapter is summarized in Section 6 where pointers to relevant works are also given.

We will omit detailed proofs as much as possible in this chapter by referring to the literature, and will only supply them when they are not available in published papers.

2 The Value-Passing Language

In this section we introduce the syntax for the language we consider in this chapter together with its operational semantics. The language is essentially the same as the one originally proposed by Milner in [Mil80, Section 2.8], often referred to as *full CCS* or *value-passing CCS*.

2.1 The Syntax

The language we consider is rather abstract as we want to focus on aspects which do not depend on the properties of the data domain. Therefore we simply assume a predefined syntactic category *DExp* of data expressions, ranged over by e, e', etc. This should include, at least, an infinite set *Val* of values, ranged over by v, v', etc. and a countably infinite set *Var* of data variables, ranged over by x, y, etc., which is disjoint from *Val*. We also assume a similar syntactic category *BExp* of boolean expressions, ranged over by b, b', etc. with the usual set of operators $\mathit{true}, \mathit{false}, \neg, \vee, \wedge$ and $\rightarrow$. Furthermore, for every pair of data expressions e, e' we assume that $e = e'$ is a boolean expression. We also assume that each data expression e and each boolean expression b has associated to it a set of free data variables, $\mathit{fv}(e)$ and $\mathit{fv}(b)$ respectively, and that only data variables can occur free in these expressions.

In general we do not worry about the expressive power of these languages although the results on symbolic bisimulations require that the language for boolean expressions is sufficiently expressive to characterize any collection of evaluations over which two open terms are bisimilar.

The definition of the language also assumes a countably infinite set of *channel names* *Chan*, ranged over by c, c', etc., which is disjoint from *Val* and *Var*. To define recursive processes we use a set of *process identifiers* $\mathcal{K}$, ranged over by P, Q, etc. Each identifier is assigned an *arity*, a non-negative integer, indicating the number of parameters it should take to form a process term.

Our language is given by the BNF-definition in Figure 1 and contains the operators listed below. $\mathbf{0}$ is the inactive process capable of doing nothing; $\alpha.t$ is action prefixing where we have three kinds of actions: the invisible action τ, representing internal communication, and the input and output actions $c?x$ and $c!e$ respectively; $\texttt{if}\ b\ \texttt{then}\ t$ is the (one armed) conditional construction; $t + u$ is nondeterministic choice (or summation); $t \mid u$ is parallel composition; $t \setminus L$, where L is a finite subset of *Chan*, is channel restriction; $t[f]$, where f is a partial function from *Chan* to *Chan* with finite domain, is channel renaming. $P(\overline{e})$, where $P \in \mathcal{K}$ and the length of $\overline{e}$ equals the arity of P, is a process which is defined recursively by associating to each such identifier a defining equation of the form

$$P(x_1, \ldots, x_n) \Leftarrow t$$

This way of giving the syntax of recursive processes turns out to be quite intuitive and therefore easy to apply. However, for theoretical purposes, it is sometimes more convenient to have recursion directly in the syntax by introducing an operator of the form $\mathbf{fix}X.\_$ where X is a process variable. This will be discussed in Section 5.5 and for the time being we will use the version based on identifiers and defining equations. In the sequel we will often abbreviate $\alpha.\mathbf{0}$ as α.

In our studies we need the notion of an *evaluation* ρ which is a total mapping from *Var* to *Val*. We use the standard notation $\rho\{v/x\}$ to denote the evaluation which differs from ρ only in that it maps x to v. We assume that an application of ρ to a data expression e, denoted $[\![e]\!]\rho$, always yields a value from *Val* and similarly for boolean expressions, $[\![b]\!]\rho$ is either *true* or *false*. Applications of evaluations also generalize to process terms, denoted $t\rho$, in the obvious way. We assume that these evaluations satisfy some standard properties such as that whenever ρ and ρ' agree on $fv(e)$ then $[\![e]\!]\rho = [\![e]\!]\rho'$. If an expression e has no free variables, namely it is *closed*, then $[\![e]\!]\rho$ is independent of ρ. Consequently we simply write $[\![e]\!]$ to denote its value and use a similar convention for boolean expressions. We will use the suggestive notation $\rho \models b$ to indicate that $[\![b]\!]\rho = \mathit{true}$, and $b \models b'$ to indicate that for every evaluation ρ, $\rho \models b$ implies $\rho \models b'$. Of course we could equally well say that $b \to b'$ is a logical theorem but our notation emphasizes the fact that we wish to describe our theory modulo the semantics of expressions. We will write $b = b'$ for $b \models b'$ and $b' \models b$.

We will also refer to substitutions of data expressions for data variables, and assume that they satisfy the expected properties. We use $e[e'/x]$ to denote the result of substituting e' for all free occurrences of x in e. More generally a substitution σ is a partial mapping from *Var* to *DExp* and we use $e\sigma$ to denote the result of an application of σ to the expression e.

An input prefix is a binding operator in the sense that $c?x.t$ binds x in t. This gives rise to the usual notions of bound and free variables. We will write $bv(t)$ and $fv(t)$ for the sets of bound and free variables in t, respectively. Bound variables lead to the standard notion of α-equivalence $\equiv_\alpha$ over terms and to that of substitution, $t[e/x]$ denoting the result of substituting all free occurrences of x in t by e. The definition of substitution in process terms obviously relies on the definition of substitution in data expressions and may involve renaming of bound variables when necessary to avoid variable capture. A term is *closed* if it contains no free variables, and is *open* otherwise. In a defining equation $P(x_1, \ldots, x_n) \Leftarrow t$ it is required that $fv(t) \subseteq \{x_1, \ldots, x_n\}$.

We let $\mathcal{T}$ denote the set of all process terms, ranged over by t, u etc., and let $\mathcal{P}$ denote the set of all closed terms, ranged over by p, q etc., which we refer to as *processes*.

$$\text{TAU}\quad \frac{}{\tau.t \xrightarrow{\tau}_{\mathsf{e}} t} \qquad\qquad \text{OUTPUT}\quad \frac{}{c!e.t \xrightarrow{c!v}_{\mathsf{e}} t}\ \text{where } v = [\![e]\!]$$

$$\text{INPUT}\quad \frac{}{c?x.t \xrightarrow{c?v}_{\mathsf{e}} t}\ v \in \mathit{Val} \qquad \text{COND}\quad \frac{t \xrightarrow{a}_{\mathsf{e}} t'}{\texttt{if}\ b\ \texttt{then}\ t \xrightarrow{a}_{\mathsf{e}} t'}\ [\![b]\!] = \mathit{true}$$

$$\text{SUM1}\quad \frac{t \xrightarrow{a}_{\mathsf{e}} t'}{t + u \xrightarrow{a}_{\mathsf{e}} t'} \qquad\qquad \text{SUM2}\quad \frac{u \xrightarrow{a}_{\mathsf{e}} u'}{t + u \xrightarrow{a}_{\mathsf{e}} u'}$$

$$\text{PAR1}\quad \frac{t \xrightarrow{a}_{\mathsf{e}} t'}{t \mid u \xrightarrow{a}_{\mathsf{e}} t' \mid u} \qquad\qquad \text{PAR2}\quad \frac{u \xrightarrow{a}_{\mathsf{e}} u'}{t \mid u \xrightarrow{a}_{\mathsf{e}} t \mid u'}$$

$$\text{COM}\quad \frac{t \xrightarrow{c?v}_{\mathsf{e}} t' \quad u \xrightarrow{c!v}_{\mathsf{e}} u'}{t \mid u \xrightarrow{\tau}_{\mathsf{e}} t' \mid u'} \qquad \text{RES}\quad \frac{t \xrightarrow{a}_{\mathsf{e}} t'}{t\backslash L \xrightarrow{a}_{\mathsf{e}} t'\backslash L}\ \mathit{chan}(a) \cap L = \emptyset$$

$$\text{REN}\quad \frac{t \xrightarrow{a}_{\mathsf{e}} t'}{t[f] \xrightarrow{f(a)}_{\mathsf{e}} t'[f]} \qquad\qquad \text{REC}\quad \frac{t[[\![\overline{e}]\!]/\overline{x}] \xrightarrow{a}_{\mathsf{e}} t'}{P(\overline{e}) \xrightarrow{a}_{\mathsf{e}} t'}\ P(\overline{x}) \Leftarrow t$$

Figure 2: (Early) Operational Semantics

We assign the following precedence to the operators (in decreasing order):

restriction and renaming, prefix, conditional, parallel composition, choice

Although we only have the one-armed conditional $\texttt{if}\ b\ \texttt{then}\ \_$ in the language, with the help of $+$ the conventional two-armed $\texttt{if}\ b\ \texttt{then}\ \_\ \texttt{else}\ \_$ expression can be defined by

$$\texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ u = \texttt{if}\ b\ \texttt{then}\ t + \texttt{if}\ \neg b\ \texttt{then}\ u$$

In what follows we will use the $\texttt{if}\,\_\,\texttt{then}\,\_\,\texttt{else}\,\_$ construction freely without further comments.

2.2 Operational Semantics

Our language is given an operational semantics in the usual way by assigning to it a *labelled transition system*, i.e. a triple $\langle \mathtt{S}, \mathtt{L}, \longrightarrow \rangle$ where $\mathtt{S}$ is a set of *states*, $\mathtt{L}$ is a set of *labels* and $\longrightarrow \subseteq \mathtt{S} \times \mathtt{L} \times \mathtt{S}$ is a *transition relation*. Usually we use the more intuitive notation $s_1 \xrightarrow{l} s_2$ in place of $(s_1, l, s_2) \in \longrightarrow$.

The standard operational semantics for our language is obtained by taking the set $\mathtt{S}$ to be $\mathcal{P}$, the set $\mathtt{L}$ to be the set of actions Act_τ, where $\mathit{Act} = \{\, c!v, c?v \mid c \in \mathit{Chan},\ v \in \mathit{Val} \,\}$ and $\mathit{Act}_\tau = \mathit{Act} \cup \{\tau\}$, ranged over by a, and the transition relation to be the relation $\longrightarrow_{\mathsf{e}} \subseteq \mathcal{P} \times \mathit{Act}_\tau \times \mathcal{P}$ defined by the rules in Figure 2. The function *chan* used in the definition is defined by $\mathit{chan}(c?v) = \mathit{chan}(c!v) = \{c\}$ and $\mathit{chan}(\tau) = \emptyset$. As explained in the introduction this semantic approach is usually referred to as the *early* operational semantics (hence the subscript e), because, when an input term such as $c?x.u$ performs the input action, the value received is immediately substituted for the variable x in u. In Section 3.2 we will consider its *late* counterpart where this substitution is delayed.

By transition induction it is easy to show that α-equivalent processes have the same transitions (up to α-equivalence) in the following sense.

Lemma 2.1 *If $p \equiv_\alpha q$ and $p \xrightarrow{a}_{\mathsf{e}} p'$ then $q \xrightarrow{a}_{\mathsf{e}} q'$ for some q' such that $p' \equiv_\alpha q'$.*

3 Bisimulation Equivalences

In this section we introduce the bisimulation based equivalences investigated in this chapter. We shall consider both the late and the early semantics and in each case we also distinguish between the strong version of bisimulation, where τ actions are considered observable, and the weak one where they are not. In what follows we refer to the equivalences introduced in this section as the *concrete equivalences* as opposed to their more abstract versions, the *symbolic equivalences*, that will be introduced in the following section. We start with the early case.

3.1 Early Bisimulation

Using the operational semantics introduced in the previous section we can define the notion of bisimulation in the standard way.

Definition 3.1 A symmetric relation R over $\mathcal{P}$ is an *early strong bisimulation* if it satisfies: $(p, q) \in R$ implies that

if $p \xrightarrow{a}_{\mathsf{e}} p'$ then $q \xrightarrow{a}_{\mathsf{e}} q'$ for some q' such that $(p', q') \in R$.

We write $p \sim_{\mathsf{e}} q$ if $(p,\ q) \in R$ for some early strong bisimulation R. □

Like in the pure case, it is easy to show that the union of any collection of early strong bisimulations is again a bisimulation, and that $\sim_{\mathsf{e}}$ is the union of all early strong bisimulations and therefore the largest such bisimulation. The relation $\sim_{\mathsf{e}}$ turns out to be an equivalence relation and is preserved by all the operators of the language. The proof of these properties is exactly the same as in the pure case and may be found in [Mil89, Section 4.4].

From Lemma 2.1 we get directly that α-equivalent processes are bisimilar in the early strong sense.

Proposition 3.2 *If $p \equiv_\alpha q$ then $p \sim_{\mathsf{e}} q$.*

A more abstract and more realistic way of relating processes behaviourally is by means of *weak* bisimulation where τ-transitions representing internal communications are abstracted away. The standard way of defining this relation is to replace the original transitions with the corresponding weak transition relations in the definition of bisimulation. For this purpose let $\Longrightarrow_{\mathsf{e}}$ be $\xrightarrow{\tau}_{\mathsf{e}}{}^{*}$ and $\overset{a}{\Longrightarrow}_{\mathsf{e}}$ be $\Longrightarrow_{\mathsf{e}}\xrightarrow{a}_{\mathsf{e}}\Longrightarrow_{\mathsf{e}}$ for $a \in Act_\tau$. Let also $\overset{\hat{\tau}}{\Longrightarrow}_{\mathsf{e}}$ be $\Longrightarrow_{\mathsf{e}}$ and $\overset{\hat{a}}{\Longrightarrow}_{\mathsf{e}}$ be $\overset{a}{\Longrightarrow}_{\mathsf{e}}$ for $a \in Act$. The early weak bisimulation is then defined as usual.

Definition 3.3 A symmetric relation R over $\mathcal{P}$ is an *early weak bisimulation* if it satisfies: $(p, q) \in R$ implies that

$$\text{if } p \xrightarrow{a}_{\mathsf{e}} p' \text{ then } q \stackrel{\hat{a}}{\Longrightarrow}_{\mathsf{e}} q' \text{ for some } q' \text{ such that } (p', q') \in R.$$

We write $p \approx_{\mathsf{e}} q$ if $(p,\ q) \in R$ for some early weak bisimulation R. □

Again, like in the strong case, $\approx_{\mathsf{e}}$ is the largest early weak bisimulation and is obtained as the union of all early weak bisimulations.

Proposition 3.4 *$\approx_{\mathsf{e}}$ is an equivalence relation.*

Proof: See [Mil89, Section 5.2, Prop. 2]. □

As in the case of the pure calculus, $\approx_{\mathsf{e}}$ is not a congruence since it is not preserved by $+$. Therefore we introduce the following modified relation:

Definition 3.5 For closed terms p and q, $p \simeq_{\mathsf{e}} q$ if and only if

- whenever $p \xrightarrow{a}_{\mathsf{e}} p'$ then $q \stackrel{a}{\Longrightarrow}_{\mathsf{e}} q'$ for some q' such that $p' \approx_{\mathsf{e}} q'$
- whenever $q \xrightarrow{a}_{\mathsf{e}} q'$ then $p \stackrel{a}{\Longrightarrow}_{\mathsf{e}} p'$ for some p' such that $p' \approx_{\mathsf{e}} q'$.

□

Note that the only difference between the definition of the relations $\approx_{\mathsf{e}}$ and $\simeq_{\mathsf{e}}$ is that in the latter $q \stackrel{\hat{a}}{\Longrightarrow}_{\mathsf{e}} q'$ is replaced by $q \stackrel{a}{\Longrightarrow}_{\mathsf{e}} q'$ on the "top level" of the definition, i.e. the matching move for an initial τ-move has to be a real τ-move.

Proposition 3.6 *$\simeq_{\mathsf{e}}$ is preserved by all the operators in the language and is the largest congruence relation contained in $\approx_{\mathsf{e}}$.*

Proof: See [Mil89, Section 7.2]. □

We refer to $\simeq_{\mathsf{e}}$ as *early observational congruence.*

3.2 Late Bisimulation

In the definition of early bisimulation in the previous section, input variables are instantiated immediately when input actions take place and thus *before* processes are compared for equivalences. As a consequence, input transitions of the form $c?v$ for different values v, arising from a single input prefix $c?x$ from one process, can be matched by transitions inferred from more than one input prefix from the other process. Let us examine this in more detail with an example.

Example 3.7 Consider the following two processes where x ranges over integers.

$$P \Leftarrow (c?x.\texttt{if } x = 0 \texttt{ then } a_1!x \texttt{ else } a_2!x) + c?x.a_3!x,$$
$$Q \Leftarrow (c?x.\texttt{if } x = 0 \texttt{ then } a_3!x \texttt{ else } a_2!x) + (c?x.\texttt{if } x = 0 \texttt{ then } a_1!x \texttt{ else } a_3!x)$$

where a_1, a_2 and a_3 are different channel names. □

Obviously P and Q are bisimilar in the early strong sense as they have isomorphic transition graphs. For instance, a move $P \xrightarrow{c?v} a_3!v$ can be matched by $Q \xrightarrow{c?v} a_3!v$ using Q's first summand when v is 0, and using its second summand when v is different from 0.

In the late semantics we take a more restrictive view of input transitions by requiring that an input prefix gives rise to a *single* transition. Thus we replace the rule INPUT in Figure 2 with

$$\text{LATE-INPUT}\ \frac{}{c?x.t \xrightarrow{c?x} t}$$

We note that the resulting term t is an open term in which x may occur free. Thus the states in the resulting transition system are elements from $\mathcal{T}$, the actions are from the set of *late actions*

$$Act^l_\tau = \{c!v | c \in Chan, v \in Val\} \cup \{c?x | c \in Chan, x \in Var\} \cup \{\tau\}$$

also ranged over by a, and the late transition relation $\longrightarrow_1$, defined by the modified rules, has the functionality

$$\longrightarrow_1 \subseteq \mathcal{P} \times Act^l_\tau \times \mathcal{T}.$$

The free variables in the residuals of input transitions are instantiated when two terms are compared for equivalence i.e. *after* the transitions have taken place (hence the adjective *late*).

Definition 3.8 A symmetric relation R between closed terms is a *late strong bisimulation* if it satisfies: $(p, q) \in R$ implies that

- whenever $p \xrightarrow{c?x}_1 t$ then $q \xrightarrow{c?y}_1 u$ for some u such that $(t[v/x], u[v/y]) \in R$ for all $v \in Val$
- whenever $p \xrightarrow{a}_1 p'$, $a \in \{\tau, c!v \mid c \in Chan, v \in Val\}$, then $q \xrightarrow{a}_1 q'$ for some q' such that $(p', q') \in R$.

We write $p \sim_1 q$ if $(p,\ q) \in R$ for some late strong bisimulation R. □

Note that, although $\longrightarrow_1$ is defined between open terms, the relation $\sim_1$ is between closed terms only, because input variables get instantiated when input transitions are matched for bisimulation.

According to this definition the two processes P and Q in the above example are *not* bisimilar, because, for instance, Q can not match $P \xrightarrow{c?x} a_3!x$ using either of its two input transitions.

It is not difficult to see that the union of any collection of late strong bisimulations is again a late strong bisimulation. Furthermore $\sim_1$ is the union of all late strong bisimulations and thus the largest such bisimulation. It is also easy to see that $\equiv_\alpha$ is included in $\sim_1$, and that $\sim_1$ is included in $\sim_e$, i.e. that late bisimulation is a finer relation than its early counterpart. This inclusion is strict as the above example demonstrates.

It can also be shown that $\sim_1$ is an equivalence relation which is preserved by all the operators of the language. The proofs of these two facts are similar (and simpler) than those of Propositions 3.12, and 3.15, and are omitted.

If we exchange the existential and universal quantifiers in the first clause in the above definition, that is, if we change the clause to

- whenever $p \xrightarrow{c?x}_1 t$ then for all $v \in Val$, $q \xrightarrow{c?y}_1 u$ for some u and y such that $(t[v/x], u[v/y]) \in R$,

then it can be shown that the resulting relation, which we refer to as $\forall\exists$-bisimulation here, coincides with the early bisimulation of Definition 3.1.

Proposition 3.9 *A relation R over $\mathcal{P}$ is a $\forall\exists$-bisimulation if and only if it is an early bisimulation.*

Proof: First, by a simple transition induction, we can prove that the following relationship holds between the early and the late transitions:

1. If $p \xrightarrow{c?x}_1 t$ then $p \xrightarrow{c?v}_e t[v/x]$ for any $v \in Val$.
   If $p \xrightarrow{c?v}_e p'$ then $p \xrightarrow{c?x}_1 t$ for some x and t with $p' \equiv t[v/x]$.
2. For $a \in \{\, \tau,\ c!v \mid c \in Chan, v \in Val \,\}$, $p \xrightarrow{a}_1 q$ if and only if $p \xrightarrow{a}_e q$.

The statement of the proposition then follows easily from these properties. $\square$

Now we turn towards defining the weak version of the late bisimulation. As usual we shall base our definition on a weak version of the transition relations. However here care must be taken in the case of input actions, because now the residual may be an open term which has no transitions. Thus, in the definition of the late weak transitions, an input transition *can not* absorb τ moves that occur after it. These τ moves will be absorbed when processes are compared for equivalence.

Let $\Longrightarrow_1$ be ${\xrightarrow{\tau}_1}^*$, $\overset{c?x}{\Longrightarrow}_1$ be $\Longrightarrow_1\xrightarrow{c?x}_1$, and $\overset{a}{\Longrightarrow}_1$ be $\Longrightarrow_1\xrightarrow{a}_1\Longrightarrow_1$ for $a \in \{\tau, c!v \mid v \in Val\}$. These weak transition relations can now be used to define the late version of the weak bisimulation relation.

Definition 3.10 A symmetric relation R over $\mathcal{P}$ is a *late weak bisimulation* if it satisfies: $(p, q) \in R$ implies that

- whenever $p \xrightarrow{c?x}_1 t$ then $q \overset{c?y}{\Longrightarrow}_1 u$ for some u such that for all $v \in Val$ there exists q' such that $u[v/y] \Longrightarrow_1 q'$ and $(t[v/x], q') \in R$
- whenever $p \xrightarrow{a}_1 p'$, where $a \in \{\tau,\ c!v \mid c \in Chan, v \in Val\}$, then $q \overset{\hat{a}}{\Longrightarrow}_1 q'$ for some q' such that $(p', q') \in R$.

We write $p \approx_1 q$ if $(p,\ q) \in R$ for some late weak bisimulation R. $\square$

Note in this definition how τ moves following an input transition are absorbed *after* the input variable gets instantiated.

Again the union of any collection of late weak bisimulations is also a late weak bisimulation. Hence, as $\approx_1$ is the the union of all late weak bisimulations, it is the largest such bisimulation.

To show that $\approx_1$ is an equivalence relation we need a lemma.

Lemma 3.11 *Suppose that R is a late weak bisimulation and that $(p, q) \in R$. Then the following holds:*

1. *If $p \stackrel{c?x}{\Longrightarrow}_1 t$ then $q \stackrel{c?y}{\Longrightarrow}_1 u$ for some u such that for all $v \in Val$ there exists q' such that $u[v/y] \Longrightarrow_1 q'$ and $(t[v/x], q') \in R$.*
2. *If $p \stackrel{\hat{a}}{\Longrightarrow}_1 p'$, where $a \in \{\tau,\ c!v \mid c \in Chan, v \in Val\}$, then $q \stackrel{\hat{a}}{\Longrightarrow}_1 q'$ for some q' such that $(p', q') \in R$.*

Proof: Follows by induction on the definition of the late weak transition relation. $\square$

Now we have the following.

Proposition 3.12 $\approx_1$ *is an equivalence relation.*

Proof: The only non-trivial part is to prove that $\approx_1$ is transitive. To this end we show that if both R and S are late weak bisimulations then so is their composition RS. Assume $(p, q) \in RS$. Then there is an r such that $(p, r) \in R$ and $(r, q) \in S$. We need to show that every transition from p can be matched properly by q. We only consider the case when the transition is an input transition.

Assume that $p \stackrel{c?x}{\longrightarrow}_1 t$. Since $(p, r) \in R$, $r \stackrel{c?z}{\Longrightarrow}_1 w$ for some w and z such that for all $v \in Val$ there is an r' with $w[v/z] \Longrightarrow_1 r'$ and $(t[v/x], r') \in R$. Since $(r, q) \in S$, by Lemma 3.11 we have that $q \stackrel{c?y}{\Longrightarrow}_1 u$ for some u and y such that for all $v \in Val$ there is a q'' with $u[v/x] \Longrightarrow_1 q''$ and $(w[v/x], q'') \in S$. Since $w[v/x] \Longrightarrow_1 r'$, by Lemma 3.11, there is a q' such that $q'' \Longrightarrow_1 q'$ and $(r', q') \in S$. Furthermore we have that $u[v/x] \Longrightarrow_1 q'$ because $u[v/x] \Longrightarrow_1 q''$ and $q'' \Longrightarrow_1 q'$. Finally $(t[v/x], q') \in RS$ follows from the fact that $(t[v/x], r') \in R$ and $(r', q') \in S$. $\square$

Proposition 3.13 $\approx_1$ *is preserved by all operators in the language except $+$.*

Proof: We only prove that $\approx_1$ is preserved by the parallel operator as the remaining cases are similar. Assume that R is a late weak bisimulation. We show that the relation $R' = \{\, (p \mid r,\ q \mid r,) \mid (p,\ q) \in R,\ r \in \mathcal{P}\,\}$ is also a late weak bisimulation. To this end assume $p \mid r \stackrel{a}{\longrightarrow}_1 p' \mid r'$. The only interesting case is when a has the form $c?x$. There are two possibilities:

- $p \xrightarrow{c?x}_1 p'$ and $r \equiv r'$. Since $(p,q) \in R$, there exist y and q' such that $q \overset{c?y}{\Longrightarrow}_1 q'$ and for all $v \in \mathit{Val}$ there is some q'' such that $q'[v/y] \Longrightarrow_1 q''$ and $(p'[v/x], q'') \in R$. This implies $q \mid r \overset{c?y}{\Longrightarrow}_1 q' \mid r$ and, since r is a closed term, $(q' \mid r)[v/y] \equiv q'[v/y] \mid r \Longrightarrow_1 q'' \mid r$. Furthermore $((p' \mid r)[v/x], q'' \mid r) \in R'$ because $(p' \mid r)[v/x] \equiv p'[v/x] \mid r$ and $(p'[v/x], q'') \in R$.

- $p \equiv p'$ and $r \xrightarrow{c?x}_1 r'$. Straightforward.

The relation $\approx_1$ is not a congruence; it is not preserved by $+$ as shown by the following standard example:

$$\tau.c?x \approx_1 c?x \quad \text{but} \quad \tau.c?x + d?x \not\approx_1 c?x + d?x$$

$\square$

To obtain a relation which is preserved by all the operators we follow the same approach as in the early case, introducing a modified version of $\approx_1$:

Definition 3.14 For $p, q \in \mathcal{P}$, $p \simeq_1 q$ if and only if

- whenever $p \xrightarrow{c?x}_1 t$ then $q \overset{c?y}{\Longrightarrow}_1 u$ for some y and u such that for all $v \in Val$ there exists some q' such that $u[v/y] \Longrightarrow_1 q'$ and $t[v/x] \approx_1 q'$

- whenever $a \in \{\, \tau, c!v \mid c \in \mathit{Chan}, v \in \mathit{Val} \,\}$ and $p \xrightarrow{a}_1 p'$ then $q \overset{a}{\Longrightarrow}_1 q'$ for some q' such that $p' \approx_1 q'$

and similarly for q. $\square$

The following proposition gives an alternative characterization of $\simeq_1$ in terms of $\approx_1$.

Proposition 3.15 *$p \simeq_1 q$ if and only if $p + r \approx_1 q + r$ for all $r \in \mathcal{P}$.*

Proof: The "only if" direction can be established by showing that $\{\, (p + r, q + r) \mid p, q, r \in \mathcal{P}$ and $p \simeq_1 q \,\}$ is a late weak bisimulation, which is straightforward.

For the "if" direction, we prove the contrapositive. Assume that $p \not\simeq_1 q$; we will prove that $p + r \not\approx_1 q + r$ for some r. By the assumption there exist, for example, a and t such that $p \xrightarrow{a}_1 t$ and the following holds:

- If a is τ or an output action, then $q \overset{a}{\Longrightarrow}_1 u$ implies $t \not\approx_1 u$;

- If $a \equiv c?x$ and $q \overset{c?y}{\Longrightarrow}_1 u$, then there exists some $v \in \mathit{Val}$ such that $u[v/y] \Longrightarrow_1 q'$ implies $t[v/x] \not\approx_1 q'$.

Now choose r to be $d?x.\mathbf{0}$ where d does not occur in p or q or in the codomain of any renaming functions used in these terms. We show $p + r \not\approx_1 q + r$. From the assumption we have $p + r \xrightarrow{a}_1 t$ and we will prove that this move can not be matched properly by $q + r$. There are two cases to consider.

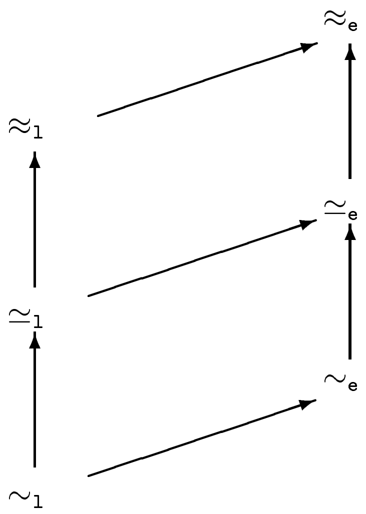


Figure 3: The Relationship between Equivalences

- If $a \equiv \tau$ and $u \equiv q + r$, then $t \not\approx_1 u$ because the transition $u \xrightarrow{d?x}_1 \mathbf{0}$ cannot be matched by t.
- Otherwise,
  - If a is not an input action then for all u such that $q + r \overset{a}{\Longrightarrow}_1 u$ we also have $q \overset{a}{\Longrightarrow}_1 u$ and therefore, by the assumption, $t \not\approx_1 u$;
  - If a is an input action of the form $c?x$ then $c \neq d$. Therefore for any u such that $q+r \overset{c?y}{\Longrightarrow}_1 u$ it is also the case that $q \overset{c?y}{\Longrightarrow}_1 u$. Hence there is some $v \in \mathit{Val}$ such that whenever $u[v/y] \Longrightarrow_1 q'$ then $t[v/x] \not\approx_1 q'$.

This proves that $p + r \not\approx_1 q + r$ and thus the proposition follows by contraposition. $\square$

It follows from Proposition 3.15 that $\simeq_1$ is a congruence relation and that it is the largest such relation contained in $\approx_1$. We refer to $\simeq_1$ as *late observational congruence.*

There is an alternative way of presenting the late semantics. Instead of using the input transition rule $c?x.t \xrightarrow{c?x}_1 t$ where the input variable is carried in the transition and the residual is an open term, one can define $c?x.t \xrightarrow{c?}_1 (x)t$ where the input variable is abstracted away from the transition and $(x)t$ is a new syntactical category called *abstraction.* Semantically an abstraction is a function from values to processes. Then the input clause in Definition 3.2 is changed accordingly:

If $p \xrightarrow{c?}_1 f$ then, for some g, $q \xrightarrow{c?}_1 g$ and $(fv,\ gv) \in R$ for all $v \in \mathit{Val}$.

Abstractions have been used extensively in the study of the π-calculus [Mil92]. We have chosen to use the open terms presentation because it is in harmony with the symbolic approach, the main subject of this chapter.

The bisimulation-based equivalence relations we have studied so far are summarized in Figure 3 where arrows stand for inclusion.

p	$\hat{p}$
$c?x.t$	$\sum_{v \in Val} c?v.\widehat{t[v/x]}$
$c!e.t$	$c![\![e]\!].\hat{t}$
$\tau.t$	$\tau.\hat{t}$
$t + u$	$\hat{t} + \hat{u}$
$t \mid u$	$\hat{t} \mid \hat{u}$
$t \backslash L$	$\hat{t} \backslash \{ c?v, c!v \mid c \in L \}$
$t[f]$	$\hat{t}[\hat{f}]$, where $\hat{f}(c?v) = f(c)?v$, $\hat{f}(c!v) = f(c)!v$
$\texttt{if } b \texttt{ then } t$	$\begin{cases} \hat{t} & \text{if } [\![b]\!] = \mathit{true} \\ \mathbf{0} & \text{otherwise} \end{cases}$
$P(\overline{e})$	$P_{\overline{[\![e]\!]}}$

Figure 4: The Translation Rules

3.3 Translating into Pure Processes

In this section we explain briefly how the language for value-passing processes may be translated into the "pure" language, as proposed in [Mil89, Section 2.8]. We assume the standard notations for pure CCS as in [Mil89]. The key step in the translation is to compile an input prefix into a summation over all possible values in the data domain. So, for each channel name c we introduce a set of action labels $\{ c?v, c!v \mid c \in \mathit{Chan},\ v \in \mathit{Val} \}$ in the pure calculus. The complement of $c?v$ is $c!v$, and vice versa. Also for each process identifier P of arity n in the value-passing language we introduce a set of indexed process identifiers $\{ P_{\overline{v}} \mid \overline{v} \in \mathit{Val}^n \}$ in the pure language. The translation is defined by induction on the structure of process expressions and is given in Figure 4 where p is the original value-passing process and $\hat{p}$ its translated form.

Furthermore, each parameterized definition $P(\overline{x}) \Leftarrow t$ is translated into a set of non-parameterized defining equations

$$\{ P_{\overline{v}} \Leftarrow \widehat{t[\overline{v}/\overline{x}]} \mid \overline{v} \in \mathit{Val}^n \}$$

Note that this translation is only defined for closed terms. This is sufficient because the only construction which may introduce free variables is input prefixing, and in the translation each input variable is instantiated with a value. Also note that, when Val is infinite, an input prefixing is translated into an infinite summation, and a parameterized definition into an infinite set of non-parameterized equations.

Now the resulting terms of the translation are terms in the underlying pure process calculus (pure CCS in this case) which is equipped with a well-defined operational semantics and the standard notion of bisimulation. This provides a simple way of giving semantics to value-passing processes via translation (at the cost of infinity). It is easy to check that the semantics so obtained coincides with the early semantics. To be precise, let $\longrightarrow$ be the transition relation corresponding to the standard operational semantics of pure CCS (as defined in [Mil89, Section 2.5]). We have the following exact correspondence between the early operational semantics defined in Figure 2 and $\longrightarrow$:

Lemma 3.16 *For $p, q \in \mathcal{P}$ and $a \in Act_\tau$, $p \xrightarrow{a}_{\mathsf{e}} q$ if and only if $\hat{p} \xrightarrow{a} \hat{q}$.*

Proof: Follows by transition induction, using the definition of $\hat{p}$. $\square$

Let $\sim$ be the bisimulation relation as defined in [Mil89, Section 4.2]. The following proposition describes the relationship between $\sim_{\mathsf{e}}$ and $\sim$:

Proposition 3.17 *For $p, q \in \mathcal{P}$, $p \sim_{\mathsf{e}} q$ if and only if $\hat{p} \sim \hat{q}$.*

Proof: Suppose R is an early bisimulation with $(p, q) \in R$. Define $\hat{R} = \{ (\hat{p}, \hat{q}) \mid (p, q) \in R \}$. By Lemma 3.16 it is immediate that $\hat{R}$ is a pure-CCS bisimulation.

For the other direction, let $\hat{R}$ be a pure-CCS bisimulation. Define $R = \{ (p, q) \mid (\hat{p}, \hat{q}) \in \hat{R} \}$. Again it is immediate that R is an early bisimulation. $\square$

The translation approach has been implemented in a tool named VP [Bru96] which acts as a front-end for the verification tool Concurrency Workbench. It takes an input file written in full CCS (with finite data domain) and translates it into pure CCS in a format that can be accepted by the Concurrency Workbench. The translation is carried out exactly according to the rules in Figure 4.

4 Symbolic Bisimulations

As explained in the Introduction, the main problem in connection with the traditional semantics for value-passing processes is that in general it leads to infinitely branching transition graphs. Therefore, to check for equivalence, infinitely many transitions have to be investigated. The aim of this section is to give a more abstract version of the concrete semantics we have considered so far which allows us to check for equivalence by only investigating finitely branching graphs.

4.1 Symbolic Operational Semantics

To motivate the rules for the symbolic operational semantics let us look at a simple example. Let P be defined as

$$P \Leftarrow c?x.\mathtt{if}\ x > 0\ \mathtt{then}\ d!x\ \mathtt{else}\ d!(-x)$$

where x ranges over integers. According to the operational semantics in Figure 2, P has a transition $\xrightarrow{c?n}$ for *each* integer n, hence its transition graph has infinitely many branches. This seems counter-intuitive since all P can do is first to input a number from channel c and then output on channel d either the number itself or its negation, depending on whether it is positive or not. This is a *finitary* description and conceptually we could take $P \xrightarrow{c?x} \mathtt{if}\ x > 0\ \mathtt{then}\ d!x\ \mathtt{else}\ d!(-x)$ as a *single* abstract transition, without instantiating x. This kind of transition has been used in defining late bisimulation in Section 3.2. However, there the instantiation is only delayed and in the input

$$\textsc{Pre}\ \frac{}{\alpha.t \xrightarrow{true,\alpha} t} \qquad \textsc{Cond}\ \frac{t \xrightarrow{b',\alpha} t'}{\texttt{if}\ b\ \texttt{then}\ t \xrightarrow{b'\wedge b,\alpha} t'}\ fv(b) \cap bv(\alpha) = \emptyset$$

$$\textsc{Sum1}\ \frac{t \xrightarrow{b,\alpha} t'}{t + u \xrightarrow{b,\alpha} t'} \qquad \textsc{Sum2}\ \frac{u \xrightarrow{b,\alpha} u'}{t + u \xrightarrow{b,\alpha} u'}$$

$$\textsc{Par1}\ \frac{t \xrightarrow{b,\alpha} t'}{t \mid u \xrightarrow{b,\alpha} t' \mid u}\ bv(\alpha) \cap fv(u) = \emptyset \quad \textsc{Par2}\ \frac{u \xrightarrow{b,\alpha} u'}{t \mid u \xrightarrow{b,\alpha} t \mid u'}\ bv(\alpha) \cap fv(t) = \emptyset$$

$$\textsc{Com}\ \frac{t \xrightarrow{b,c?x} t' \quad u \xrightarrow{b',c!e} u'}{t \mid u \xrightarrow{b\wedge b',\tau} t'[e/x] \mid u'} \qquad \textsc{Res}\ \frac{t \xrightarrow{b,\alpha} t'}{t\backslash L \xrightarrow{b,\alpha} t'\backslash L}\ Chan(\alpha) \notin L$$

$$\textsc{Ren}\ \frac{t \xrightarrow{b,\alpha} t'}{t[f] \xrightarrow{b,f(\alpha)} t'[f]} \qquad \textsc{Rec}\ \frac{t[\overline{e}/\overline{x}] \xrightarrow{b,\alpha} t'}{P(\overline{e}) \xrightarrow{b,\alpha} t'}\ P(\overline{x}) \Leftarrow t$$

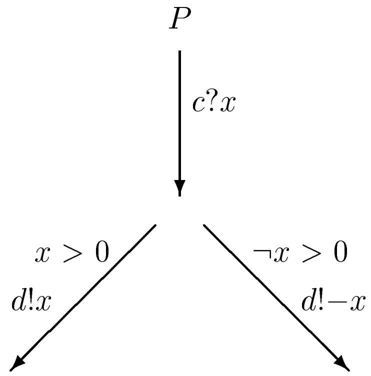
Figure 5: Symbolic Transitional Semantics

clause of the definition of late bisimulation, infinitely many comparisons for the residuals are still required, one for each possible value of the data domain. Here we want to go further, to avoid such infinite comparisons as well. As the residual of such an abstract input transition is in general an open term, we will neither be able to evaluate boolean conditions nor data expressions in output prefixes. Instead we shall allow these expressions to be carried around unevaluated, i.e. *symbolically*, in the transitions. So in the above example we shall have that $\texttt{if}\ x > 0\ \texttt{then}\ d!x\ \texttt{else}\ d!(-x) \xrightarrow{x>0,d!x} \mathbf{0}$ and $\texttt{if}\ x > 0\ \texttt{then}\ d!x\ \texttt{else}\ d!(-x) \xrightarrow{\neg(x>0),d!(-x)} \mathbf{0}$. This leads to the notion of *symbolic transition relations* which have the form $\xrightarrow{b,\ \alpha}$ where b is a boolean expression and α a symbolic action, namely

$$\alpha \in Act_\tau^{symb} = \{\, \tau, c!e, c?x \mid c \in Chan, e \in DExp, x \in Var \,\}$$

Intuitively $t \xrightarrow{b,\ \alpha} u$ means that, under circumstances in which b holds, t can perform α and thereby evolve into u. We will abbreviate $\xrightarrow{true,\ \alpha}$ as $\xrightarrow{\alpha}$. The set of bound variables of a symbolic action is defined thus: $bv(c?x) = \{x\}$ and $bv(\tau) = bv(c!e) = \emptyset$. Formally the symbolic operational semantics for the language is obtained by taking $\mathcal{T}$ as the set of states, $BExp \times Act_\tau^{symb}$ as the set of labels and $\longrightarrow \subseteq \mathcal{T} \times (BExp \times Act_\tau^{symb}) \times \mathcal{T}$ as defined in Figure 5. These relations are defined up-to α-equivalence. The rule $\textsc{Par1}$ has a side condition which ensures that, in the case when α is an input action of the form $c?x$, the variable x which is local in t will not be confused with any free variables in u. This comment applies to $\textsc{Par2}$ as well. The rule $\textsc{Cond}$ also has a side condition to make sure that whenever $t \xrightarrow{b,c?x} t'$ the boolean condition b does not mention x; the occurrence of $c?x$ in the label of the transition means that typically we are about to introduce x as a fresh variable in the sequel t' and therefore x should not be constrained by the already existing b.

As an example, the process P defined above has the following (finite) symbolic transition graph derived from the rules in Figure 5.



Note that we do not subscribe these symbolic transition relations because they will be used to define both the early and the late versions of strong symbolic bisimulation.

It is easy to check that whenever $t \xrightarrow{b,\alpha} t'$ then $fv(b) \subseteq fv(t)$, $bv(\alpha) \cap fv(b) = \emptyset$ and $fv(t') \subseteq fv(t) \cup bv(\alpha)$.

4.2 Early Symbolic Bisimulation

Based on symbolic transitional semantics we now set to define *symbolic bisimulation*. To motivate the definition, let us consider the two processes

$$P \Leftarrow c?x.((\tau.\texttt{if } x = 0 \texttt{ then } d!0 \texttt{ else } d!1) + \tau.d!2)$$
$$Q \Leftarrow c?x.(\tau.(\texttt{if } x = 0 \texttt{ then } d!0 \texttt{ else } d!2) + \tau.(\texttt{if } x = 0 \texttt{ then } d!2 \texttt{ else } d!1))$$

Their corresponding symbolic transition graphs are shown in Figure 6.

It is easy to check that $p \sim_{\mathtt{e}} q$ directly using Definition 3.1 but let us instead have a look at how this fact might be deduced from the symbolic transition semantics. The symbolic move $p \xrightarrow{c?x} P_1$ can be matched by the corresponding move $q \xrightarrow{c?x} Q_1$ if we can show that P_1 and Q_1 are symbolically equivalent. Here special care must be taken as we can not expect, for example, the move $P_1 \xrightarrow{\tau} P_{11}$ to be entirely matched by one move from Q_1, say $Q_1 \xrightarrow{\tau} Q_{11}$. These two moves actually only match one another in the case when the value associated with x is 0. For other values of x this move from P_1 has to be matched by $Q_1 \xrightarrow{\tau} Q_{12}$. In general we must allow a symbolic move from one (open) term to be matched by symbolic moves arising from different components of the other for different subsets of the value space associated with their free variables. We can use boolean expressions to represent such subspaces. For instance the boolean expression $x > 1$ indicates that x should be instantiated with integers which are greater than 1.

This leads us to considering a symbolic equivalence parameterized by boolean expressions, $\sim^b$. In the present example we will have that $P_{11} \sim^b Q_{11}$ in the case where b implies $x = 0$ and that $P_{11} \sim^b Q_{12}$ when b implies $x \neq 0$. Let us now reexamine how the nodes P_1 and Q_1 can be properly matched, i.e. why we can claim that $P_1 \sim^b Q_1$ for

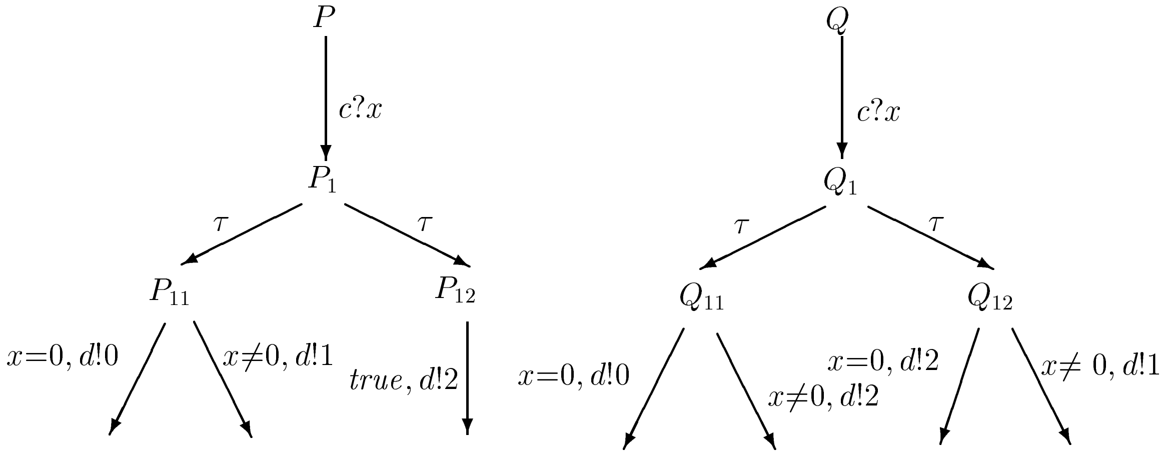


Figure 6: Symbolic Transition Graphs for P and Q

a particular b. The most we can hope for in matching the typical move $P_1 \xrightarrow{\tau} P_{11}$ is that the case we are investigating, represented by b, can be divided up into finitely many different sub-cases and for each sub-case the move can be properly matched by Q. Such a set of sub-cases can be given by a finite set of booleans B such that $b = \bigvee B$. For example if we are trying to check if $P_1 \sim^{true} Q_1$ then we can consider the cases given by $B = \{x = 0,\ x \neq 0\}$; in the case $x = 0$ the move is matched by $Q_1 \xrightarrow{\tau} Q_{11}$ while in the case $x \neq 0$ it is matched by $Q_1 \xrightarrow{\tau} Q_{12}$.

A finite set of boolean expressions B is called a *b-partition* if $\bigvee B = b$. Semantically B can be regarded as a partition of the value space represented by b, or more precisely for any evaluation ρ, $\rho \models b$ if and only if $\rho \models b'$ for some $b' \in B$.

In what follows we write $\alpha =^b \alpha'$ to mean that if $\alpha \equiv c!e$ then α' has the form $c!e'$ with $b \models e = e'$, and $\alpha' \equiv \alpha$ otherwise.

Definition 4.1 A *BExp*-indexed family of symmetric relations $\mathbf{S} = \{\, S^b \mid b \in BExp \,\}$ is an *early strong symbolic bisimulation* if it satisfies: $(t, u) \in S^b$ implies that

> if $t \xrightarrow{b_1,\alpha} t'$ with $bv(\alpha) \cap fv(b, t, u) = \emptyset$, then there is a $b \wedge b_1$-partition B such that for each $b' \in B$ there exist b_2, α' and u' with $b' \models b_2$, $\alpha =^{b'} \alpha'$, $u \xrightarrow{b_2,\alpha'} u'$ and $(t', u') \in S^{b'}$.

We write $t \sim^b_{\mathtt{E}} u$ if $(t, u) \in S^b \in \mathbf{S}$ for some early symbolic bisimulation $\mathbf{S}$. □

As an illustration let us consider again the example introduced at the beginning of this section. Let

$$\begin{aligned} A_1 &= \{(P, Q), (P_1, Q_1)\} \\ A_2 &= \{(P_{11}, Q_{11}), (P_{12}, Q_{12})\} \\ A_3 &= \{P_{11}, Q_{12}), (P_{12}, Q_{11})\} \end{aligned}$$

Set

$$S^{true} = A_1 \cup A_1^{-1} \cup \mathbf{id}, \quad S^{x=0} = A_2 \cup A_2^{-1} \cup \mathbf{id}, \quad S^{x\neq 0} = A_3 \cup A_3^{-1} \cup \mathbf{id}$$

where **id** is the identity relation. Then $\mathbf{S} = \{S^{true},\ S^{x=0},\ S^{x\neq 0}\}$ is a symbolic early bisimulation. To check this let us investigate $(P_1, Q_1) \in S^{true}$. We know that $P_1 \stackrel{true,\tau}{\longrightarrow} P_{11}$ and we must show that Q_1 can match this move properly. For this purpose we use $B = \{x = 0, x \neq 0\}$ as the partition for *true*. Then we have two cases to investigate: $x = 0$ and $x \neq 0$. As an example we consider the first one, $x = 0$. If we take $b_2 = true$ in the definition then we see that $x = 0 \models true$, $\tau =^{x=0} \tau$, $Q_1 \stackrel{true,\tau}{\longrightarrow} Q_{11}$ and $(P_{11}, Q_{11}) \in S^{x=0}$, as required by the definition. The reader is encouraged to investigating the other cases keeping in mind that $B = \emptyset$ is a partition for *false* or any boolean equivalent to *false*.

In the above example, the need for a partition arises from the boolean guards carried around in the symbolic transitions. However, this is not the only reason why partitions are needed. Consider, for instance, the following definitions (where x ranges over integers):

$$\begin{aligned} P_1(x) &\Leftarrow c!x + c!|x| + c!-|x| \\ P_2(x) &\Leftarrow c!|x| + c!-|x| \end{aligned}$$

To establish $P_1(x) \sim_{\mathtt{E}}^{true} P_2(x)$ we need to use a partition $\{x \geq 0,\ x \leq 0\}$ (or, equivalently, $\{x = |x|,\ x = -|x|\}$) for *true*, although all symbolic transitions involved carry only the trivial boolean guard *true*.

It is worth pointing out that when α is an input action of the form $c?x$, i.e. when $t \stackrel{b_1,c?x}{\longrightarrow} t'$, the input variable x may occur *free* in the partition B. This allows one to perform a case analysis on the value space associated with x *before* the input transition takes place, thus permitting to use different matching transitions from u for different cases. This reflects the "early instantiation" strategy. As an illustration let us consider again Example 3.7 introduced at the beginning of Section 3.2:

$$\begin{aligned} &P \Leftarrow (c?x.\mathtt{if}\ x = 0\ \mathtt{then}\ a_1!x\ \mathtt{else}\ a_2!x) + c?x.a_3!x \\ &Q \Leftarrow (c?x.\mathtt{if}\ x = 0\ \mathtt{then}\ a_3!x\ \mathtt{else}\ a_2!x) + (c?x.\mathtt{if}\ x = 0\ \mathtt{then}\ a_1!x\ \mathtt{else}\ a_3!x) \end{aligned}$$

As mentioned before P and Q are early bisimilar. To show that $P \sim_{\mathtt{E}}^{true} Q$, we need to match the symbolic move $P \stackrel{true,c?x}{\longrightarrow} a_3!x$ by moves from Q. We can use the partition $B = \{x = 0,\ x \neq 0\}$ for *true*. In the case of $x = 0$, this move is matched by

$$Q \stackrel{true,c?x}{\longrightarrow} \mathtt{if}\ x = 0\ \mathtt{then}\ a_3!x\ \mathtt{else}\ a_2!x$$

while in the case of $x \neq 0$ it is matched by

$$Q \stackrel{true,c?x}{\longrightarrow} \mathtt{if}\ x = 0\ \mathtt{then}\ a_1!x\ \mathtt{else}\ a_3!x$$

Here it is essential to use a partition in which x occurs free. Should this be disallowed it would not be possible to establish $P \sim_{\mathtt{E}}^{true} Q$.

Next we will determine the relationship between the symbolic and the concrete versions of the bisimulations. We start by showing the connection between the symbolic transitions and the concrete ones. Since symbolic transitions are defined for open terms, we need an evaluation mapping which assigns values to data variables.

Lemma 4.2 *For all t, u, p and ρ,*

1. *$t\rho \stackrel{\tau}{\longrightarrow}_{\mathtt{e}} p$ if and only if there exist b, t' such that $\rho \models b$, $p \equiv_\alpha t'\rho$ and $t \stackrel{b,\tau}{\longrightarrow} t'$,*

2. $t\rho \xrightarrow{c!v}_{\mathsf{e}} p$ *if and only if there exist* b, e, t' *such that* $\rho \models b$, $[\![e]\!]\rho = v$, $p \equiv_\alpha t'\rho$ *and* $t \xrightarrow{b,c!e} t'$,

3. $t\rho \xrightarrow{c?v}_{\mathsf{e}} p$ *if and only if there exist* b, t' *and* $x \notin fv(t)$ *such that* $\rho \models b$, $p \equiv_\alpha t'\rho\{v/x\}$ *and* $t \xrightarrow{b,c?x} t'$.

Proof: Follows by transition induction. The details of the proof can be found in [HL96, page 389]. $\Box$

The early strong symbolic bisimulation corresponds to its concrete counterpart in the following sense:

Theorem 4.3 $t \sim^b_{\mathtt{E}} u$ *if and only if* $t\rho \sim_{\mathsf{e}} u\rho$ *for every evaluation* ρ *such that* $\rho \models b$.

Proof: (Outline) To prove the "only if" direction assume that $\mathbf{S} = \{\, S^b \mid b \in BExp \,\}$ is an early strong symbolic bisimulation and let

$$R_{\mathbf{S}} = \{\, (t\rho, u\rho) \mid \exists b.\ \rho \models b \text{ and } (t,u) \in S^b \,\}$$

The result follows immediately if we can prove that $R_{\mathbf{S}}$ is a bisimulation.

For the "if" direction, assume that R is an early strong bisimulation and let

$$S^b_R = \{\, (t,u) \mid \forall \rho.\ \rho \models b \text{ implies } (t\rho, u\rho) \in R \,\}$$

for any boolean expression b. Again the result follows if we can show that $\mathbf{S}_R = \{\, S^b_R \mid b \in BExp \,\}$ is a symbolic bisimulation. Details can be found in [HL95, Theorem 6.5]. $\Box$

It follows from this theorem that for closed terms p and q, $p \sim_{\mathsf{e}} q$ if and only if $p \sim^{true}_{\mathtt{E}} q$.

As a further corollary to this theorem, if $t \sim^b_{\mathtt{E}} u$ then $t[y/x] \sim^{b[y/x]}_{\mathtt{E}} u[y/x]$ for any $y \notin fv(b,t,u)$. Thus *symbolic bisimulations are invariant under renaming of free variables.*

Definition 4.1 requires that, for t and u to be symbolically bisimilar, if $t \xrightarrow{b_1,c?x} t'$ then u must match this transition with an input transition $u \xrightarrow{b_2,c?x} u'$, on the same channel, using the same input variable. Since the rule Pre in Figure 5 asserts $\alpha.t \xrightarrow{true,\alpha} t$, the reader may wonder how to establish that, for instance, $c?x.d!x$ and $c?y.(d!y + d!y)$ are symbolically bisimilar according to this definition. Since symbolic transitional semantics is defined upto α-equivalence, we may choose, for instance, $c?x.d!x \equiv_\alpha c?y.d!y \xrightarrow{true,c?y} d!y$ which can be matched by $c?y.(d!y+d!y) \xrightarrow{true,c?y} d!y+d!y$. Of course we may use *any* data variable z in the input action, provided it is not free in the terms under investigation. The important fact is that, when comparing a pair of input transitions for symbolic bisimulation, *only one* such variable is needed. It is immaterial which one is chosen. In general, if $t \xrightarrow{b_1,c?x} t'$, $u \xrightarrow{b_2,c?x} u'$ and $t' \sim^{b'}_{\mathtt{E}} u'$, then for any $z \notin fv(t,u)$, $t \xrightarrow{b_1,c?z} t'[z/x]$, $u \xrightarrow{b_2,c?z} u'[z/x]$, and we also have $t'[z/x] \sim^{b'[z/x]}_{\mathtt{E}} u'[z/x]$, because symbolic bisimulations are invariant under renaming of free variables. This remark applies to all symbolic equivalence relations discussed in this chapter.

4.3 Late Symbolic Bisimulation

Recall that in Section 3.2, when defining late bisimulations, we have to modify the transition semantics for input prefixing to allow abstract input transitions of the form $\xrightarrow{c?x}$, so that the instantiation of values for input variables can be deferred after input actions are matched for bisimulation. Since the symbolic operational semantics of Figure 5 already uses abstract actions, the same rules can be adopted to define the late version of symbolic bisimulation.

Definition 4.4 A *BExp*-indexed family of symmetric relations $\mathbf{S} = \{ S^b \mid b \in BExp \}$ is a *late strong symbolic bisimulation* if it satisfies: $(t, u) \in S^b$ implies that

> if $t \xrightarrow{b_1,\alpha} t'$ with $bv(\alpha) \cap fv(b,t,u) = \emptyset$, then there is a $b \wedge b_1$-partition B with $fv(B) \subseteq fv(b,t,u)$ such that for each $b' \in B$ there exist b_2, α' and u' with $b' \models b_2$, $\alpha =^{b'} \alpha'$, $u \xrightarrow{b_2,\alpha'} u'$ and $(t', u') \in S^{b'}$.

We write $t \sim_{\mathrm{L}}^b u$ if $(t, u) \in S^b \in \mathbf{S}$ for some late symbolic bisimulation $\mathbf{S}$. □

It is important to note that we now require $fv(B) \subseteq fv(b,t,u)$ (and this is actually the only difference between the definitions of the late and the early symbolic bisimulation relations). This implies that, when $\alpha \equiv c?x$, it is guaranteed that $x \notin fv(B)$. Therefore case analysis on the value space associated with x is *not* allowed before input actions are compared for bisimulation. As a consequence, an input transition from one process has to be matched by a *single* such transition from the other. This reflects the "late instantiation" strategy, and makes all the differences between late and early bisimulations!

As an illustration let us consider Example 3.7 again:

$$P \Leftarrow (c?x.\mathtt{if}\ x = 0\ \mathtt{then}\ a_1!x\ \mathtt{else}\ a_2!x) + c?x.a_3!x$$
$$Q \Leftarrow (c?x.\mathtt{if}\ x = 0\ \mathtt{then}\ a_3!x\ \mathtt{else}\ a_2!x) + (c?x.\mathtt{if}\ x = 0\ \mathtt{then}\ a_1!x\ \mathtt{else}\ a_3!x)$$

Now it is not allowed for Q to match the move $P \xrightarrow{true,c?x} a_3!x$, as any partition that permits this move to be matched by Q inevitably involves x which is not allowed according to the new definition.

As in the early case, late strong symbolic bisimulation captures late strong bisimulation in the following sense:

Theorem 4.5 *$t \sim_{\mathrm{L}}^b u$ if and only if $t\rho \sim_1 u\rho$ for every evaluation ρ such that $\rho \models b$.*

Proof: See [HL95, Theorem 4.5]. □

In Definition 4.4 we note that, whenever $t \sim_{\mathrm{L}}^b u$ holds, each symbolic transition from t may induce a different partition of b. Similarly, different transitions from u may induce different partitions of b. Sometimes it is more convenient to combine these partitions into a fine grain one which works uniformly for all symbolic transitions from both terms (as in the proof of Proposition 5.19). This is possible because there are only finitely many symbolic transitions from each term, as shown in the following lemma.

Lemma 4.6 *Assume that* $t \equiv \sum_I \alpha_i.t_i$ *and* $u \equiv \sum_J \beta_j.u_j$ *where* $bv(\alpha_i) \cap bv(\beta_j) \cap fv(b,t,u) = \emptyset$. *Then* $t \sim^b_{\mathtt{L}} u$ *if and only if there exists a disjoint b-partition* B *with* $fv(B) \subseteq fv(b,t,u)$ *such that for each* $b' \in B$ *the following holds:*

- *For each* $i \in I$ *there is a* $j \in J$ *such that* $\alpha_i =^{b'} \beta_j$ *and* $t_i \sim^{b'}_{\mathtt{L}} u_j$.
- *For each* $j \in J$ *there is an* $i \in I$ *such that* $\alpha_i =^{b'} \beta_j$ *and* $t_i \sim^{b'}_{\mathtt{L}} u_j$.

Proof: The "if" part of the proof is trivial and we only consider the "only if" direction here. Since $t \sim^b_{\mathtt{L}} u$, for each $i \in I$ there exists a b-partition B_i with $fv(B_i) \subseteq fv(t,\ u)$ such that for each $b_i \in B_i$ there exists $j \in J$ with $\alpha_i =^{b_i} \beta_j$ and $t_i \sim^{b_i}_{\mathtt{L}} u_j$; similarly for each $j \in J$ there exists a b-partition B'_j with $fv(B'_j) \subseteq fv(b,t,u)$ such that for each $b'_j \in B'_j$, there exists $i \in I$ with $\alpha_i =^{b'_j} \beta_j$ and $t_i \sim^{b'_j}_{\mathtt{L}} u_j$.

Let D_I denote the set of booleans $\{ \wedge_{i \in I} b_i \mid b_i \in B_i \}$, D_J the set $\{ \wedge_{j \in J} b'_j \mid b'_j \in B_j \}$ and set $B = \{ b_1 \wedge b_2 \mid b_1 \in D_I,\ b_2 \in D_J \}$. Then $\bigvee B = b$, $fv(B) \subseteq fv(b,t,u)$ and each $b' \in B$ has the form $(\wedge_i b_i) \wedge (\wedge_j b'_j)$ with $b_i \in B_i,\ b'_j \in B'_j$. For each $i \in I$, $b' \models b_i$ for some $b_i \in B_i$ and therefore there is a $j \in J$ such that $\alpha_i =^{b'} \beta_j$ and $t_i \sim^{b'}_{\mathtt{L}} u_j$. For each $j \in J$ $b' \models b'_j$ for some $b'_j \in B'_j$ and thus there is an $i \in I$ such that $\alpha_i =^{b'} \beta_j$ and $t_i \sim^{b'}_{\mathtt{L}} u_j$. $\Box$

This lemma holds for early symbolic bisimulation as well.

4.4 Weak Symbolic Equivalences

Our next task is to define the symbolic counterparts of the weak equivalences we have defined in Section 3 and study their relationships. Unlike in the strong case (but like in the concrete semantics) we need slightly different versions of the weak transition relations to define each of the early and late versions of weak symbolic bisimulation. Therefore we index the early weak transition relations with $\mathtt{E}$ and its late counterpart with $\mathtt{L}$.

The early weak symbolic transition relations are defined as the least relations over $\mathcal{T}$ which satisfy:

- $t \overset{true,\varepsilon}{\Longrightarrow}_{\mathtt{E}} t$
- $t \xrightarrow{b,\alpha} u$ implies $t \overset{b,\alpha}{\Longrightarrow}_{\mathtt{E}} u$
- $t \xrightarrow{b,\tau} \overset{b',\alpha}{\Longrightarrow}_{\mathtt{E}} u$, where $bv(\alpha) \cap fv(b) = \emptyset$, implies $t \overset{b \wedge b',\alpha}{\Longrightarrow}_{\mathtt{E}} u$
- $t \overset{b,\alpha}{\Longrightarrow}_{\mathtt{E}} \xrightarrow{b',\tau} u$ implies $t \overset{b \wedge b',\alpha}{\Longrightarrow}_{\mathtt{E}} u$

Concerning bound variables we now have:

Lemma 4.7 *If* $t \overset{b,c?x}{\Longrightarrow}_{\mathtt{E}} t'$ *then* $fv(b) \subseteq fv(t) \cup \{x\}$ *and* $x \notin fv(t)$. $\Box$

We have the following relationship between the concrete and the symbolic early weak transitions, similar to the strong case.

Lemma 4.8 *For all* t, u, b *and* ρ,

1. $t\rho \overset{\varepsilon}{\Longrightarrow}_{\mathtt{e}} p$ *if and only if there exist* b, t' *such that* $\rho \models b$, $p \equiv_\alpha t'\rho$ *and* $t \overset{b,\varepsilon}{\Longrightarrow}_{\mathtt{E}} t'$.

2. $t\rho \overset{\tau}{\Longrightarrow}_{\mathtt{e}} p$ *if and only if there exist* b, t' *such that* $\rho \models b$, $p \equiv_\alpha t'\rho$ *and* $t \overset{b,\tau}{\Longrightarrow}_{\mathtt{E}} t'$.

3. $t\rho \overset{c!v}{\Longrightarrow}_{\mathtt{e}} p$ *if and only if there exist* b, e, t' *such that* $\rho \models b, [\![e]\!]\rho = v$, $p \equiv_\alpha t'\rho$ *and* $t \overset{b,c!e}{\Longrightarrow}_{\mathtt{E}} t'$.

4. $t\rho \overset{c?v}{\Longrightarrow}_{\mathtt{e}} p$ *if and only if there exist* b, t' *and* $x \notin fv(t)$ *such that* $fv(b) \subseteq fv(t) \cup \{x\}$, $\rho\{v/x\} \models b$, $p \equiv_\alpha t'\rho\{v/x\}$ *and* $t \overset{b,c?x}{\Longrightarrow}_{\mathtt{E}} t'$.

Proof: Follows by induction on the definitions of the weak transitions. See [HL96, Lemmas 4.3-4.6] for the details. □

Definition 4.9 A *BExp*-indexed family of symmetric relations, $\mathbf{S} = \{ S^b \mid b \in BExp \}$, over $\mathcal{T}$ is an *early weak symbolic bisimulation* if it satisfies: $(t, u) \in S^b$ implies that

> if $t \overset{b_1,\alpha}{\longrightarrow} t'$ with $bv(\alpha) \cap fv(b, t, u) = \emptyset$, then there is a $b \wedge b_1$-partition B such that for each $b' \in B$ there exist b_2, α' and u' with $b' \models b_2$, $\alpha =^{b'} \alpha'$, $u \overset{b_2,\hat{\alpha}'}{\Longrightarrow}_{\mathtt{E}} u'$ and $(t', u') \in S^{b'}$.

We write $t \approx^b_{\mathtt{E}} u$ if $(t, u) \in S^b \in \mathbf{S}$ for some early weak symbolic bisimulation $\mathbf{S}$. □

Again we have to modify $\approx_{\mathtt{E}}$ so that it is preserved by $+$.

Definition 4.10 For $t, u \in \mathcal{T}$, $t \simeq^b_{\mathtt{E}} u$, if and only if

> whenever $t \overset{b_1,\alpha}{\longrightarrow} t'$ with $bv(\alpha) \cap fv(b, t, u) = \emptyset$, then there is a $b \wedge b_1$-partition B such that for each $b' \in B$ there exist b_2, α' and u' with $b' \models b_2$, $\alpha =^{b'} \alpha'$, $u \overset{b_2,\alpha'}{\Longrightarrow}_{\mathtt{E}} u'$ and $t' \approx^{b'}_{\mathtt{E}} u'$

and symmetrically for u. □

We refer to $\simeq^b_{\mathtt{E}}$ as *early symbolic observational congruence with respect to b*.

Note that in the definition above it is still essential to use partitions when matching moves. For example

$$\tau.p \simeq^{true}_{\mathtt{E}} \texttt{if } b \texttt{ then } \tau.p \texttt{ else } \tau.\tau.p$$

but the symbolic move $\tau.p \overset{true,\tau}{\longrightarrow} p$ of the left hand side can not be matched properly by a single symbolic move from the right hand side.

The two versions of early weak bisimulation equivalence/congruence can be related as in the case of strong bisimulation.

Theorem 4.11 1. $t \approx^b_{\mathtt{E}} u$ *if and only if* $t\rho \approx_{\mathtt{e}} u\rho$ *for every* ρ *such that* $\rho \models b$.

2. $t \simeq^b_{\mathtt{E}} u$ *if and only if* $t\rho \simeq_{\mathtt{e}} u\rho$ *for every* ρ *such that* $\rho \models b$.

Proof: See [HL96, Theorem 4.2]. $\square$

In the rest of this section we introduce the late version of weak symbolic bisimulation. Again, care must be taken when defining the corresponding weak symbolic transition relations and, like in the definition of $\Longrightarrow_{\mathtt{I}}$, we should not allow input moves to absorb τ moves that occur after them.

The late weak symbolic transition relations are defined as the least binary relations over $\mathcal{T}$ satisfying the following rules.

- $t \stackrel{true,\varepsilon}{\Longrightarrow}_{\mathtt{L}} t$
- $t \stackrel{b,\alpha}{\longrightarrow} u$ implies $t \stackrel{b,\alpha}{\Longrightarrow}_{\mathtt{L}} u$
- $t \stackrel{b,\tau}{\longrightarrow}\stackrel{b',\alpha}{\Longrightarrow}_{\mathtt{L}} u, bv(\alpha) \cap fv(b) = \emptyset,$ implies $t \stackrel{b \wedge b',\alpha}{\Longrightarrow}_{\mathtt{L}} u$
- if α is not of the form $c?x$ then $t \stackrel{b,\alpha}{\Longrightarrow}_{\mathtt{L}}\stackrel{b',\tau}{\longrightarrow} u$ implies $t \stackrel{b \wedge b',\alpha}{\Longrightarrow}_{\mathtt{L}} u$

The late counterpart of Lemma 4.7 is stated as follows:

Lemma 4.12 *If* $t \stackrel{b,c?x}{\Longrightarrow}_{\mathtt{L}} t'$ *then* $fv(b) \subseteq fv(t)$ *and* $x \notin fv(t)$.

Definition 4.13 A *BExp*-indexed family of symmetric relations $\mathbf{S} = \{\, S^b \mid b \in BExp \}$ is a *late weak symbolic bisimulation* if it satisfies: $(t,u) \in S^b$ implies that

> if $t \stackrel{b_1,\alpha}{\longrightarrow} t'$ with $bv(\alpha) \cap fv(b,t,u) = \emptyset$, then there is a $b \wedge b_1$-partition B with $fv(B) \subseteq fv(b,t,u)$ such that for each $b' \in B$ there exist b_2, α' and u' with $b' \models b_2$, $\alpha =^{b'} \alpha'$, $u \stackrel{b_2,\hat{\alpha}'}{\Longrightarrow}_{\mathtt{L}} u'$ and such that
>
> - if $\alpha \equiv c?x$ then there is a b'-partition B' such that for each $b'' \in B'$ there are b_2' and u'' with $b'' \models b_2'$, $u' \stackrel{b_2',\varepsilon}{\Longrightarrow}_{\mathtt{L}} u''$ and $(t', u'') \in S^{b''}$
> - otherwise $(t', u') \in S^{b'}$

We write $t \approx^b_{\mathtt{E}} u$ if $(t,u) \in S^b \in \mathbf{S}$ for some weak late symbolic bisimulation $\mathbf{S}$. $\square$

In the above definition another partition B', in which the input variable x may occur free, is needed after two input transitions are matched, and for each b'' in B' (symbolic) τ-moves after the matching input transition are abstracted away. This concurs with the input clause in the definition of late weak concrete bisimulation where τ-moves following the matching input transition get absorbed only after the input variable is instantiated.

As usual late symbolic observation congruence is defined in terms of late weak symbolic bisimulation:

Definition 4.14 For $t, u \in \mathcal{T}$, $t \simeq^b_{\mathtt{L}} u$, if and only if

> whenever $t \stackrel{b_1,\alpha}{\longrightarrow} t'$ with $bv(\alpha) \cap fv(b,t,u) = \emptyset$, then there is a $b \wedge b_1$-partition B with $fv(B) \subseteq fv(b,t,u)$ such that for each $b' \in B$ there exist b_2, α' and u' with $b' \models b_2$, $\alpha =^{b'} \alpha'$, $u \stackrel{b_2,\alpha'}{\Longrightarrow}_{\mathtt{L}} u'$ and such that

- if $\alpha \equiv c?x$ then there is a b'-partition B' such that for each $b'' \in B'$ there are b'_2 and u'' such that $b'' \models b'_2$, $u' \overset{b'_2, \varepsilon}{\Longrightarrow}_{\mathtt{L}} u''$ and $t' \approx^{b''}_{\mathtt{L}} u''$
- otherwise $t' \approx^{b'}_{\mathtt{L}} u'$

and symmetrically for u. □

We refer to $\simeq^b_{\mathtt{L}}$ as *late symbolic observational congruence with respect to b.*

We have the late counterpart of Theorem 4.11:

Theorem 4.15 *1. $t \approx^b_{\mathtt{L}} u$ if and only if $t\rho \approx_1 u\rho$ for every ρ such that $\rho \models b$.*

2. $t \simeq^b_{\mathtt{L}} u$ if and only if $t\rho \simeq_1 u\rho$ for every ρ such that $\rho \models b$.

Proof: See [HL96, Theorem 5.1]. □

5 Proof Systems

This section is devoted to developing proof systems for the bisimulation-based equivalences over various fragments of our value-passing language. We will consider finite, i.e. recursion-free, processes first and also postpone the parallel composition, renaming and restriction operators to a later section. For technical convenience we start with late bisimulation.

5.1 The Core System

A neat equational axiomatization for strong bisimulation was presented in [HM85], as shown in Figure 7. This set of axioms is sound and complete for recursion-free pure CCS with **0**, prefix and choice operators, in the sense that two finite processes are strongly bisimilar if and only if they can be proven equal from these axioms using equational reasoning. To generalize it to value-passing processes a key issue is how to deal with input prefixing, because this is exactly where data variables are introduced. A possible solution is to add the following rule:

$$\omega\text{−INPUT}\ \frac{t[v/x] = u[v/y] \quad \text{for each } v \in \mathit{Val}}{c?x.t = c?y.u}$$

In fact it can be shown that the resulting proof system is sound and complete for early strong bisimulation over recursion-free value-passing processes. However the problem here is that to apply the ω−INPUT rule we have to inspect an infinite number of premises when *Val* is infinite, and therefore this rule is of little practical use.

To obtain an effective proof system we need to replace the rule ω−INPUT with a finitary one. As the infinity in the rule is caused by instantiating input variables with all possible values from the data domain, to avoid such infinity we should be prepared to deal with open terms. Inspired by the work on symbolic bisimulations, which are indexed by boolean expressions, instead of pure equations we employ *conditional equations* of the

$$
\begin{array}{ll}
\mathrm{S1} & X + \mathbf{0} = X \\
\mathrm{S2} & X + X = X \\
\mathrm{S3} & X + Y = Y + X \\
\mathrm{S4} & (X + Y) + Z = X + (Y + Z)
\end{array}
$$

Figure 7: The Axioms For Strong Bisimulation

form $b \rhd t = u$, where the condition b is a boolean expression, as judgments of our proof system. The intended meaning of $b \rhd t = u$ is that "$t \sim^b u$", or "t and u are bisimilar when the variables that occur free in these terms are instantiated with values satisfying b". In this setting ordinary equations are regarded as special conditional equations with the condition being *true*, i.e. $t = u$ is taken to be an abbreviation for $true \rhd t = u$.

Reasoning about value-passing processes will inevitably involve reasoning about data. However, instead of inventing proof rules for all possible data domains, we would like to factor out reasoning about data from reasoning about processes as much as possible. Therefore our proof system will be parameterized over data reasoning of the form $b \models b'$, with the intuitive meaning that whenever b is true then so is b'. As a consequence, all the completeness results in the following sections are relative to data domain reasoning. The proposed inference rules are listed in Figure 8. Before starting to investigate properties of this proof system, some informal discussion about these rules are in order.

The set of inference rules we put forward can be seen as a natural generalization of pure equational reasoning. For each construct in our language there is a corresponding introduction rule. The OUTPUT rule refers to reasoning about data domains by including $b \models e = e'$ in the premise. It says that if we can deduce a statement $b \rhd t = u$ from this proof system, and we know that b implies $e = e'$ from the knowledge about data, then we can conclude that $b \rhd c!e.t = c!e'.u$. The INPUT rule requires a side condition which guarantees the input variable to be "fresh". This is crucial to the soundness of the rule. Without this proviso we could, for instance, derive

$$x = 1 \rhd c?x.c!1 = c?x.c!x$$

from

$$x = 1 \rhd c!1 = c!x$$

The premise is true but the conclusion is false. With the side condition this derivation is not possible because x is free in the condition $x = 1$. The introduction rule for the conditional construct is COND which performs a case analysis; an identity of the form `if` b `then` $t = u$ can be established by considering two cases, one when b is true and the other when it is false. Finally there are two "structural rules", PARTITION and ABSURD, which do not deal with any specific constructs in the language, but are useful to "glue" pieces of derivations together. The PARTITION rule allows partitioning of the current value space, represented by b, into two subspaces, represented by b_1 and b_2, respectively. Like OUTPUT, this rule also requires the establishment of facts about the data domain. Taking $b_1 \equiv b_2 \equiv b'$ it specializes to a useful rule

$$\text{CONSEQUENCE} \quad \frac{b \models b', \; b' \rhd t = u}{b \rhd t = u}$$

EQUIV $\dfrac{}{t = t}$ $\dfrac{t = u}{u = t}$ $\dfrac{t = u \quad u = v}{t = v}$

α-CONV $\dfrac{}{c?x.t = c?y.t[y/x]}$ $y \notin fv(t)$

AXIOM $\dfrac{}{true \rhd t = u}$ $t = u$ is an axiom instance

CHOICE $\dfrac{b \rhd t = t'}{b \rhd t + u = t' + u}$

INPUT $\dfrac{b \rhd t = u}{b \rhd c?x.t = c?x.u}$ $x \notin fv(b)$

OUTPUT $\dfrac{b \models e = e', \quad b \rhd t = u}{b \rhd c!e.t = c!e'.u}$

TAU $\dfrac{b \rhd t = u}{b \rhd \tau.t = \tau.u}$

COND $\dfrac{b \wedge b' \rhd t = u \quad b \wedge \neg b' \rhd \mathbf{0} = u}{b \rhd \texttt{if } b' \texttt{ then } t = u}$

PARTITION $\dfrac{b \models b_1 \vee b_2, \quad b_1 \rhd t = u \quad b_2 \rhd t = u}{b \rhd t = u}$

ABSURD $\dfrac{}{false \rhd t = u}$

Figure 8: The Inference Rules

We write $\vdash b \triangleright t = u$ to mean that $b \triangleright t = u$ can be derived from axioms S1–S4 using the rules in Figure 8. With a slight abuse of notation we will also use $\vdash$ to denote this set of rules and axioms.

The soundness of $\vdash$ is stated below:

Theorem 5.1 *If $\vdash b \triangleright t = u$ then $t\rho \sim_l u\rho$ for any ρ such that $\rho \models b$.*

Proof: The proof follows by induction on the derivation of $b \triangleright t = u$ and a case analysis on the last rule used. □

Before embarking on the task of proving the completeness result, we first establish some useful facts about the proof system.

Proposition 5.2 *1. If $b \models \neg b'$ then $\vdash b \triangleright \texttt{if } b' \texttt{ then } t = \mathbf{0}$*

2. *if $b \models b'$ then $\vdash b \triangleright t = \texttt{if } b' \texttt{ then } t$*
3. $\vdash \texttt{if } b \texttt{ then } (\texttt{if } b' \texttt{ then } t) = \texttt{if } b \wedge b' \texttt{ then } t$
4. $\vdash t = t + \texttt{if } b \texttt{ then } t$
5. $\vdash b \wedge b' \triangleright t = u$ *implies* $\vdash b \triangleright \texttt{if } b' \texttt{ then } t = \texttt{if } b' \texttt{ then } u$
6. $\vdash \texttt{if } b \texttt{ then } (t + u) = \texttt{if } b \texttt{ then } t \; + \; \texttt{if } b \texttt{ then } u$
7. $\vdash \texttt{if } b \texttt{ then } u \; + \; \texttt{if } b' \texttt{ then } u = \texttt{if } b \vee b' \texttt{ then } u$
8. *if $fv(b) \cap bv(\alpha) = \emptyset$ then* $\vdash \texttt{if } b \texttt{ then } \alpha.t = \texttt{if } b \texttt{ then } \alpha.(\texttt{if } b \texttt{ then } t)$

Proof: We only give the details of the proof for three of these statements.

1. An application of COND reduces the goal to

$$b \wedge b' \triangleright t = \mathbf{0} \;\text{ and }\; b \wedge \neg b' \triangleright \mathbf{0} = \mathbf{0}$$

The first subgoal can be settled by CONSEQUENCE and ABSURD, while the second follows by EQUIV.

4. Because $true \models b \vee \neg b$, by the PARTITION rule it is sufficient to prove the two statements

$$\vdash b \triangleright t = t + \texttt{if } b \texttt{ then } t \;\text{ and }\; \vdash \neg b \triangleright t = t + \texttt{if } b \texttt{ then } t. \tag{1}$$

By S2 and CHOICE the first part of (1) can be reduced to $\vdash b \triangleright t = \texttt{if } b \texttt{ then } t$. By COND this can be further reduced to

$$\vdash b \wedge b \triangleright t = t \;\text{ and }\; b \wedge \neg b \triangleright t = \mathbf{0}$$

which are simple consequences of EQUIV and ABSURD, respectively.

The second part of (1) can be derived from S1 and Proposition 5.2.1.

8 . Two applications of COND and ABSURD reduce this to

$$\vdash b \rhd \alpha.t = \alpha.(\texttt{if } b \texttt{ then } t)$$

The proof then proceeds by a case analysis on the form of α. For example if it is $c?x$, by assumption $x \notin fv(b)$ and therefore, by INPUT, it can be reduced to $\vdash b \rhd t = \texttt{if } b \texttt{ then } t$, which follows from Proposition 5.2.2. If α is τ or $c!e$ the reasoning is even more straightforward.

□

This proof system is complete for late strong bisimulation over the recursion-free subset of our value-passing language, as stated in the following theorem.

Theorem 5.3 *(Completeness of* $\vdash$*)* $t \sim^b_{\mathrm{L}} u$ *implies* $\vdash b \rhd t = u$.

As in the case of pure CCS, the proof of this theorem proceeds by induction on the heights of terms and relies on the notion of a normal form.

Definition 5.4 The height of t, denoted $\mid t \mid$, is defined inductively as follows:

1. $\mid \mathbf{0} \mid = 0$
2. $\mid t + u \mid = \ max\{\mid t \mid, \mid u \mid\}$
3. $\mid \texttt{if } b \texttt{ then } t \mid = \mid t \mid$
4. $\mid \alpha.t \mid = 1 + \mid t \mid$

□

Definition 5.5 A process term t is a *normal form*, or is *in normal form*, if it has the form $\sum_{i \in I} \texttt{if } b_i \texttt{ then } \alpha_i.t_i$ where each t_i is in normal form (by convention the sum above is taken to be $\mathbf{0}$ when the index set I is empty). □

Lemma 5.6 *For every term t there exists a normal form* t' *such that* $fv(t) = fv(t')$, $\mid t \mid = \mid t' \mid$ *and* $\vdash t = t'$.

Proof: By structural induction using the elementary facts about the proof system given in Proposition 5.2. □

Proof of Theorem 5.3
By Lemma 5.6 we may assume that t and u are in normal form:

$$t \equiv \sum_{i \in I} \texttt{if } c_i \texttt{ then } \alpha_i.t_i \qquad u \equiv \sum_{j \in J} \texttt{if } d_j \texttt{ then } \beta_j.u_j$$

The proof proceeds by induction on $|\, t \,| + |\, u \,|$ where the base case follows immediately. For the inductive step we first show that

$$\vdash b \wedge c_i \rhd u + \texttt{if } c_i \texttt{ then } \alpha_i.t_i = u \tag{2}$$

for each $i \in I$. For the proof of (2) we only examine the case $\alpha_i \equiv c?x$ (the remaining cases are similar). By α-conversion we may assume $x \notin fv(b, t, u)$. Since $t \sim_{\mathtt{L}}^{b} u$ and $t \xrightarrow{c_i, c?x} t_i$, there exists a $b \wedge c_i$-partition B such that $fv(B) \subseteq fv(b, t, u)$ and for each $b' \in B$ there is a $j \in J$ such that $b' \models d_j$, $u \xrightarrow{d_j, c?x} u_j$ and $t_i \sim_{\mathtt{L}}^{b'} u_j$. By induction

$$\vdash b' \rhd t_i = u_j$$

Since $x \notin fv(b')$, applying INPUT we obtain

$$\vdash b' \rhd c?x.t_i = c?x.u_j$$

Since $b' \models c_i$ and $b' \models d_j$, by Proposition 5.2.2 and transitivity

$$\vdash b' \rhd \texttt{if } c_i \texttt{ then } c?x.t_i = \texttt{if } d_j \texttt{ then } c?x.u_j$$

Using S2 we derive that

$$\vdash b' \rhd u + \texttt{if } c_i \texttt{ then } c?x.t_i = u$$

Since $\bigvee B = b \wedge c_i$, by a repeated use of PARTITION

$$\vdash b \wedge c_i \rhd u + \texttt{if } c_i \texttt{ then } c?x.t_i = u$$

This is the required (2) above.
Now we proceed with the completeness proof as follows. By Proposition 5.2.*1*

$$\vdash b \wedge \neg c_i \rhd \texttt{if } c_i \texttt{ then } \alpha_i.t_i = \mathbf{0}$$

and thus, by CHOICE and S1

$$\vdash b \wedge \neg c_i \rhd u + \texttt{if } c_i \texttt{ then } \alpha_i.t_i = u \tag{3}$$

By (2), (3) and PARTITION

$$\vdash b \rhd u + \texttt{if } c_i \texttt{ then } \alpha_i.t_i = u$$

This is true for each $i \in I$. By repeated use of this property we get

$$\vdash b \rhd u + t = u$$

Symmetrically we can derive

$$\vdash b \rhd u + t = t$$

and the result follows by transitivity. $\square$

5.2 The Axioms for Early Bisimulation

As explained in Section 4.2, early bisimulation differs from late bisimulation only in that, when matching an input transition from one process, it is possible to use input transitions from different components of the other for different input values. Reflected at the syntactical level, to obtain a proof system for early bisimulation from the late one, what is needed is a rule which allows us to "break" the scopes of input prefixes at the same level. This can be achieved in two ways: either by adding a rule schema or an axiom schema to the proof system for late bisimulation. The axiom schema for early bisimulation was first proposed in [PS93] for an axiomatization of the π-calculus, and is adapted for value-passing processes in [HL96]. The rule schema was used in [HL96]. Here we first include into our system the axiom schema which has the following form:

$$\text{EA} \quad c?x.t + c?x.u = c?x.t + c?x.u + c?x.(\texttt{if } b \texttt{ then } t \ + \ \texttt{if } \neg b \texttt{ then } u)$$

Note that here b is an arbitrary boolean expression. It is not difficult to see that the proof system is sound with respect to $\sim^b_{\mathtt{E}}$. Let us write $\vdash_{\mathtt{E}} b \rhd t = u$ to mean that $b \rhd t = u$ can be deduced from the proof system of the previous section extended with EA. EA can be generalized to allow arbitrary summations as follows.

Proposition 5.7 *For any finite set of booleans $\{ b_i \mid 1 \leq i \leq n \}$ such that $\bigvee_{1\leq i\leq n} b_i = b$ and $b_i \wedge b_j = false$ for $i \neq j$,*

$$\vdash_{\mathtt{E}} b \rhd \sum_{1\leq i\leq n} c?x.t_i = \sum_{1\leq i\leq n} c?x.t_i + c?x. \sum_{1\leq i\leq n} \texttt{if } b_i \texttt{ then } t_i$$

Proof: See [HL96, Proposition 3.1]. □

Now we have the following counterpart of Theorem 5.3.

Theorem 5.8 *(Completeness of $\vdash_{\mathtt{E}}$) $t \sim^b_{\mathtt{E}} u$ implies $\vdash_{\mathtt{E}} b \rhd t = u$.*

Proof: The proof follows the same pattern as the proof of Theorem 5.3, using Proposition 5.7. See [HL96, Theorem 3.2] for the details. □

Another way of dealing with the early case is to replace EA by the following rule schema

$$\text{E-INPUT} \quad \frac{b \rhd \sum_{i\in I} \tau.t_i = \sum_{j \in J} \tau.u_j}{b \rhd \sum_{i\in I} c?x.t_i = \sum_{j\in J} c?x.u_j} \quad x \notin fv(b)$$

This also is quite general but at least its application only depends on the structure of the terms in the proof being elaborated.

Note that the use of τ in this rule is essential. If τ is omitted then the rule becomes

$$\frac{b \rhd \sum_{i\in I} t_i = \sum_{j \in J} u_j}{b \rhd \sum_{i\in I} c?x.t_i = \sum_{j\in J} c?x.u_j} \quad x \notin fv(b)$$

$$\begin{array}{ll} \text{T1} & \alpha.\tau.X = \alpha.X \\ \text{T2} & X + \tau.X = \tau.X \\ \text{T3} & \alpha.(X + \tau.Y) + \alpha.Y = \alpha.(X + \tau.Y) \end{array}$$

Figure 9: The τ Laws

which is not sound with respect to early strong bisimulation. For example since $(p + q) + r = p + (q + r)$ we could use this rule to derive

$$c?x.(p + q) + c?x.r = c?x.p + c?x.(q + r)$$

for which there are obvious counterexamples.

With the help of COND and PARTITION, EA can be easily derived from E-INPUT. Hence the proof system obtained by replacing the axiom schema EA with E-INPUT is also complete.

5.3 The τ Laws

One of the classical results in the theory of bisimulation is that, adding the three τ laws, presented in Figure 9, to the complete axiomatization of strong bisimulation results in a complete axiomatization of observational congruence. The aim of this section is to show that this result carries over to value-passing processes. We start with the late equivalence. Let us write $\vdash_{\mathtt{wL}} b \triangleright t = u$ to mean that $b \triangleright t = u$ can be deduced from the core system extended with the τ laws.

The structure of the completeness proof below follows closely a corresponding proof in pure CCS [Mil89]. As in that reference, the proof relies on a notion of *full normal forms* which are τ-saturated. But again here we need to be careful with the τ transitions occurring after an input action; they should not be absorbed. These τ actions will be dealt with in the proof of the main theorem.

Definition 5.9 A normal form $t \equiv \sum_{i \in I} \mathtt{if}\ b_i\ \mathtt{then}\ \alpha_i.t_i$ is a *late full normal form* if

$$t \overset{b,\alpha}{\Longrightarrow}_{\mathtt{L}} t' \text{ implies } t \overset{b,\alpha}{\longrightarrow} t' \text{ and each } t_i \text{ is in late full normal form.}$$

□

To show that every term can be transformed into a late full normal form we need a lemma permitting the absorption of τ actions by syntactic manipulation. This lemma requires a generalization of T3:

Lemma 5.10 *If $fv(b) \cap bv(\alpha) = \emptyset$ then* $\vdash_{\mathtt{wL}} \alpha.(t + \mathtt{if}\ b\ \mathtt{then}\ \tau.u) = \alpha.(t + \mathtt{if}\ b\ \mathtt{then}\ \tau.u) + \mathtt{if}\ b\ \mathtt{then}\ \alpha.u$

Proof: See [HL96, Lemma 4.7]. □

Lemma 5.11 *(Absorption) If* $t \overset{b,\alpha}{\Longrightarrow}_{\mathtt{L}} t'$*, with* $fv(b) \cap bv(\alpha) = \emptyset$*, then* $\vdash_{\mathtt{wL}} t = t + \mathtt{if}\ b\ \mathtt{then}\ \alpha.t'$*.*

Proof: See [HL96, Lemma 4.8]. □

With the absorption lemma it can be easily shown that every term is equivalent to a late full normal form:

Lemma 5.12 *For any normal form* t *there is a full normal form* t' *such that* $fv(t) = fv(t'), \mid t \mid = \mid t' \mid$ *and* $\vdash_{\mathtt{wL}} t = t'$*.*

By this lemma and Lemma 5.6, every term can be transformed into a full normal form of equal height.

We need one more ingredient for the completeness proof.

Proposition 5.13 $t \approx^b_{\mathtt{E}} u$ *if and only if there is a* b*-partition* B *such that for all* $b' \in B$*,* $t \simeq^{b'}_{\mathtt{E}} u$ *or* $t \simeq^{b'}_{\mathtt{E}} \tau.u$ *or* $\tau.t \simeq^{b'}_{\mathtt{E}} u$*.*

Proof: See [HL96, Proposition 4.2]. □

Theorem 5.14 *(Completeness of* $\vdash_{\mathtt{wL}}$*)* $t \simeq^b_{\mathtt{L}} u$ *implies* $\vdash_{\mathtt{wL}} b \rhd t = u$*.*

Proof: See [HL96, Theorem 5.3]. □

As in the case of strong equivalence, to obtain a complete proof system for early observation congruence, all we need is to add either the axiom schema EA or rule schema E-INPUT to the proof system. For the sake of variation here we use E-INPUT. So let us write $\vdash_{\mathtt{wE}} b \rhd t = u$ to mean that $b \rhd t = u$ can be deduced from the proof system for $\vdash_{\mathtt{wL}}$ plus E-INPUT. As can be expected, the proof of the completeness result for early congruence differs from that for late only in the case of input prefixing. The E-INPUT rule has a generalized form:

Proposition 5.15 *Suppose* $x \notin fv(b, c_i, d_j),\ i \in I, j \in J$*. Then from*

$$\vdash_{\mathtt{wE}} b \rhd \sum_{i \in I} \mathtt{if}\ c_i\ \mathtt{then}\ \tau.t_i = \sum_{j \in J} \mathtt{if}\ d_j\ \mathtt{then}\ \tau.u_j$$

we can infer

$$\vdash_{\mathtt{wE}} b \rhd \sum_{i \in I} \mathtt{if}\ c_i\ \mathtt{then}\ c?x.t_i = \sum_{j \in J} \mathtt{if}\ d_j\ \mathtt{then}\ c?x.u_j$$

Proof: See [HL96, Proposition 4.3]. □

Theorem 5.16 *(Completeness of* $\vdash_{\mathtt{wE}}$*)* $t \simeq^b_{\mathtt{E}} u$ *implies* $\vdash_{\mathtt{wE}} b \rhd t = u$*.*

Proof: See [HL96, Theorem 4.3]. □

$$
\begin{array}{ll}
\mathbf{0}\backslash L = \mathbf{0} & \mathbf{0}[f] = \mathbf{0} \\
(X+Y)\backslash L = X\backslash L + Y\backslash L & (X+Y)[f] = X[f] \ + Y[f] \\
(\alpha.X)\backslash L = \begin{cases} \alpha.(X\backslash L) & \text{if } \mathit{Chan}(\alpha) \notin L \\ \mathbf{0} & \text{if } \mathit{Chan}(\alpha) \in L \end{cases} & (\alpha.X)[f] = f(\alpha).X[f]
\end{array}
$$

Figure 10: Axioms for Restriction and Renaming

Let $t = \sum_i \alpha_i.t_i$ and $u = \sum_j \beta_j.u_j$, with $\mathit{fv}(t) \cap \mathit{bv}(u) = \mathit{fv}(u) \cap \mathit{bv}(t) = \emptyset$. Then

$$t \mid u = \mathit{sync\_move}(t,u) + \mathit{async\_move}(t,u)$$

where

$$
\begin{aligned}
\mathit{sync\_move}(t,u) = \quad & \sum \{\, \tau.(t_i[e/x] \mid u_j) \mid \alpha_i \equiv c?x, \beta_j \equiv c!e \,\} + \\
& \sum \{\, \tau.(t_i \mid u_j[e/x]) \mid \alpha_i \equiv c!e, \beta_j \equiv c?x \,\} \\
\mathit{async\_move}(t,u) = \quad & \sum_i \alpha_i.(t_i \mid u) + \sum_j \beta_j.(t \mid u_j)
\end{aligned}
$$

Figure 11: The Expansion Law

5.4 Adding Other Operators

So far we have concentrated on the core language which does not include the restriction, renaming and parallel composition operators. These operators are however easy to axiomatize. The equations characterizing the restriction and renaming operator are the same as for pure processes and are listed in Figure 10. These laws are all fairly standard and need no further explanation. The expansion law for the parallel operator, shown in Figure 11, is an adaption of the one proposed in [PS93] for the π-calculus.

We know from the previous sections that $\sim_{\mathtt{e}}$, $\simeq_{\mathtt{e}}$, $\sim_{\mathtt{l}}$ and $\simeq_{\mathtt{l}}$ are all preserved by the new operators. It is also easy to check that the new equations are sound for these congruence relations. Moreover, by adding these new equations to the core proof system, each term in the extended language can be transformed to a term in the core language. This in turn ensures the completeness of this extended proof system over the extended language with respect to each of the equivalences mentioned above.

5.5 Reasoning about Recursion

The proof systems considered so far are complete with respect to the corresponding equivalence relations over finite processes only. Processes exhibiting infinite behaviours are defined recursively. To be able to reason about infinite processes we need inference rules to handle recursion. For this purpose it is more convenient to have recursion terms in the language, instead of using process identifiers and defining equations. In pure CCS recursive terms can be constructed using process variables and the **fix** operator: if X is a process variable and t is a process term then $\mathbf{fix}Xt$ is a recursive term. However, in the value-passing calculus, a process variable with non-zero arity needs to take data parameters to form a process term. With the notation $\mathbf{fix}Xt$ it is not clear how free data

$$\textsc{fix}\ \frac{t[\mathbf{fix}X(\overline{x})t/X][\overline{v}/\overline{x}] \xrightarrow{a} t'}{(\mathbf{fix}X(\overline{x})t)\overline{e} \xrightarrow{a} t'} \text{ where } \overline{v} = [\![\overline{e}]\!] \qquad \textsc{Fix}\ \frac{t[\mathbf{fix}X(\overline{x})t/X][\overline{e}/\overline{x}] \xrightarrow{b,\alpha} t'}{(\mathbf{fix}X(\overline{x})t)\overline{e} \xrightarrow{b,\alpha} t'}$$

Figure 12: Operational Semantics for Recursion

variables in t are bound, and what are the free variables and parameters of this term. To avoid these confusions we will use *process abstractions* instead of process terms to form recursions.

Let *PVar* be a countably infinite set of *process abstraction variables* (or just *process variables* for short), ranged over by X, Y, Z etc. Each process abstraction variable is associated with a non-negative *arity* indicating the number of data expressions it should take to form a process term. We add a new kind of process term and a new clause for abstractions to the language introduced in Section 2. Also we shall omit parallel composition, restriction and renaming operators. The resulting language is known as *regular value-passing CCS* and has the following BNF grammar:

$$\begin{array}{rcl} t & ::= & \mathbf{0} \,\big|\, \alpha.t \,\big|\, t+t \,\big|\, \texttt{if } b \texttt{ then } t \,\big|\, F\overline{e} \\ F & ::= & X \,\big|\, (\overline{x})t \,\big|\, \mathbf{fix}XF \\ \alpha & ::= & \tau \,\big|\, c?x \,\big|\, c!e \end{array}$$

An abstraction is either a process variable X, a process abstraction $(\overline{x})t$ (where $\overline{x}$ is a vector of distinct names), or a recursion abstraction $\mathbf{fix}XF$. The *degree* of an abstraction is defined as follows: The degree of X is the arity of X; the degree of $(\overline{x})t$ is the length of $\overline{x}$; the degree of $\mathbf{fix}XF$ is that of X which is required to be equal to the degree of F. Abstractions can be applied to data expressions to form process terms, and in $F\overline{e}$ we always require the length of $\overline{e}$ to be equal to the degree of F.

In $(\overline{x})t$ each data variable in $\overline{x}$ is bound in t. In a recursively defined term, $\mathbf{fix}XF$, the process variable X is bound in F. As usual a variable which is not bound in t is called *free* in t. We will write $BV(t)$ and $FV(t)$ for the sets of bound and free process variables in t. A term is *data closed* if it contains no free data variables. A term is *process closed* if it contains no free process variables. A term is *closed* if it is both data and process closed. A term is *data open* if it is not data closed and it is *process open* if it is not process closed; it is *open* if it is either data open or process open. We will only use data closed abstractions, so in $(\overline{x})t$ it is always required that $fv(t) \subseteq \{\overline{x}\}$. A process variable X is *(weakly) guarded* in a process term or an abstraction if each free occurrence of X is within the scope of some prefix operator $\alpha.\_$.

Using data closed abstractions to form recursion terms has two main advantages: In a recursion term $(\mathbf{fix}X(\overline{x})t)\overline{e}$ the free data variables are exactly those of $\overline{e}$, and the passing of the actual data parameters $\overline{e}$ to free occurrences of the recursion variable X in t is taken care of by β-conversion. As a consequence the inference rules for recursion become very simple, as simple as those for pure calculi. For instance, the rule for recursion unfolding is simply $\mathbf{fix}XF = F[\mathbf{fix}XF/X]$. Below is an example demonstrating how

$$\text{REC}\ \frac{}{\mathbf{fix}XF = F[\mathbf{fix}XF/X]}$$

$$\text{CONGR}-\mathbf{fix}\ \frac{F = G}{\mathbf{fix}XF = \mathbf{fix}XG}$$

$$\text{UFI}\ \frac{G = F[G/X]}{G = \mathbf{fix}XF}\ X \text{ guarded in } F$$

$$\text{ABS}\ \frac{t = u}{(\overline{x})t = (\overline{x})u}$$

$$\text{APP}\ \frac{F = G}{F\overline{e} = G\overline{e}}$$

$$\text{BETA}\ \frac{}{((\overline{x})t)\overline{e} = t[\overline{e}/\overline{x}]}$$

Figure 13: Inference Rules for Recursion

this rule acts (where F stands for $(x,y)(c!(x+y).d?z.X(z,y)))$:

$$\begin{array}{cl} & (\mathbf{fix}XF)(u+1, v-1) \\ \overset{\text{REC}}{=} & ((x,y)(c!(x+y).d?z.(\mathbf{fix}XF)(z,y)))(u+1,v-1) \\ \overset{\text{BETA}}{=} & (c!(x+y).d?z.(\mathbf{fix}XF)(z,y))[u+1, v-1/x,y] \\ = & c!(u+1+v-1).d?z.(\mathbf{fix}XF)(z,v-1) \\ = & c!(u+v).d?z.(\mathbf{fix}XF)(z,v-1) \end{array}$$

The concrete and symbolic transitional semantics for recursion are given by the rules FIX and FIX, respectively, in Figure 12. The definitions of all the versions of bisimulations are the same as before, but since now we have a new syntactical category, namely abstractions, we need to extend the notions of bisimulation to it. We may wish to extend bisimulation to open process terms as well. This can be easily done by requiring that two open terms are bisimilar if and only if all their closed instantiations are, and two abstractions are bisimilar if and only if all their applications to data parameters are. To be precise, let $\sim$ be the generic symbol for different bisimulations. Then $t \sim u$ if and only if $t[\overline{p}/\overline{X}] \sim u[\overline{p}/\overline{X}]$, where $\overline{X} = FV(t,u)$, for any closed terms $\overline{p}$. $F \sim G$ if and only if $F\overline{e} \sim G\overline{e}$ for all vectors of values (in the case of concrete bisimulations) or data expressions (in the case of symbolic bisimulations) $\overline{e}$.

The proof rules for recursion are presented in Figure 13. REC is the standard rule for recursion unfolding and CONGR-**fix** expresses that all bisimulation equivalences we consider are congruent with respect to **fix**. Rules ABS and APP allow us to switch between abstractions and process terms. The most interesting rule is UFI, for Unique Fixpoint Induction. It asserts that every guarded abstraction has exactly one fixpoint. Syntactically it appears the same as its counterpart in the pure calculus [Mil84]. However, here it works at the level of abstractions, while in the pure case it works at the level of processes. The latter can be viewed as a special case of the former when the arity of the process abstraction involved is 0. Let us write $\vdash_{\mathtt{r}} b \rhd t = u$ (where t and u may also be abstractions) to mean that $b \rhd t = u$ can be derived using the core rules and axioms in Section 5.1 together with the rules in Figure 13. We state the soundness of $\vdash_{\mathtt{r}}$:

Theorem 5.17 *If* $\vdash_{\mathtt{r}} b \rhd t = u$ *then* $t\rho \sim_1 u\rho$ *for any* $\rho \models b$.

For the completeness result we shall confine ourselves to guarded terms. As we have seen in the previous sections, to obtain a completeness result, a key instrument is the notion of a normal form whose syntax directly reflects its symbolic transition semantics. The completeness proof can then be performed by induction on the "height" (the maximum nesting depth of action prefixes) of the normal forms. This works fine for finite terms. But since the language we are considering now includes recursion we can not rely on such simple notion of a normal form. Instead we use *standard equation sets.*

Let $\overline{X} = \{X_1, X_2, ..., X_m\}$ and $\overline{W} = \{W_1, W_2, ..., W_n\}$ be disjoint sets of process variables. Then

$$E : X_i(\overline{x}_i) = H_i \qquad 1 \leq i \leq m,$$

where $fv(H_i) \subseteq \{\overline{x}_i\}$ and $FV(H_i) \subseteq \overline{X} \cup \overline{W}$, is an equation set with formal process variables in $\overline{X}$ and free process variables in $\overline{W}$. E is *process closed* if $\overline{W} = \emptyset$. E is *standard* if each H_i has the form

$$\sum_{k \in K_i} \mathtt{if}\ b_{ik}\ \mathtt{then}\ \big(\sum_{p \in P_{ik}} \alpha_{ikp}.X_{f(i,k,p)}(\overline{e}_{ikp}) + \sum_{p' \in P'_{ik}} W_{f'(i,k,p')}(\overline{w}_{ikp'})\big)$$

In $X_{f(i,k,p)}$, the notation $f(i,k,p)$ is used to indicate that the subscript depends on i, k and p. An abstraction F provably satisfies equation set E if there are $\overline{F} = \{F_1, \ldots, F_n\}$ with $\vdash_{\mathtt{r}} F = F_1$ such that

$$\vdash_{\mathtt{r}} b_i \rhd F_i = (\overline{x}_i)H_i[\overline{F}/\overline{X}]$$

A process term t provably b-satisfies equation set E if there exist $\overline{b}$ and $\overline{t}$ with $fv(b_i) \cup fv(t_i) \subseteq \{\overline{x}_i\}$, $1 \leq i \leq m$, $b_1 = b$ and $\vdash_{\mathtt{r}} b_1 \rhd t_1 = t$, such that

$$\vdash_{\mathtt{r}} b_i \rhd t_i = H_i[(\overline{x}_k)(\mathtt{if}\ b_k\ \mathtt{then}\ t_k)/X_k | 1 \leq k \leq m] \qquad 1 \leq i \leq m$$

We will simply say that "t provably satisfies E" when $b \equiv true$.

Following [Mil84], the proof of the completeness theorem consists of three major steps: First we show that each guarded term provably satisfies a standard equation set; then we show that if two terms are symbolically bisimilar over b then a single guarded equation set can be constructed which is provably b-satisfied by both terms; finally we show that if two terms t and u both provably satisfy a guarded equation set over b then $\vdash_{\mathtt{r}} b \rhd t = u$.

Proposition 5.18 *Any guarded process term* t *provably satisfies a standard equation set* E*. Moreover, if* t *is process closed then so is* E*.*

Proof: By mutual induction on the structures of process terms and abstractions. □

Proposition 5.19 *Assume that t and u are guarded process terms. If $t \sim_{\mathtt{L}}^{b} u$ then there exists a standard equation set E such that both t and u provably b-satisfy E.*

Proof: For simplicity we assume t and u are process closed. It is straightforward to extend the proof to the case when they contain free process variables. Let E_1 and E_2 be the standard equation sets for t and u, respectively:

$$\begin{aligned} E_1 : X_i(\overline{x}_i) &= \sum_{k \in K_i} \texttt{if } c_{ik} \texttt{ then } \sum_{p \in P_{ik}} \alpha_{ikp}.X_{f(i,k,p)}(\overline{e}_{ikp}) \quad & i \in I \\ E_2 : Y_j(\overline{y}_j) &= \sum_{l \in L_j} \texttt{if } d_{jl} \texttt{ then } \sum_{q \in Q_{jl}} \beta_{jlq}.Y_{g(j,l,q)}(\overline{e}'_{jlq}) \quad & j \in J \end{aligned}$$

where $\overline{x}_i \cap \overline{y}_j = \emptyset$, and all input prefixes in E_1, E_2 use the same bound variable not appearing in E_1 or E_2.

Thus there exist t_i, u_j with $fv(t_i) \subseteq \overline{x}_i$, $fv(u_j) \subseteq \overline{y}_j$ such that

$$\vdash_{\mathtt{r}} t_i = \sum_{k \in K_i} \texttt{if } c_{ik} \texttt{ then } t_{ik} \tag{4}$$

$$\vdash_{\mathtt{r}} u_j = \sum_{l \in L_j} \texttt{if } d_{jl} \texttt{ then } u_{jl} \tag{5}$$

where $t_{ik} \equiv \sum_{p \in P_{ik}} \alpha_{ikp}.t_{f(i,k,p)}[\overline{e}_{ikp}/\overline{x}_{f(i,k,p)}]$, $u_{jl} \equiv \sum_{q \in Q_{jl}} \beta_{jlq}.u_{g(j,l,q)}[\overline{e}'_{jlq}/\overline{y}_{g(j,l,q)}]$.

For each pair $(i,\ j)$, let b_{ij} with $fv(b_{ij}) \subseteq \{\overline{x}_i, \overline{y}_j\}$ be such that, for any ρ,

$$\rho \models b_{ij} \text{ if and only if } t_i\rho \sim_l u_j\rho$$

The b_{ij} are the weakest conditions under which t_i and u_j are bisimilar. In fact we can show that for any b'

$$t_i[\overline{e}/\overline{x}_i] \sim_L^{b'} u_j[\overline{e}'/\overline{y}_j] \text{ implies } b' \models b_{ij}[\overline{e}, \overline{e}'/\overline{x}_i, \overline{y}_j] \tag{6}$$

To this end, assume $t_i[\overline{e}/\overline{x}_i] \sim_L^{b'} u_j[\overline{e}'/\overline{y}_j]$ and $\rho \models b'$. By Theorem 5.17 we have $t_i[\overline{e}/\overline{x}_i]\rho \sim_l u_j[\overline{e}'/\overline{y}_j]\rho$. Because $\overline{x}_i$ and $\overline{y}_j$ are disjoint, this can be rewritten as

$$t_i\rho[\overline{e}, \overline{e}'/\overline{x}_i, \overline{y}_j] \sim_l u_j\rho[\overline{e}, \overline{e}'/\overline{x}_i, \overline{y}_j]$$

Hence, by the definition of b_{ij}, $\rho[\overline{e}, \overline{e}'/\overline{x}_i, \overline{y}_j] \models b_{ij}$. Therefore $\rho \models b_{ij}[\overline{e}, \overline{e}'/\overline{x}_i, \overline{y}_j]$.

In particular, taking $i = j = 1$ in (6) we get $b \models b_{11}[\overline{e}_1, \overline{e}'_1/\overline{x}_1, \overline{y}_1]$.

Set $b_{ikjl} = c_{ik} \wedge d_{jl} \wedge b_{ij}$. Since both $\{\, c_{ik} \mid k \in K_i \,\}$ and $\{\, d_{jl} \mid l \in L_j \,\}$ are disjoint sets of booleans, $t_{ik} \sim_l^{b_{ikjl}} u_{jl}$. Let B_{ikjl} be the b_{ikjl}-partition guaranteed by Lemma 4.6. For each $b' \in B_{ikjl}$ define

$$I^{b'} = \{\, (p,q) \mid \alpha_{ikp} =^{b'} \beta_j \text{ and } t_{f(i,k,p)}[\overline{e}_{ikp}/\overline{x}_{f(i,k,p)}] \sim_L^{b'} u_{g(j,l,q)}[\overline{e}'_{jlq}/\overline{y}_{g(j,l,q)}] \,\}.$$

Note that each $I^{b'}$ is total and surjective on $P_{ik} \times Q_{jl}$.

Let $\{\, Z_{ij} \mid i \in I,\ j \in J \,\}$ be a set of new process variables and

$$V^{b'} = \sum_{(p,q) \in I^{b'}} \alpha_{ikp}.Z_{f(i,k,p)g(j,l,q)}(\overline{e}_{ikp}, \overline{e}'_{jlq})$$

Consider the standard equation set

$$E : Z_{ij}(\overline{x}_i, \overline{y}_j) = \sum_{k\in K_i, l\in L_j} \texttt{if } c_{ik} \wedge d_{jl} \texttt{ then } \sum_{b'\in B_{ikjl}} V^{b'} \quad i \in I,\ j \in J$$

We claim that E is provably b-satisfied by t when each Z_{ij} is instantiated with $(\overline{x}_i\overline{y}_j)(\texttt{if } b_{ij} \texttt{ then } t_i)$. We are to prove that, for each $i,\ j$

$$\vdash_{\mathtt{r}} b_{ij} \rhd t_i = (\sum_{k\in K_i, l\in L_j} \texttt{if } c_{ik} \wedge d_{jl} \texttt{ then } \sum_{b'\in B_{ikjl}} V^{b'})\theta_{ij} \tag{7}$$

where $\theta_{ij} \equiv [(\overline{x}_i\overline{y}_j)(\texttt{if } b_{ij} \texttt{ then } t_i)/Z_{ij} | i \in I, j \in J]$. By (4) and PARTITION (7) can be reduced to, for each pair of $k,\ l$

$$\vdash_{\mathtt{r}} b_{ikjl} \rhd \sum_{k\in K_i} \texttt{if } c_{ik} \texttt{ then } t_{ik} = \texttt{if } c_{ik} \wedge d_{jl} \texttt{ then } \sum_{b'\in B_{ikjl}} V^{b'}\theta_{ij}$$

which, by Proposition 5.2 (noting that $b_{ikjl} \models c_{ik} \wedge d_{jl}$) and PARTITION, can be further reduced to, for each $b' \in B_{ikjl}$

$$\vdash_{\mathtt{r}} b' \rhd t_{ik} = V^{b'}\theta_{ij} \tag{8}$$

We can derive (8) as follows

$$\begin{array}{ll}
\vdash_{\mathtt{r}} & b' \ \rhd\ V^{b'}\theta_{ij} \\
= & \displaystyle\sum_{(p,q)\in I^{b'}} \alpha_{ikp}.(\overline{x}_{f(i,k,p)}\overline{y}_{g(j,l,q)})(\texttt{if } b_{f(i,k,p)g(j,l,q)} \texttt{ then } t_{f(i,k,p)})(\overline{e}_{ikp}, \overline{e}'_{jlq}) \\
\stackrel{\text{BETA}}{=} & \displaystyle\sum_{(p,q)\in I^{b'}} \alpha_{ikp}.\texttt{if } b_{f(i,k,p)g(j,l,q)}[\overline{e}_{ikp}, \overline{e}'_{jlq}/\overline{x}_{f(i,k,p)}, \overline{y}_{g(j,l,q)}] \texttt{ then } t_{f(i,k,p)}[\overline{e}_{ikp}/\overline{x}_{f(i,k,p)}] \\
\stackrel{5.2}{=} & \displaystyle\sum_{(p,q)\in I^{b'}} \alpha_{ikp}.\texttt{if } b' \texttt{ then if } b_{f(i,k,p)g(j,l,q)}[\overline{e}_{ikp}, \overline{e}'_{jlq}/\overline{x}_{f(i,k,p)}, \overline{y}_{g(j,l,q)}] \texttt{ then } t_{f(i,k,p)}[\overline{e}_{ikp}/\overline{x}_{f(i,k,p)}] \\
\stackrel{(6),5.2}{=} & \displaystyle\sum_{(p,q)\in I^{b'}} \alpha_{ikp}.\texttt{if } b' \texttt{ then } t_{f(i,k,p)}[\overline{e}_{ikp}/\overline{x}_{f(i,k,p)}] \\
\stackrel{5.2}{=} & \displaystyle\sum_{(p,q)\in I^{b'}} \alpha_{ikp}.t_{f(i,k,p)}[\overline{e}_{ikp}/\overline{x}_{f(i,k,p)}] \\
\stackrel{S2-S4}{=} & \displaystyle\sum_{p\in P_{ik}} \alpha_{ikp}.t_{f(i,k,p)}[\overline{e}_{ikp}/\overline{x}_{f(i,k,p)}] \\
\equiv & t_{ik}
\end{array}$$

The step last but one relies on $I^{b'}$ being total.

Symmetrically, using the fact $I^{b'}$ is surjective, we can show E is provably b-satisfied by u. □

Proposition 5.20 *Let $E : X_i(\overline{x}_i) = H_i,\ 1 \leq i \leq m$, be a standard equation set. If t and u both provably b-satisfy E then $\vdash_{\mathtt{r}} b \rhd t = u$.*

Proof: Similar to the proof of Theorem 5.7 in [Mil84]. □

Summarizing the above three propositions gives the completeness theorem:

Theorem 5.21 *For guarded process terms t and u, $t \sim_{\mathtt{L}}^{b} u$ implies $\vdash_{\mathtt{r}} b \rhd t = u$.*

It can also be shown that adding EA or E-INPUT to $\vdash_{\mathtt{r}}$ gives rise to a proof system for early bisimulation, complete over guarded processes. Furthermore, with the addition of the three τ-laws these proof systems can be extended to observation congruences. The details are rather involved and are omitted here.

So far we have only considered guarded recursions. For pure CCS there are elegant laws which are sufficient to remove all unguarded recursions [Mil89b] and thus extending the axiomatization to the whole regular language. However, in value-passing calculi the combination of unguarded recursion and the conditional construct may result in non-trivial situations. For example let us consider processes P and Q defined as

$$\begin{array}{rcl} P & \Leftarrow & c?x.d!0.P \\ Q & \Leftarrow & c?x.Q'(x) \\ Q'(x) & \Leftarrow & \texttt{if } x = 0 \texttt{ then } d!x.Q \texttt{ else } Q'(x-1) \end{array}$$

where x ranges over natural numbers. Clearly P and Q are equivalent, but in order to derive this we need to rely on the inductive nature of natural numbers, and close interaction between reasoning about processes and reasoning about data becomes inevitable.

6 Summary and Related Work

We have discussed the complications arising from generalizing the standard theory of pure process algebra to the value-passing setting, and presented an approach to tackle these problems. The main focus is on the theory of bisimulation. We have demonstrated how the data aspect can be separated from processes so that a theory of bisimulation for value-passing processes can be developed without depending on any particular theory of data. This "symbolic" approach manipulates open terms with boolean expressions constraining their free data variables, in contrast to the traditional process algebra theory which only deal with closed terms. As a result the problem of infinite branching, due to instantiating input variables with values, is avoided, while the standard bisimulation equivalences are captured.

We have also developed finitary proof systems for the various bisimulation-based equivalences in our value-passing language. Again, reasoning about data is separated from reasoning about processes. These proof systems preserve the standard equational axioms for bisimulation-based equivalences while generalizing equational reasoning by introducing inference rules.

There are many aspects of value-passing calculi that are not covered in this chapter. Denotational semantics for value-passing processes is discussed at length in [HI93b, HI93]. Various modal logics have been proposed for such processes [HL93, RH97]. Algorithmic aspects are addressed in [Lin96, Pac96, Lin98b]. The "symbolic" style proof systems presented in Section 5 has been implemented in the interactive proof assistant VPAM [Lin93].

In our language, channel names have been separated from values and hence are not allowed to be transmitted between processes. Lifting this restriction results in a very expressive language known as π-calculus which is covered in [Par99].

For other approaches to value-passing processes, we want to mention μCRL which extends ACP with data introduced by algebraic specifications [GP90, GP91]. The theory of μCRL is described in detail in [GR00].

In [Sch94] *parameterized graphs*, which are similar to symbolic transition graphs, were proposed as a model for value-passing processes.

Acknowledgment The work presented in this chapter is the result of a long term collaboration with Matthew Hennessy, and we are indebted to him for many original ideas and insights. Thanks also go to Joachim Parrow for his constructive comments on the very first draft of this chapter, to Julian Rathke and Luca Aceto for correcting many English mistakes and suggesting several improvements. Comments from the anonymous referee are also helpful.

References

[BHR84] Brookes, S.D., Hoare, C.A.R. and Roscoe, A.W., A Theory of Communicating Sequential Processes, *Journal of the ACM* 31(3), pp. 560-599, 1984.

[Bru96] G. Bruns. *Distributed Systems Analysis with CCS.* Prentice Hall, 1996.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based verification tool for finite state systems. *ACM Transactions on Programming Systems*, 15:36–72, 1989.

[DNH84] DeNicola, R. and M. Hennessy. "Testing Equivalences for Processes." *Theoretical Computer Science*, 24, 1984, pp. 83-113.

[FLP84] Francez, N., Lehman, D. and Pnueli, A., A Linear History of Semantics for Languages with Distributed Processing, *Theoretical Computer Science*, 32, 25-46, 1984.

[GL98] J.F. Groote and S.P. Luttik. Undecidability and completeness results for process algebras with alternative quantification over data. Technical Report SEN-R9806, CWI, Amsterdam, 1998.

[GP90] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, 1990.

[GP91] J.F. Groote and A. Ponse. Proof theory for μCRL. Technical Report CS-R9138, CWI, Amsterdam, 1991.

[GR00] J.F. Groote and M.A. Reniers. Algebraic Process Verification. Chapter 6.1 of this handbook.

[Hen88] M. Hennessy. *An Algebraic Theory of Processes.* MIT Press, 1988.

[Hen91] M. Hennessy. A proof system for communicating processes with value–passing. *Formal Aspects of Computer Science*, 3:346–366, 1991.

[HI93b] Hennessy, M., Ingólfsdóttir, A. Communicating Processes With Value - Passing and Assignment, *Journal of Formal Aspects of Computing*, (1993) 5:432-466, 1993

[HI93] M. Hennessy and A. Ingolfsdottir. A theory of communicating processes with value-passing. *Information and Computation*, 107(2):202–236, 1993.

[HL95] M. Hennessy and H. Lin. Symbolic bisimulations. Theoretical Computer Science, 138:353–389,1995.

[HL96] M. Hennessy and H. Lin. Proof Systems for message passing processes. Formal Aspects of Computing, 8: 379–407, 1996.

[HL97] M. Hennessy and H. Lin. Unique Fixpoint Induction for Message-Passing Process Calculi. *CATS'97, Proc. Computing: the Australasian Theory Symposium*, pages 122–131. 1997.

[HL93] M. Hennessy and X. Liu. A modal logic for message passing processes. In *Proc. of 5th International Conference on Computer Aided Verification*, number 697 in Lecture Notes in Computer Science, pages 359–370, 1993.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. Journal of the Association for Computing Machinery, Vol 32, No 1, pages 137-161. 1985.

[HP80] M, Hennessy and G. Plotkin. A Term Model for CCS, *Springer-Verlag Lecture Notes in Computer Science*, vol.88, 1980.

[HR86] C.A.R. Hoare and A.W. Roscoe. The laws of occam. Technical Report PRG Monograph 53, Oxford University Computing Laboratory, 1986.

[Hoa78] Hoare, C.A.R. Communicating Sequential Processes, *Comm. ACM*, 21(8), 666-677, 1978.

[Ing95] A. Ingólfsdóttir. Late and early semantics coincide for testing. *Theoretical Computer Science*, 146(1–2):341–349, 1995.

[Lin93] H. Lin. A verification tool for value-passing processes. In *Proceedings of* 13^{th} *International Symposium on Protocol Specification, Testing and Verification*, IFIP Transactions. North-Holland, 1993.

[Lin96] H. Lin. Symbolic Transition Graphs with Assignment. In U. Montanari and V.Sassone, editors, *Proceedings CONCUR 96,* Pisa, volume 1119 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 1996.

[Lin98b] H. Lin. "On-the-fly" Instantiation of Value-passing Processes. In *Proceedings FORTE/PSTV'98*, Paris, Nov. 1998.

[MM79] G. Milne , R. Milner: Concurrent Processes and Their Syntax, *Journal of the Association for Computing Machinery*, Vol. 26, No. 2, pp. 302-321, 1979.

[Mil80] Milner, R. A Calculus of Communicating Systems, *Lecture Notes in Computer Science 92*. Springer-Verlag, Berlin, 1980.

[Mil83] Milner, R. Calculi for Synchrony and Asynchrony, *Theoretical Computer Science*, 25, pp 267–310, 1983

[Mil84] R. Milner. A complete Inference System for a Class of Regular Behaviours. *J. Computer and System Science*, 28(3):439–466, 1984.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mil89b] R. Milner. A complete axiomatisation for observational congruence of finite-state behaviours. *Information and Computation*, 81:227–247, 1989.

[Mil92] R. Milner. The polyadic pi-calculus. In W. R. Cleaveland, editor, *Proceedings CONCUR 92,* Stony Brook, New York, volume 630 of *Lecture Notes in Computer Science*, p.1, Springer-Verlag, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I and II. *Information and Computation*, 100(1):1–40, 1992.

[Par99] J. Parrow. An Introduction to the π-Calculus. Chapter 3.2 of this Handbook.

[PS93] J. Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. *Information and Computation*, 120(2):174–197, 1995.

[Pac96] P. Paczkowski. Characterizing Bisimilarity of Value-passing Parameterised Processes. The Infinity Workshop, Pisa, Italy, pp.47-55, August 1996.

[Plo81] Plotkin, G., "A Structural Approach to Operational Semantics", Technical Report

[Par81] D. Park: *Concurrency and Automata on Infinite Sequences*, Theoretical Computer Science VII, Lecture Notes in Computer Science 104, Springer-Verlag, 1981. DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[Rat97] J. Rathke. Unique Fixpoint Induction for Value-Passing Processes. *Proc. 12th IEEE Symposium on Logic in Computer Science*, IEEE Press, 1997.

[Rat97b] J. Rathke. Symbolic Techniques for Value-passing Calculi. Ph.D Thesis, University of Sussex, 1997.

[RH97] J. Rathke and M. Hennessy. Local model checking for value-passing processes. *Proc. TACS'97, International Symposium on Theoretical Aspects of Computer Software*, Sendai. Springer-Verlag, 1997.

[Sa96] D. Sangiorgi. A Theory of bisimulation for π-calculus. *Acta Informatica*, 33, 1996.

[Sch94] M.Z. Schreiber. Value-passing Process Calculi as a Formal Method Ph.D Thesis, Imperial College, London, 1994.