April 1991

Formal Models for Concurrent Communicating Systems

Anthony S. Kosky
University of Pennsylvania

Recommended Citation

Anthony S. Kosky, "Formal Models for Concurrent Communicating Systems", . April 1991.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-34.

Formal Models for Concurrent Communicating Systems

Abstract

This report was originally written to fulfill in part the requirements of the author's WPE examinations, part of the qualifying examinations for the University of Pennsylvania'a Computer Science Ph.D program. The report first introduces CCS and uses it to illustrate various features of established methods of modelling concurrent, communicating systems. The report then goes on to describe and investigate two new models for such systems: The *Chemical Abstract Machine*, a simple yet predominant in most models for such systems; and the π-calculus, a calculus similar in many respects to CCS, but able to model mobile processes and other, more difficult phenomena.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-34.

Formal Models for Concurrent Communicating Systems

MS-CIS-91-34
LOGIC & COMPUTATION 33

Anthony S. Kosky

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

April 1991

Formal Models for Concurrent Communicating Systems

Anthony S. Kosky

April 29, 1991

Abstract

This report was originally written to fulfill in part the requirements of the author's WPE examinations, part of the qualifying examinations for the University of Pennsylvania's Computer Science PhD. program. The report first introduces CCS and uses it to illustrate various features of established methods of modelling concurrent, communicating systems. The report then goes on to describe and investigate two new models for such systems: the *Chemical Abstract Machine*, a simple yet expressive abstract machine model, free from the architectual considerations predominant in most models for such systems; and the π-calculus, a calculus similar in many respects to CCS, but able to model mobile processes and other, more difficult phenomena.

1 Introduction

Systems involving concurrent, communicating processes are becoming increasingly common and important in computer science. The use of formal models for such systems can give us insights in their nature and improve our understanding of how they can be expected to function. This leads to greater confidence in such systems, and possibly to improvements in their design and implementation. A well thought out formal methodology can aid in the design and analysis of a system, and hence help make it more reliable and easily maintained. In addition the study of such models may well lead to developments in concurrent programming languages and paradigms which, as yet, are still in their infancy.

We will look at three different formalisms in this paper: CCS, the chemical abstract machine and the π-calculus or calculus of mobile processes. Of the three CCS is the oldest and best established, and has a well understood and highly developed theory. We will use it to illustrate the problems involved with modelling concurrency and some common approaches to dealing with these problems.

Chemical abstract machines provide an new abstract machine formalism for representing concurrent systems, rather than a particular calculus or language. Such abstract machine formalisms have proved to be useful in the study of sequential systems, but there is no sufficiently expressive and well-defined formalism of this type for concurrent systems in common usage.

The π-calculus is a process algebra, similar in nature to CCS, which attempts to capture the concept of dynamically varying architectures of systems. As a paradigm, it is considerable more expressive than CCS and has aspirations toward being a canonical calculus for concurrent systems.

We will be concentrating on trying to capture the motivations behind these models, and the rationales for their various design decisions, rather than embarking on a detailed mathematical analysis, since comprehensive theoretical studies of all three models are available elsewhere.

2 A Calculus of Communicating Systems

We will start our look at models for concurrency with Milner's *Calculus of Communicating Systems* ([Mil89]) or *CCS*. CCS was designed as a calculus for specifying and reasoning about "complex dynamic systems". It is by far the most well established of the three models we will look at, and has been applied to a number of real world problems. Consequently it provides a good model to use to explain the concepts prevalent in the study of concurrency, and an excellent backdrop against which to study the two newer and more radical models we will be considering later.

A system, from the CCS view of things, consists of a number of autonomous *agents*, each of which may execute *actions* concurrently with other agents in the system. Each action may either be internal to the agent that executes it, or may constitute a communication with another agent in the system. Some agents may in turn be considered to be systems in their own right, consisting of an number of *subagents* communicating with each other and carrying out their own internal actions. Consequently the internal actions of such an agent may be communications between its subagents, or internal actions executed by its subagents.

CCS adopts an observational view of systems and agents; that is, it equates an agent with its ability to communicate with other agents in a system, and a system with its ability to communicate with the outside world. In particular a system is thought of as communicating with an external observer, and the observer, in turn, is thought of as part of a larger system in which the first system is an agent. This uniformity of approach is one of CCS's greatest strengths; it means that the same notions as used to model the communications of agents can be used to model the "output" of a system, thus leading to an exceptionally simple and elegant calculus.

One of the first things to decide when devising a calculus such as CCS is what mechanism to use for communication. If we take the view that agents communicate by sending messages to one another then there are still a number of questions which arise when trying to decide on a discipline for communication: should there be a delay between an agent sending a message and another agent receiving it, and, if so, should it be possible for there to be several messages waiting for a receiving agent at one time? Should it possible for an agent to receive messages in a "random" order, or should they be received in the same order that they are sent? Should it be possible to receive or read a message several times, or should each message be sent and received exactly once? Even the decision that a communication consists of one agent sending a message to another is by no means pre-ordained: Hoare's CSP calculus ([Hoa85]) takes the view that a process should simultaneously "broadcast" a message to all agents capable of receiving it, while models based on shared memory are also possible.

CCS uses "handshakes" for communication between agents, which provides the simplest possible discipline for which each atomic communication involves exactly one sender and one receiver. Two agents synchronize on a communication, so that the sender and the receiver both carry out communication actions simultaneously. It can be shown that, using only this simple discipline, we can simulate the more complex mediums for communication mentioned above by introducing intermediary agents which receive messages from the initial, sending agent and then pass them on to the receiving agent. Consequently it is sufficient to make this simple form of communication the only primitive form of communication in the calculus.

2.1 A basic version of CCS

The version of CCS used for practical applications supports value passing communications, conditional operators, parameterised definitions of agents and so on. In this section we will describe a more basic version of CCS without these things (in particular communications are considered to be merely synchronizations without any additional informational content), however, as we will show in section 2.4, these extensions can be modeled in the basic calculus. It follows that we can consider these more useful features of CCS to be shorthand notations for more complex expressions in our basic calculus, and so we may safely limit our theoretical studies to the basic calculus.

We will define the set $\mathcal{P}$ of *agents* to be the basic entities with which CCS is concerned, and will use the letters $P, Q, R, \ldots$ to range over arbitrary agents. An agent carries out actions that are either communications or internal (un-observable) actions, and on carrying out an action it changes *state*. We equate the concepts of states and agents so that, when an agent carries out an action and goes into a new state, we consider it to change into a new agent.

We will assume a set $\mathcal{X}$ of *agent variables*, ranged over by $X, Y, \ldots$, which we will need in order to define recursive agents later, and a set $\mathcal{N}$ of *names*, ranged over by $a, b, \ldots$, which correspond to channels for communication. We define the set $\mathcal{L}$ of *labels*, ranged over by $l, l', \ldots$, which represent the communications ports of agents, by $\mathcal{L} = \{a, \overline{a} | a \in \mathcal{N}\}$. Here the label a is said to be an *input port* on channel a, and $\overline{a}$ is said to be an *output port* on channel a, though the distinction between input and output ports is purely arbitrary when we are considering the non-value-passing calculus (they will have more significance when we consider the value passing calculus later). We say that the label $\overline{a}$ is the *complement* of a, and a is the complement of $\overline{a}$, and, at certain points, will write $\overline{\overline{a}}$ for a.

In addition we will use the labels $\overline{a}$ and a to represent the *actions* of performing a single, atomic communication on the ports $\overline{a}$ and a respectively. There is one more kind of action that we have yet to define, and we will come to it in a minute. In the mean time we will use the symbols $\alpha, \beta, \ldots$ to range over actions.

Basic agents

The first and simplest agent we will look at is the *inaction agent*, $\mathbf{0}$. It is not capable of performing any action whatsoever and is used to represent a process that has "stopped" or "finished". In addition each of our agent variables, X, is an agent.

The next class of agents to consider are those formed using the *prefixing operator* ".". If α is an action and P is an agent then $\alpha.P$ denotes the agent which performs the action α and then proceeds to behave like P. For example the agent

$$a.\overline{b}.\mathbf{0}$$

takes an input signal on channel a, outputs a signal on channel b and then stops.

For every action α (including those we have yet to define) we will define a *transition relation* $\xrightarrow{\alpha}$ on agents, where $P \xrightarrow{\alpha} P'$ means that the agent P may perform the action α, changing to P' in doing so. Continuing the example above we would have

$$a.\overline{b}.\mathbf{0} \xrightarrow{a} \overline{b}.\mathbf{0}$$

We will define the transition relations to be the smallest relations satisfying a number of inference rules, and we will introduce the relevant inference rules for each class of agents together with the definition of the agents. So our inference rule for prefixing is:

prefix

$$\alpha.P \xrightarrow{\alpha} P$$

meaning that any agent of the form $\alpha.P$ can perform the action α and become the agent P. There are no inference rules for $\mathbf{0}$ or agent variables X since there are no transitions which apply to them. Often we will omit the $\mathbf{0}$ on the end of an agent of the form $\alpha_1.\alpha_2\ldots\alpha_n.\mathbf{0}$, writing it simply as $\alpha_1.\alpha_2\ldots\alpha_n$.

Parallel Composition

The next class of agents are *parallel compositions*, formed using the *composition operator* $|$. An agent $P|Q$ represents the two agents P and Q operating in parallel. $P|Q$ can carry out any action that P or Q can individually, with the relevant transformations being applied to its components, but, in addition, it can also carry out internal actions which constitute communications between P and Q. Such an internal action consists of P and Q both executing atomic communications on some mutually complemented ports. For example there are three possible actions that the agent

$$a.b|\overline{a}.c$$

can carry out. It can execute the action a, transforming to $b|\overline{a}.c$; or $\overline{a}$, becoming $a.b|c$; or it can carry out an internal action consisting of a communication between its two components on the channel a, and become $b|c$.

There are two inference rules needed to represent the external actions of a parallel compostion of agents, which are

com1

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

com2

$$\frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

The first rule should be read as saying "if $P \xrightarrow{\alpha} P'$ then it follows that that $P|Q \xrightarrow{\alpha} P'|Q$", and the second rule should be interpreted in a simillar manner.

When it comes to giving a rule for a communication between P and Q in the agent $P|Q$ we are faced with a problem: clearly some transformation takes place, but none of the actions we have so far defined fits. Clearly we need some action(s) to represent the communications between the components of a parallel composition of agents. The approach taken in CCS is to say that all internal actions are indistinguishable, and represent them by a single action τ, so that we can now define our set of *actions* by $Act = \mathcal{L} \cup \{\tau\}$. τ represents a *complete* action formed by two agents running in parallel carrying out a handshake on some communication channel. The decision to consider all internal actions to be indistinguishable is a very important one, and is responsible for much of the nature of CCS. In fact, to as large an extent as possible, internal actions are considered to be invisible, and a theory of *observational equivalence* is developed where τ actions are only indirectly observable through the effect they have on which observable actions may take place.

The inference rule for internal communications is

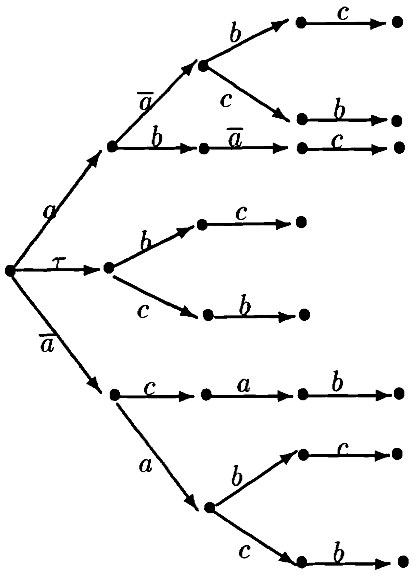
com3

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

so, for example we would have

$$a.b|\overline{a}.c \xrightarrow{\tau} b|c$$

We can represent the transitions applicable to such an agent by means of a transition tree:



Recursive Agents

Our next constructor for agents is a *fixed point operator* which allows us to construct recursively defined agents. Recursive agents have the form

$$\mathbf{fix}_j(\{X_1 = P_1, \ldots, X_n = P_n\})$$

which denotes the agent X_j where the agent variables $X_1, \ldots, X_n$ behave like the agents $P_1, \ldots, P_n$ respectively. Since each of the P_i can contain any of the variables X_j this allows for recursive definitions of agents. For example the agent $\mathbf{fix}_1(\{X = a.\overline{b}.X\})$ repeatedly takes an input signal on channel a and then outputs a signal on channel b. This agent could also be written as $\mathbf{fix}_1(\{X_1 = a.X_2, X_2 = \overline{b}.X_1\})$. In order to make our notation a little less cumbersome we will often use the shorthand notation for writing sequences of expressions, so that we would write $\widetilde{X}$ for the sequence of agent variables $X_1, \ldots, X_n$ and

$$\mathbf{fix}_j(\widetilde{X} = \widetilde{P})$$

for the fixed point expression shown above.

In defining the necessary inference rules for fixed point operators we will borrow the notion of *substitution* from the lambda calculus: we write $P\{\widetilde{Q}/\widetilde{X}\}$ for the agent formed by simultaneously replacing each occurence of the agent variables X_i in P by Q_i for $i = 1, \ldots, n$. The rules for fixed point operators are then

fix$_j$

$$\frac{P_j\{\widetilde{\mathbf{fix}(\widetilde{X} = \widetilde{P})}/\widetilde{X}\} \xrightarrow{\alpha} P'}{\mathbf{fix}_j(\widetilde{X} = \widetilde{P}) \xrightarrow{\alpha} P'}$$

Intuitively one can read this rule as saying any action that can be performed by unfolding the fix expression once can be performed by the fix expression itself.

Note that this is the only rule we will give which gives any meaning to agent variables. An agent variable which is used in a **fix** expression is said to be bound that the **fix** operator, and an agent variable is said to be free if it is not bound by any **fix** operator. Since we have no rules to deal with free agent variables they will behave just like inaction, and, for the remainder of our discussion of CCS, we will assume that all the agents we deal with do not contain any free variables.

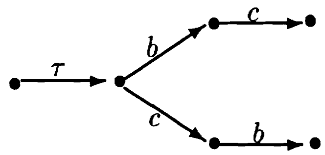
The fixed point notation, while theoretically elegant and easy to handle, is a little cumbersome for practical use. Instead pratical versions of CCS often make use of *agent constants*, $A, B, C, \ldots$, such that, for each agent constant A there is a corresponding definition $A \overset{\text{def}}{=} P$, meaning that A behaves like the agent P. Since the agents used in these definitions can contain agent constants, this also allows for recursive definitions of agents. For example the agent described in the example above, which repeatedly carries out the actions a and then $\overline{b}$, could be represented by the agent constant A with definition

$$A \overset{\text{def}}{=} a.\overline{b}.A$$

It is clear that agent constants can be considered to be short hand notation or "macros" for the CCS calculus using only the fixed point operator, and so, while we may use them in some of our examples, we will limit our formal study to the calculus with fixed point operators.

Restriction

We now introduce the restriction operator $\setminus$. If P is an agent and L is a set of names then $P \setminus L$ is the agent that behaves like P, except it can't perform either of the actions a or $\overline{a}$ for any $a \in L$ (and neither can any agent reached from it via a series of transitions). Thus the restriction operator "hides" some communication channels of an agent so that they may only be used for internal communications. For example the agent $(a.b|\overline{a}.c) \setminus \{a\}$ may only perform the action τ followed by the actions b and c in either order, as shown in the transition tree:



The rule defining the behaviour of restricted agents is

restrict

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \text{where } \alpha \notin L \cup \overline{L}$$

where $\overline{L}$ denotes the set of complements of labels in L. If a is a single name then we will use $P \setminus a$ as shorthand for $P \setminus \{a\}$.

Summation

Finally we introduce summations. Given a family of agents $\{P_i | i \in I\}$ the summation $\Sigma_{i \in I} P_i$ is an agent which chooses one of the agents P_i to behave like. The choice, however, is driven by the environment in which the agent exists: if the environment offers the summation the ability to carry out an action α then the summation must choose an agent capable of executing that action. Put another way: the choice of which agent in the summation to execute is put off until an action is executed by one of the agents and then the agent executing the action is chosen and the rest are discarded. For example the agent $\Sigma\{b.c,\ c.b\}$ can either perform the action b followed by the action c, or c followed by b. It is interesting to note that, if we consider the initial action τ to be invisible in the example $(a.b|\overline{a}.c)\backslash a$ given earlier, then these two agents have the same observable behaviour.

We have introduced an operator which allows infinite summations, and we will need to make use of them for simulating the value passing calculus later. However, for most purposes, binary summation is sufficient and we use the binary operator $+$ to denote this. Using this binary operator the example above would be written $b.c + c.b$.

The rule for summation is

sum

$$\frac{P_i \xrightarrow{\alpha} P'}{\Sigma_{i \in I} P_i \xrightarrow{\alpha} P'}$$

It is worth remarking that if several agents in a summation are capable of carrying out an action α and the environment offers this action then summation must choose non-deterministically between the suitable agents. In particular, since any agent may execute a τ action at any time, if several agents in a summation are capable of carrying out τ actions then the summation can non-deterministically choose to execute one of those agents.

2.2 Simulating concurrency in an asynchrnous calculus

Notice that the semantics of CCS, as defined by the transition relations $\xrightarrow{\alpha}$, does not actually model concurrency directly: that is, two agents running in parallel can not actually both carry out actions at the same time, during a single atomic transition, unless they are the two parts of a single communication. Instead the actions of two agents composed in parallel are interleaved in a non-deterministic manner. Such a calculus is known as *asynchronous*, meaning that the actions of the individual agents in a parallel composition do not synchronise with one another. A *synchronous* calculus is one where, on each time step, every agent in a parallel composition of agents performs exactly one action. In such a calculus if an agent wants to wait while other agents carry out various actions it must explicitly execute a "do-nothing" action. Such a calculus is described in [Mil89] and it is shown that it can be used to simulate CCS.

So we can see that there are certain aspects of parallelism that are not captured by CCS, but the question arises of whether they are in fact of any significance. In effect what we are doing is imposing an ordering on the actions carried out by a parallel composition of agents, such that, if an agent in the composition carries out an action α before an action β, then α will come before β in the ordering (or, if α is part of a complete communication, then the β will come after the corresponding τ action of the composition, and, similarly, if β is part of a communication), but otherwise the ordering is arbitary. We can attempt to justify this ordering by claiming that each (visible) action constitutes an observation, and that observations must be made in some order even if the actual actions occur and become ready to be observed simultaneously.

What is more relevant is whether we lose any expressivity by imposing such an ordering on actions, and we can argue that we don't. Suppose that two agents in a parallel composition, P and Q say, simultaneously carry out two actions, α and β respectively. If α and β do not constitute a single communication between P and Q then it follows that P executing α on its own has no effect on Q, and Q executing β has no effect on P. Consequently we can arbitrarily assume that either α is executed before β or β is executed before α without effecting the overall change to the system, or the actions carried out by the system (except that it now takes two transitions to model a change that actually occured in a single transformation). It follows that we can express all the possible transformations a system can undergo using sequences of transitions, one per action, and so we don't lose any of the expressivity of the model by adopting this technique. We can conclude that, for the purposes of specifying and reasoning about the behaviour of concurrent systems, an interleaving semantics is satisfactory.

2.3 Bisimulation and Observational Equivalence

As we stated earlier, CCS is oriented toward modeling the observable behaviour of processes, and so a great deal of effort has gone into constructing equivalence relations on agents which capture the idea of *observational equivalence*. However in order to define such an equivalence relation we must take into account some special special considerations: we must decide whether we consider internal actions to be observable (in general all internal actions are considered to be indistinguishable); we must consider which sequences of observations one can fail to make in addition to which sequences one can make; and we must consider whether the equivalence is preserved by our various operators. The second consideration here seems strange at first if one is used to thinking about deterministic programs. In a non-deterministic calculus, such as CCS, it may be possible to observe some behaviour of a process on one execution and then fail to observe the same behaviour on the next execution: for example an input to the process which was accepted after some sequence of actions on the first execution might be rejected, or cause the system to deadlock, on a second execution of the process.

We will consider three notions of equivalence in this section: *strong bisimilarity*, *weak bisimil-*

iarity and *observational congruence.* Of the three strong bisimilarity makes more distinctions than we would like, differentiating between agents which behave identically in all situations except for the internal actions they execute, while weak bisimilarity makes too few distinctions, failing to be a congruence. Observational congruence is accepted as the definition of equality amongst agents, though other equivalences, such as *failures equivalence* have also been proposed.

Strong Bisimilarity

Strong bisimilarity equates two agents if they can carry out and fail to carry out exactly the same sequences of (internal and external) actions. Formally we first define strong bisimulation relations as follows: a relation $\mathcal{R}$ on agents is said to be a *strong bisimulation* iff

1. If $P\mathcal{R}Q$ and $P \xrightarrow{\alpha} P'$ then there is a Q' such that $Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{R}Q'$.
2. If $P\mathcal{R}Q$ and $Q \xrightarrow{\alpha} Q'$ then there is a P' such that $P \xrightarrow{\alpha} P'$ and $P'\mathcal{R}Q'$.

Strong bisimilarity, written $\sim$, is then defined to be the largest strong bisimulation on agents. Equivalently we can define $\sim$ by saying $P \sim Q$ iff there is a strong bisimulation $\mathcal{R}$ such that $P\mathcal{R}Q$ (this alternative definition can be shown to be equivalent, and is easier to prove for practical examples).

For example we would have

$$(a.b|\overline{a}.c) \setminus a \ \sim \ \tau.(b.c + c.b)$$

but, on the other hand

$$(a.b|\overline{a}.c) \setminus a \ \not\sim \ (b.c + c.b)$$

though, as we will soon discover, these two agents are weakly bisimilar. As we can see, if we want to think of τ transitions as being unobservable, then this equivalence is too strong.

Weak Bisimilarity

Weak Bisimilarity is similar to strong bisimilarity except that it only considers the sequences of observable actions that an agent may execute or fail to execute. We define it in a similar though slightly more complex way. First we must introduce some more notation.

For any sequence of actions $t = \alpha_1 \ldots \alpha_n \in Act^*$ we define the relation $\overset{t}{\Longrightarrow}$ on agents by

$$P \overset{t}{\Longrightarrow} Q \text{ iff } P(\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^* Q$$

That is $P \overset{t}{\Longrightarrow} Q$ iff P can transform to Q executing the sequence of actions t interspersed with any number of additional τ transitions. If $s \in Act^*$ is a sequence of actions then we write $\widehat{s}$ for the sequence of all the observable actions in s (that is, s with all the τ actions stripped out).

A relation $\mathcal{R}$ is said to be a *weak bisimulation* iff

1. If $P\mathcal{R}Q$ and $P \xrightarrow{\alpha} P'$ then there is a Q' such that $Q \overset{\widehat{\alpha}}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$.
2. If $P\mathcal{R}Q$ and $Q \xrightarrow{\alpha} Q'$ then there is a P' such that $P \overset{\widehat{\alpha}}{\Longrightarrow} P'$ and $P'\mathcal{R}Q'$.

We now define *weak bisimilarity* as the largest weak bisimulation relation, and write $P \approx Q$ if P and Q are weakly bisimilar. Once again we can give an alternative definition; saying that P is weakly bisimilar to Q iff there is a weak bisimulation, $\mathcal{R}$ such that $P\mathcal{R}Q$.

Weak bisimularity captures the idea of agents being capable of accepting and rejecting the same sequences of observable actions by themselves. However it is not acceptable as a notion of equality for agents because it is not a congruence. In particular it is not preserved by the operator $+$. For example we have $b \approx \tau.b$ but not $a + b \approx a + \tau.b$, since $a + \tau.b \xrightarrow{\tau} b$ but there is no Q such that $a + b \overset{\widehat{\tau}}{\Longrightarrow} Q$ and $Q \approx b$.

Observational Congruence

Observational Congruence is an attempt to strengthen weak bisimilarity to be a congruence relation. It states that, not only do two agents have the same observational behaviour, but that any larger agents or systems built out of the two agents in the same manner will also have the same observational behaviours.

In order to give a definition of observational congruence we will borrow some more notation from the lambda-calculus: a *context*, $C[]$ is an agent with a *hole*, $[]$, in it in the place of some sub-agent. If $C[]$ is a context then $C[P]$ denotes $C[]$ with the hole replaced by P.

An agent P is said to be *observationally congruent* to an agent Q, written $P = Q$, iff for every context, $C[]$, $C[P] \approx C[Q]$.

It is clear from this definition that observational congruence is indeed a congruence relation, and that if $P = Q$ then $P \approx Q$ (it can also be easily shown that, if $P \sim Q$ then $P = Q$). However this definition is somewhat difficult to check in practice, and so the following definition, which can be shown to be equivalent, is often used: two agents, P and Q, are said to be observationally congruent iff

1. if $P \xrightarrow{\alpha} P'$ then there is a Q' such that $Q \overset{\alpha}{\Longrightarrow} Q'$ and $P' \approx Q'$
2. if $Q \xrightarrow{\alpha} Q'$ then there is a P' such that $P \overset{\alpha}{\Longrightarrow} P'$ and $P' \approx Q'$

For example we can show that $(b.a.d|\overline{a}.c) \setminus a \; = \; (b.c.d + b.d.c)$ even though $(b.a.d|\overline{a}.c) \setminus a \; \not\sim$ $(b.c.d + b.d.c)$.

2.4 Extending CCS with Value-Passing

The calculus introduced so far is very simple and would be awkward to use for specifying and modeling real systems. We have already seen on section 2.1 that we can make the calculus more usable by adding agent constants, but that these may be thought of as short hand notation for expressions of the calculus with just the fixed point operator, and so we do not need to take them into account in our theoretical studies of the calculus. In this section we will look at some other useful extensions and see how they may be translated into terms of our basic calculus.

The first extension we will consider allows additional information, namely values, to be passed in communications, rather than limitting them to being the simple synchronisations we have been considering so far. We assume a set of *values*, V, ranged over by $v, w, \ldots$, and some *value variables*, $x, y, \ldots$. We now allow our prefixes to have the form $\overline{a}(v)$ (or $\overline{a}(x)$), $a(x)$ and τ, where a is any name. An agent of the form $\overline{a}(v).P$ means "output the value v on channel a and then proceed to execute P", while the agent $a(x).P$ means "input a value v on channel a, then bind it to the variable x in P and execute P". In an agent of the form $a(x).P$ the prefix $a(x)$ is said to *bind* the occurences of x in P (unless they are bound by the prefix of some sub-agent). In general we will assume that agents do not contain any *free* value variables. For example the agent

$$a(x).(\overline{b}(x)|\overline{c}(x))$$

takes some value v as input on channel a, and then outputs v on the channels b and c (in either order).

We can represent value passing in our basic calculus as follows: for every name, a, in our extended calculus we introduce a family of names, $\{a_v | v \in V\}$ into our basic calculus; we translate a term with an output prefix, $\overline{a}(v).E$ to $\overline{a_v}.\widehat{P}$, where $\widehat{P}$ is the translation of P; and we translate an input prefixed term, $a(x).E$ into a summation of versions of the term for each possible value v, $\Sigma_{v \in V} a_v.\widehat{E\{v/x\}}$. We also need to translate restrictions in order to take account of the new set of names.

We introduced the concept of agent constants earlier, claiming that they were more convenient to use than explicit fixed point operators. It is useful to further extend these agent constants by parameterising them on values. To do this we assume that every agent constant A has an *arity* n associated with it, and a defining equation of the form $A(x_1, \ldots, x_n) \overset{\text{def}}{=} E$, where the free variables of the term E come from $x_1, \ldots, x_n$. We can then use $A(v_1, \ldots, v_n)$ to denote the agent formed by substituting $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$ in E.

For example we can write a "one place buffer" agent, which takes an input on the channel in and then outputs it on the channel out and repeats the process, as the agent constant BUF (with arity zero), where

$$\begin{aligned} BUF() &\overset{\text{def}}{=} \mathtt{in}(x).BUF'(x) \\ BUF'(x) &\overset{\text{def}}{=} \overline{\mathtt{out}}(x).BUF() \end{aligned}$$

For any agent constant, A with arity n, in our extended calculus and every n-tuple of values, $(v_1, \ldots, v_n)$ (for which we will use the shorthand $\tilde{v}$), we introduce an agent constant, $A_{\tilde{v}}$ to our basic calculus. If A has the defining equation

$$A(x_1, \ldots, x_n) \overset{\text{def}}{=} E$$

then $A_{\tilde{v}}$ has the defining equation

$$A_{\tilde{v}} \overset{\text{def}}{=} E\widehat{\{\tilde{v}/\tilde{x}\}}$$

where we use the notation $\widehat{P}$ to denote the translation of an extended calculus agent P into the basic calculus.

There are a number of other useful extensions for handling values and value passing in CCS, for example conditional statements, all of which can be translated into the basic calculus in a similar manner. We summarise the translation of the extensions we have just described in the following table:

E	$\widehat{E}$
$\mathbf{0}$	$\mathbf{0}$
$\overline{a}(v).E$	$\overline{a_v}.\widehat{E}$
$a(x).E$	$\sum_{v \in V} a_v.E\widehat{\{v/x\}}$
$P \mid Q$	$\widehat{P} \mid \widehat{Q}$
$P \setminus L$	$\widehat{P} \setminus \{a_v \mid a \in L, v \in V\}$
$A(v_1, \ldots, v_n)$	$A_{v_1 \ldots v_n}$

3 Chemical Abstract Machines

The use of "abstract machines" in the study of sequential programming is well established: models such as Turing Machines are commonly used to study issues of computability and complexity, while models such as the SECD machine and Categorical Abstract Machine have been used to study the implementation of functional languages. However the use of abstract machines in the study of concurrent systems is not as well established. Certain existing models for concurrency, such as Petri nets or communicating automata, could be considered to be abstract machines, but lack expressivity. Process calculi, such as CCS, are well suited to specifying and reasoning about the behaviour of systems, but do not give us an idea of the computational complexity of a problem, or of where implementation difficulties are likely to arise. In addition existing models are predominantly concerned with the geographical or architectural aspects of systems, that is how individual process are interconnected and how they may interact with each other, which makes concurrent programs and threads of control difficult to reason about.

The *Chemical Abstract Machine*, as described in [BB90], utilizes the analogy of a *chemical solution* in which a number of *molecules* float freely, in order to provide a abstract machine model for concurrent systems which is expressive, intuitive and allows allows processes to interact in a manner which is largely independent of architectural constraints. The molecules in such a solution move around freely and interact with eachother as if the solution is being continuously stirred by some mechanism (such as Brownian motion). Molecules can *react* with one another according to a number of *reaction rules.*

For example let's consider a chemical abstract machine (CHAM) which generates the prime numbers between 2 and n. We take integers as our molecules and start with a solution containing all the integers from 2 to n. We equip the CHAM with just one reaction rule: any integer reacts with its multiples by destroying them. It is clear that, once no more reactions are possible, we are left with only prime numbers.

A chemical solution may be *heated*, in order to make large molecules break down into their component parts, or *cooled*, so that molecules come together to form larger compound molecules. A molecule which is capable of taking part in a reaction is called an *ion*, and, in general, a molecule breaks down into ions when it is heated sufficiently, and an ion cannot be heated any further.

Molecules may contain *subsolutions* enclosed in *membranes*, such that the subsolution may evolve independently within the molecule. This facility is, in a large part, responsible for the expressiveness of the chemical abstract machine paradigm, but also stretches the intuition somewhat. In order to allow a subsolution to react with the molecules outside, membranes are somewhat porous. In fact a single molecule can be isolated from the remainder of a subsolution by means of an *airlock* (again, a concept which stretches the intuition), and it is a molecule thus isolated that is liable to take part in reactions with molecules outside the membrane.

3.1 A CHAM for a Subset of CCS

In this section we will introduce the concepts of chemical abstract machines by means of an extended example, before going on to a formal definition in the next section. We will give a CHAM interpretation of CCS$^-$, a subset of CCS containing only the operators inaction, $\mathbf{0}$, prefixing, ".", parallel composition, $|$, the fixed point operators, $\mathbf{fix}_i$, and restriction, $\backslash$. Also we will disallow the prefix τ.

We will take CCS$^-$ agents as our molecules, and take solutions to be multisets of molecules, denoted $S = \{\!|p, q, r, \ldots|\!\}$. To start with we will only look at τ transitions.

Instead of implementing rules to extend the actions of individual molecules to parallel compositions, we break down parallel compositions into their component parts, and then allow these parts to react freely. So our first rules are:

parallel

$$p|q \rightleftharpoons p, q$$

reaction

$$a.p, \overline{a}.q \rightarrow p, q$$

The first rule is reversible, and says that a parallel composition, $p|q$, can be *heated* (symbol $\rightharpoondown$) to break down into its component molecules, p and q, and that any two molecules, p and q, can be *cooled* (symbol $\rightharpoondown$) to form a parallel composition $p|q$. It follows that a *hot* solution (one to which no more heating rules can be applied) has all the parallel compositions broken down so that the component molecules may interact freely with one another.

A molecule of the form $\alpha.p$ is said to be an *ion*, which means it is capable of taking part in a reaction. α is said to be the *valence* of the ion $\alpha.p$. The second law states that two ions with complementry valences can *react* (symbol $\rightarrow$) and in doing so discard their prefixes.

The laws can be applied to any group of molecules in a solution (though not to subterms of a molecule). In order to execute an agent p we form a solution $\{\!|p|\!\}$, heat the solution so that any possible reactions can be carried out, and then repeatedly apply the reaction rule (while keeping the solution hot), and finally *freeze* the solution (applying cooling rules until no more can be applied) to make it into a single molecule again. For example consider the agent $a.b.\mathbf{0}|\overline{a}.\mathbf{0}|\overline{b}.\mathbf{0}$. The execution goes along the lines of

$$\begin{array}{cll} & \{\!|a.b.\mathbf{0}|\overline{a}.\mathbf{0}|\overline{b}.\mathbf{0}|\!\} & \\ \overset{*}{\rightharpoondown} & \{\!|a.b.\mathbf{0}, \overline{a}.\mathbf{0}, \overline{b}.\mathbf{0}|\!\} & \text{(parallel)} \\ \rightarrow & \{\!|b.\mathbf{0}, \mathbf{0}, \overline{b}.\mathbf{0}|\!\} & \text{(reaction)} \\ \rightarrow & \{\!|\mathbf{0}, \mathbf{0}, \mathbf{0}|\!\} & \text{(reaction)} \end{array}$$

Notice that we are left with some extra $\mathbf{0}$'s at the end. We can get rid of them by introducing the rule

inaction cleanup

$$p, \mathbf{0} \ \rightharpoonup \ p$$

which just says that excess $\mathbf{0}$'s evapourate away on heating.

Intuitively our reaction transitions, $\rightarrow$, correspond to τ transitions, or, more accurately, $p \xrightarrow{\tau} q$ iff $\{\!|p|\!\} \overset{*}{\rightleftharpoons} \rightarrow \overset{*}{\rightleftharpoons} \{\!|q|\!\}$, while our heating and cooling rules correspond to structural eqivalences on agents.

In order to interpret an agent using a fixed point operator we must unfold the operator in order to make any possible reactions available. So we introduce the following heating rule for fixed point operators:

fixed point

$$\mathbf{fix}_j(\widetilde{X} = \widetilde{E}) \ \rightharpoonup \ E_j\{\widetilde{\mathbf{fix}}(\widetilde{X} = \widetilde{E})/\widetilde{X}\}$$

Restriction agents of the form $p \setminus L$ are more of a problem. If p is an ion then we have a molecule of the form $(\alpha.q) \setminus L$ and we can just bring the valence, α, outside the restriction provided that α is not in L or $\overline{L}$ (the complements of names in L). So we introduce the rule

restr. ion

$$(\alpha.p) \setminus L \ \rightleftharpoons \ \alpha.(p \setminus L) \text{ provided } \alpha \notin L \cup \overline{L}$$

However problems occur when p is a compound molecule: we want p to be able to evolve independently within the restriction $p \setminus L$. We handle this problem by enclosing p in a *membrane* (operator $\{\!| \cdot |\!\}$), which makes it into a *subsolution* of the molecule. All the rules that can be applied to a normal solution can be applied to such a subsolution and so it can undergo reactions by itself within the restriction.

restr. memb

$$p \setminus L \ \rightleftharpoons \ \{\!|p|\!\} \setminus L$$

We need some way of allowing the molecules in the subsolution to interact with molecules outside the restriction. In particular we need some way of isolating a molecule from the remainder of the subsolution and moving its valence outside the restriction so it can take part in a reaction. We do this in two stages by means of an *airlock*. An airlock is a device which isolates a molecule from a solution by forming a large molecule consisting of the isolated molecule and the remainder of the subsolution contained in a membrane, separated by the operator $\lhd$. We first introduce the *meta-rule* for airlocks:

airlock

$$\{\!|p, q_1, \ldots, q_n|\!\} \;\rightleftharpoons\; \{\!|p \lhd \{\!|q_1, \ldots, q_n|\!\}|\!\}$$

Next, if p is an ion, we make the whole *heavy molecule* into an ion with the valence of p using the rule:

heavy ion

$$(\alpha.p) \lhd S \;\rightleftharpoons\; \alpha.(p \lhd S)$$

Once a heavy ion has been formed the valence can be taken outside the restriction, using the *restr. ion* rule, so that it may be used in a reaction.

Great care has been excercised in formulating these rules to ensure that the heating rules dealing with membranes and airlocks are reversible. In particular, at each stage, care has been taken to make sure that the information about which molecule the valence originated from is not lost until after the valence is used. In general, if a heating rule precludes some future reaction, then it should be ensured that the heating rule is reversible. This is so that if the machine is repeatedly trying out different sequences of heating rules in order to offer different valences to the environment, it will not get stuck with a large molecule which can't react (like a sediment), unable to offer an appropriate valence to the environment, even though it could have continued had different applications of heating rules been chosen.

As an example, consider the execution of the agent $a.\mathbf{0}|((b.\mathbf{0}|\overline{a}.\overline{b}.\mathbf{0}) \setminus b)$:

$$\begin{array}{lll}
 & \{\!|a.\mathbf{0}|((b.\mathbf{0}|\overline{a}.\overline{b}.\mathbf{0}) \setminus b)|\!\} & \\
\rightharpoonup & \{\!|a.\mathbf{0}, \{\!|b.\mathbf{0}, \overline{a}.\overline{b}.\mathbf{0}|\!\} \setminus b|\!\} & \text{(parallel, restr. memb.)} \\
\rightharpoonup & \{\!|a.\mathbf{0}, \{\!|\overline{a}.\overline{b}.\mathbf{0} \lhd \{\!|b.\mathbf{0}|\!\}|\!\} \setminus b|\!\} & \text{(airlock)} \\
\rightharpoonup & \{\!|a.\mathbf{0}, \{\!|\overline{a}.(\overline{b}.\mathbf{0} \lhd \{\!|b.\mathbf{0}|\!\})|\!\} \setminus b|\!\} & \text{(heavy ion)} \\
\rightharpoondown & \{\!|a.\mathbf{0}, (\overline{a}.(\overline{b}.\mathbf{0} \lhd \{\!|b.\mathbf{0}|\!\})) \setminus b|\!\} & \text{(restr. memb)} \\
\rightharpoonup & \{\!|a.\mathbf{0}, \overline{a}.((\overline{b}.\mathbf{0} \lhd \{\!|b.\mathbf{0}|\!\}) \setminus b)|\!\} & \text{(restr. ion)} \\
\rightarrow & \{\!|\mathbf{0}, (\overline{b}.\mathbf{0} \lhd \{\!|b.\mathbf{0}|\!\}) \setminus b|\!\} & \text{(reaction)} \\
\rightharpoonup & \{\!|\mathbf{0}, \{\!|\overline{b}.\mathbf{0} \lhd \{\!|b.\mathbf{0}|\!\} \setminus \mathrm{b}|\!\} & \text{(restr. memb.)} \\
\rightharpoondown & \{\!|\mathbf{0}, \{\!|\overline{b}.\mathbf{0}, b.\mathbf{0}|\!\} \setminus b|\!\} & \text{(airlock)} \\
\rightarrow & \{\!|\mathbf{0}, \{\!|\mathbf{0}, \mathbf{0}|\!\} \setminus b|\!\} & \text{(reaction)}
\end{array}$$

The ability to isolate a particular ion from a solution and then turn the entire solution into a heavy ion with the same valence as the ion that was isolated, and then reverse the process, reinstating the remainder of the ion, after the valence is used, gives us a method of representing the labelled transitions of CCS. For each label α we define a transition relation $\xrightarrow{\alpha}$ on solutions by: $S \xrightarrow{\alpha} P$ iff there is a molecule m such that $S \overset{*}{\rightleftharpoons} \{\!|\alpha.m|\!\}$ and $\{\!|m|\!\} \overset{*}{\rightleftharpoons} P$. For example

$$\{\!|a.\overline{b}.\mathbf{0}|b.\mathbf{0}|\!\} \xrightarrow{a} \{\!|\overline{b}.\mathbf{0}|b.\mathbf{0}|\!\}$$

with $m = \overline{b}.\mathbf{0} \triangleleft \{\!|b.\mathbf{0}|\!\}$. We can show that, for agents p and q, $p \xrightarrow{\alpha} q$ iff $\{\!|p|\!\} \xrightarrow{\alpha} \{\!|q|\!\}$.

It is also possible to introduce a the prefix τ, and τ transitions in a similar manner (indeed it would be necessary if we were to implement $+$), but this is considerably more complex than the approach taken here. In [BB90] the authors claim that the difficulty in representing τ prefixes and summation is an indication that these concepts are difficult and unnatural to implement in practice.

There are some notable advantages in providing a semantics for a calculus such as CCS by means of a CHAM. In particular various structural equivalences, such as the commutativity and associativity of $|$, are direct results of our heating and cooling rules. It is no longer necessary to derive a suitable congruence relation for agents and use it to prove these results, as it was with the structural operational semantics approach. However proving these results may give us additional confidence in the correctness of the calculus.

In addition the way that transformations are carried out is, in general, that the solution is heated as much as possible in order to exhibit all possible reactions, and then the reactions take place. This avoids the need to intersperse the application of reation rules with large numbers of structural rules, and the application of large numbers of inference rules, so that the execution of reactions is in general simpler.

3.2 Formal definitions of CHAMs

The definition of a CHAM consists of the definitions of its *molecules* and the definition of a set of *transformation rules.* Molecules are terms over some algebra, and a *solution* is a multiset of molecules. In addition a solution can be enclosed in a *membrane* (operator $\{\!| \cdot |\!\}$), in order to form a molecule, allowing solutions to occur within larger molecules.

Transformation rules

Transformation rules take the form

$$m_1, \ldots, m_k \rightarrow m'_1, \ldots, m'_l$$

where the m_i and m'_j are molecules. In fact rules are presented by means of *rule schemata*, with the actual rules being the instances of the schemata in the normal way. In order to prevent *multiset matching* all occurences of subsolutions in the rule schemata must either be *solution variables* of the form S, or singleton multisets of the form $\{\!|m|\!\}$.

It is not adequately explained in [BB90] why multiset matching should be disallowed. However some thought reveals that, in potential instances of the rule schemata, the multisets we are matching against may be infinite, or at least arbitarily large. This means that the process of multiset matching may take an arbitarily long time (or, in the case of infinite multisets, be undecidable). In order for CHAMs to be of use as abstract machines, so that we can use them for such tasks as reasoning about computational complexity, we must be able to find a finite and predetermined bound on the amount of time that any operation in the execution of a CHAM may take.

Meta Laws

The semantics of a CHAM is given by a transition relation on solutions, $\rightarrow$, which is determined by the transformation rules. Formally $\rightarrow$ is the smallest relation satisfying the following laws.

Reaction Law: If there is a rule

$$m_1, \ldots, m_k \rightarrow m'_1, \ldots, m'_l$$

and $M_1, \ldots, M_k$ and $M'_1, \ldots, M'_l$ are instances of $m_1, \ldots, m_k$ and $m'_1, \ldots, m'_l$ respectively, then

$$\{\!|M_1, \ldots, M_k|\!\} \rightarrow \{\!|M'_1, \ldots, M'_l|\!\}$$

(this law describes how a transformation rule is applied).

Chemical Law: If

$$S \rightarrow S'$$

and P is any solution, then

$$S \uplus P \rightarrow S' \uplus P$$

where $\uplus$ denotes multiset union. (this rule states that a transformation can take place in a solution containing molecules which are additional to those involved in the transformation).

Membrane Law: (For this we borrow the context notation of the lambda-caculus). If

$$S \rightarrow S'$$

and $C[]$ is any context, then

$$C[S] \rightarrow C[S']$$

(this says that a subsolution may evolve independently within any molecule).

In addition some, but not all, CHAMs make use of an operator called the *airlock operator*, denoted $\triangleleft$, whose behaviour is governed by the following reversible law:

Airlock Law If S is any solution and m is any molecule, then

$$\{\!|m|\!\} \uplus S \leftrightarrow \{\!|m \triangleleft S|\!\}$$

In [BB90] the authors claim that CHAMs are *intrinsically parallel* in that any number of rules can be applied simultaneously provided they involve different molecules, however the laws given above only describe an interleaving semantics for CHAMs (similar to the structured operational semantics given for CCS). In order to achieve a truly parallel implementation we must change the *chemical law* and the *membrane law* to:

Chemical Law If $S \rightarrow S'$ and either $P \rightarrow P'$ or $P = P'$, then

$$S \uplus P \rightarrow S' \uplus P'$$

Membrane Law If $S_1 \rightarrow S'_1, S_2 \rightarrow S'_2, \ldots, S_n \rightarrow S'_n$, and $C[]$ is an n-hole context, then

$$C[S_1, \ldots, S_n] \rightarrow C[S'_1, \ldots, S'_n]$$

If we are only interested in the expressive power of CHAMs then the interleaving semantics is fine, since we can model any parallel application of rules by a series of single applications of rules (as with CCS). However, if we want to consider such things as computational complexity or the degree of parallelism inherent in a problem, then we must consider the parallel laws. Since these are the sorts of problems to which the use of abstract machine models is well suited, it is important to be able to give parallel laws, and is gratifying that, for CHAMs, these laws seem natural.

Distinguishing Transformation Rules

We divide the transformation rules of a CHAM into three classes: heating rules, cooling rules and reaction rules. These distinctions are purely aesthetic and have no semantic consequence, so that the following descriptions of how to classify the rules of a CHAM should be considered merely to be guidelines.

Heating and *cooling* rules are generally structural rules, allowing the conversion between conceptually equivalent groups of molecules. Heating rules generally break down molecules into forms, possibly consisting of several smaller molecules, which are more ready to take part in reactions, while cooling rules are usually the inverses of heating rules, and allow us to build more complex, "heavier" molecules out of component molecules. In particular, if a heating rule prohibits certain future reactions from taking place, it should be reversible. A solution is said to be *hot* if no further heating rules can be applied to it, and *frozen* if no further cooling rules can be applied to it.

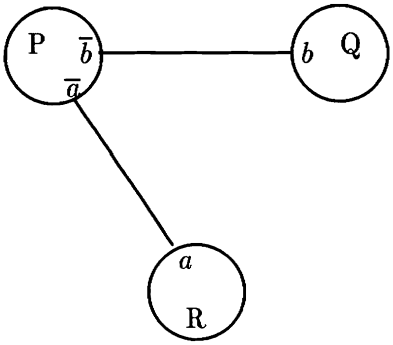
Reaction rules are non-reversible rules, which generally involve several molecules interacting with one another, representing the actual steps of a computation. A molecule which can take part in an application of a reaction rule is called an *ion*, and ions are generally formed by heating molecules as much as possible. A solution is said to be *inert* if no reaction rules are applicable to it or to any solution that may be obtained from it by heating.

4 Mobile Processes

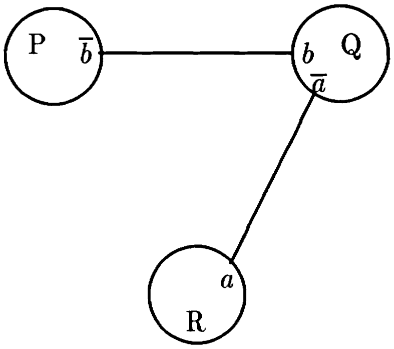
In this section we will look at Milner's π-calculus, or calculus of mobile processes, which was originally introduced in [MPW89] though the version of the calculus and semantics presented here are based on those given in [Mil90].

In [Mil90] Milner describes the π-calculus as a "step towards a canonical calculus for concurrent systems". Presumably the intention here is to say that it is, in some sense, a minimal calculus such that all programs that are conceptually computable by a concurrent system can be encoded in the it in a natural way; so that the π-calculus hopes to play a role simillar to that played by the lambda-calculus for sequential computation. This gives us the idea of having chemical abstract machines as the Turing machines of concurrent systems and the π-calculus as the lambda-calculus of concurrent systems.

The π-calculus is a process algebra, similar to CCS, but is designed to model systems with dynamically changing structure: that is, the links between the components of a system can vary dynamically during the evolution of the system. This property, which is called *mobility*, can at best be modelled indirectly in established process algebras. For example consider the system illustrated bellow. Suppose the process P wishes to pass the value 3 to R via the link a (which is private to P and R).



Now suppose, instead of wanting to pass the value directly to R, P wishes to delegate the task to the process Q. In order to do so P would have to send Q the name of the link to use, a, and the value to pass, 3. Q could then use the channel name that it received in order to communicate with R. Assuming that P would make no further communications with R the structure of the system will have changed into that shown below.



The π-calculus models such things by allowing channel names to be passed as values in communications. In fact the π-calculus combines the concepts of channel names, values and value variables into a single syntactic class: *names.* It is important to note that the π-calculus is not a higher order caluculus: it is only accesses to agents that are being passed in communications, not the agents themselves. The passing of agents as parameters in communications is undesirable since agents would then become replicated, and the replication of agents with state is conceptually difficult and not something we would like to be primitive in our calculus. In addition limiting ourselves to the passing of accesses means that we can allow certain agents only limited access to other agents, and have several agents having different access abilities to some common agent.

It is clear that concurrent systems should be at least as expressive as sequential systems, and so a canonical calculus for concurrent systems should be able to encode the lambda calculus in a natural manner. While it is true that, for any formalism in which we can simulate a universal Turing machine (including CCS), we can always construct an *interpreter* for the lambda-calculus, that is a program or expression which evaluates syntactic representations of lambda expressions, it is not necessarily possible to find a straight forward *translation* from lambda expressions to expressions of that formalism. Such a translation would have to, in some sense, preserve and be consistent with the semantics of the lambda calculus. In [Mil90] it is shown that there is such a translation into the π-calculus, though we will not give the details here. This result is perhaps surprising since the π-calculus is not higher order, unlike the lambda calculus where lambda expressions (which are interpreted as agents) can be passed as arguments to functions and bound to variables. One might expect that we would have to extend our process calculi to be higher order in order to make them as expressive as the lambda calculus, as was done in [Bou89], but the π-calculus is limited to passing simple *names*.

On reflection, perhaps it is not so surprising that we can express the lambda calculus in such a formalism: after all we can construct natural implementations of functional and other higher order programming languages on conventional machines, working on the basis of passing simple data items between registers and carrying out simple operations on them, where these data items function either as pointers to the code of functions or other complex data structures, or as values, instead of passing the functions and complex data structures themselves. Perhaps the most valuable aspect of the π-calculus is that it gives us an abstract, mathematical way to model this kind of computing, and so allows us to reason about such implementations in a formal way.

4.1 The π-calculus

To start with we assume as primitive a set, $\mathcal{N}$, of *names*, ranged over by $x, y, z, \ldots$. The calculus consists a set, $\mathcal{P}$, of *agents*, which we will use $P, Q, R \ldots$ to range over. We will also assume a set of *agent variables* ranged over by $X, Y, \ldots$.

Our first class of agents are *guarded agents* which have the form $g.P$, where g has one of the forms

$$g ::= \overline{x}y \mid x(y)$$

and is called a *guard*. Intuitively the agent $\overline{x}y.P$ outputs the name y on the *channel* x and then proceeds to execute P, while the agent $x(y).P$ receives a name z on the channel x and then executes $P\{z/y\}$. Here $P\{z/y\}$ denotes the result of replacing all *free* occurrences of y in P by z, and renaming any bound names in P as necessary in order to prevent any of the newly introduced z's from becoming bound.

The second class of agents are *parallel compositions*. We write $P|Q$ for the parallel composition of P and Q, which means that the agents P and Q are running side by side and able to interact with eachother as well as the environment. For example in the parallel composition

$$\overline{x}v.P|x(y).Q$$

the two agents can interact forming the parallel composition agent $P|Q\{v/y\}$.

In addition we introduce an agent *inaction*, denoted $\mathbf{0}$, which has no communication ability and is a unit for parallel composition.

The next class of agents are recursive agents formed using the fixed point operators **fix**. As for CCS, these are agents of the form $\mathbf{fix}_j(\tilde{X} = \tilde{E})$, and they behave in a similar manner. In [Mil90] a simpler but more limited *replication operator*, "!", was used in place of fixed point operators. It was stated that this operator was powerful enough to encode most desirable forms of recursion, and was shown that, even with this operator, the calculus was general enough to encode the lambda calculus. We will, however, present the calculus with the more general fixed point operators so as to try to make our presentation as similar to that of CCS as possible.

Our final category of agents are *restricted agents*, which have the form $(x)P$. The prefix (x) is called a *restriction* and means that P can't use the channel x except for internal actions. For example no interaction is possible between the two agents of the parallel composition

$$\overline{x}v.P \mid (x)(x(y).Q)$$

while an internal action is still possible in the agent

$$(x)(\overline{x}v.P|x(y).Q)$$

An occurence of a name x is said to be *bound* if it occurs within an expression of the form $(x)P$ or $y(x).P$, and is said to be free otherwise. We write $fn(P)$ to denote the set of all free names occuring in P.

While an agent of the form $(x)P$ may not transmit or recieve any names on the channel x, the question arises of what happens if P tries to transmit or receive the name x to or from some other agent on some other channel. In the case of some other agent sending the name x

to $(x)P$, we can use alpha-conversion to replace $(x)P$ with $(z)(P\{z/x\})$ for some new name z, and then proceed as normal. If P wishes to transmit the name x on some other channel then the situation is more complex and something called *scope extrusion* occurs, which means that the scope of the restriction (x) is expanded to include the agent to which the message is being sent, and the message is then sent. For example the agent

$$x(y).P \mid (z)(\overline{x}z.Q)$$

can carry out an internal action transforming to

$$(z)(P\{z/x\}|Q)$$

provided $z \notin fn(P)$. This phenomenon seems a little strange at first, and is perhaps best explained by likening it to dynamic binding in functional programming languages. In the example above, any occurences of z outside the restriction will have a different meaning (that is, refer to a different link) to the meaning of z in Q. It follows that if we want the meaning of the z's substituted into P to be the same as those in Q we must expand the scope of the restriction to include P. It should be noted that, if the name z already occurs freely in P, we can use alpha-conversion to replace z by some other name in $(z)(\overline{x}z.Q)$.

We will often use the abreviation $\overline{x}(y).P$ for $(y)(\overline{x}y.P)$. This can be thought of as simultaneously creating and sending a new private name y.

We have now completed our definition of the syntax of the π-calculus (the calculus originally proposed in [MPW89] was richer than the version presented here). The similarity of this calculus to CCS with value passing is notable: the most significant difference being that the concepts of channel names, values and variables are unified into a single syntactic category: *names*.

4.2 The Semantics of the π-Calculus

We will present the semantics of the π-calculus by means of a CHAM. This differs from the "CHAM-inspired" semantics given in [Mil90] where the author first defined a structural congruence relation, $\equiv$, on agents, and then defined a transition relation, $\rightarrow$, on the equivalence classes of agents under the congruence. The heating and cooling rules of our CHAM can be thought of as encoding the structural congruence relation, while the reaction rules encode the transition relation. The only difference is that we break down the parallel compositions to form multisets of agents, rather than leaving them intact.

In order to prevent us from worrying about bound name clashes we will consider any two alpha-convertable agents to be equivalent, and will assume that, where some agents $P_1, \ldots, P_n$

appear in some context, their bound names are chosen so as not to coincide with any of their free names.

The molecules of the CHAM are formed using the constructors of the π-calculus, and, in addition, the membrane and airlock operators, so that the molecules form a superset of the π-calculus terms. It is important to ensure however that a solution can always be heated and then frozen in such a way as to form a single π-calculus term.

Heating and Cooling Rules

Our first heating/cooling rules are for breaking down parallel compositions and unfolding fixed point operators, and are similar to those for CCS.

parallel

$$P|Q \rightleftharpoons P,Q$$

fix

$$\mathbf{fix}_j(\widetilde{X} = \widetilde{E}) \rightharpoonup E_j\{\widetilde{\mathbf{fix}}(\widetilde{X} = \widetilde{E})/\widetilde{X}\}$$

Our next set of rules are for restrictions. First we need some rules to encode certain structural equivalences.

restr. order

$$(x)(y)P \rightharpoonup (y)(x)P$$

restr. cleanup

$$(x)P \rightharpoonup P \text{ if } x \notin fn(P)$$

We have rules, similar to those for CCS, for moving valencies outside of restrictions, and for introducing membranes and using airlocks (the rule for introducing airlocks is a meta-rule for CHAMs and so is not stated here).

restr. ion 1

$$(x)(y(z).P) \rightleftharpoons y(z).((x)P) \text{ where } x \neq y \text{ and } x \neq z$$

restr. ion 2

$$(x)(\overline{y}z.P) \rightleftharpoons \overline{y}z.((x)P) \text{ where } x \neq y \text{ and } x \neq z$$

restr. membrane

$$(x)P \rightleftharpoons (x)\{\!|P|\!\}$$

heavy ion

$$g.P \lhd A \rightleftharpoons g.(P \lhd A)$$

In addition, in order to allow for scope extrusion, we allow molecules to migrate into a restriction provided they do not contain free variables which would be bound by the restriction:

restr. scope

$$P, (x)A \rightleftharpoons (x)(\{\!|P|\!\} \uplus A) \text{ provided } x \notin fn(P)$$

Finally we will need some cleanup rules to get rid of excess **0**'s and empty membranes:

inaction cleanup

$$P, \mathbf{0} \rightharpoonup P$$

membrane cleanup

$$\{\!|\,|\!\} \rightharpoonup$$

Internal and Observable Transition Rules

The internal actions of agents are represented by the reaction rules for the CHAM. There is only one such rule which is straightforward, scope extrusion being handled by the heating/cooling rules:

reaction

$$x(y).P, \overline{x}z.Q \rightarrow P\{z/y\}, Q$$

We can then define the internal transition relation on agents, $\rightarrow$, by $P \rightarrow Q$ iff $\{\!|P|\!\} \overset{*}{\rightleftharpoons}\rightarrow\overset{*}{\rightleftharpoons} \{\!|Q|\!\}$.

There are three forms of *observable actions*: *input* $(x(y))$, *output* $(\overline{x}y)$ and *restricted output* $(\overline{x}(y))$. We use α to range over actions, so

$$\alpha ::= x(y) \mid \overline{x}y \mid \overline{x}(y)$$

For each action α we define a transition relation $\xrightarrow{\alpha}$ on agents by $P \xrightarrow{\alpha} Q$ iff, for some molecule m, $\{\!|P|\!\} \overset{*}{\rightleftharpoons} \{\!|\alpha.m|\!\}$ and $\{\!|m|\!\} \overset{*}{\rightleftharpoons} \{\!|Q|\!\}$. For example

$$(z)(\overline{x}y.P|Q) \xrightarrow{\overline{x}y} (z)(P|Q)$$

with $m = (z)(P \lhd \{\!|Q|\!\})$.

4.3 Examples

In the remainder of this section we will give some examples of how the π-calculus can be used to implement a number of processes that are difficult or impossible to model in established process algebras such as CCS.

Passing Series of Parameters

Suppose we had a collection of processes, $P_1, \ldots, P_m$, such that each P_i wished to send a pair of names, say u_i and v_i, to one of a collection of processes $Q_1, \ldots, Q_n$. We might try to implement this by a system $(a)(P_1| \ldots |P_m|Q_1| \ldots |Q_n)$, where

$$\begin{array}{rcll} P_i & = & \overline{a}u_i.\overline{a}v_i.P'_i & \text{for } i = 1, \ldots, m \\ Q_j & = & a(x).a(y).Q'_j & \text{for } j = 1, \ldots, n \end{array}$$

However this wouldn't work since, firstly, a P_i might send u_i to some Q_j and v_i to another $Q_{j'}$, and secondly a Q_j might receive u_i from P_i and then be sent a $u_{i'}$ or $v_{i'}$ from some other $P_{i'}$ before it has a chance to receive v_i from P_i. The problems stem from the fact that we are using two distinct communications to convey the information; if we could find some way to transmit the information using effectively a single, atomic communication then this would solve the problem. In the π-calculus we can use a single communication in order to set up a private linc between a P_i and a Q_j and then use that linc for the remainder of the communication. The P_i's and Q_j's would now have the form:

$$\begin{array}{rcll} P_i & = & \overline{a}(w).\overline{x}u_i.\overline{w}v_i.P'_i & \text{for } i = 1, \ldots, m \\ Q_j & = & a(z).z(x).z(y).Q'_j & \text{for } j = 1, \ldots, n \end{array}$$

Clearly this approach is not possible in CCS. Further, while it is possible to solve this sort problem for some fixed, predetermined number of sending and receiving processes, CCS does not offer any way to construct a general solution for an arbitary number of processes.

Simulating Higher Order Computation

While the π-calculus is not a higher order calculus, so that we cannot pass processes themselves in communications or substitute them for names, we can achieve similar effects by preffixing processes by *triggers* and then passing the triggers as parameters in communications.

We will need some new notation: first we assume a special name, ϵ, which will never be bound. We write $\overline{x}.P$ as shorthand for $\overline{x}\epsilon.P$ and $x.P$ as shorthand for $x(y).P$ where $y \notin fn(P)$. So the prefixes $x.$ and $\overline{x}.$ represent communications on the channel x which do not convey any additional information, similar to the simple *handshakes* we used in our basic version of CCS.

Now let us consider the agent

$$\mathit{Exec}(x) \;=\; x(y).\overline{y}.\mathbf{0}$$

(here, and in the remainder of our examples, we will use *agent constants* with possibly recursive definitions in a similar manner to those described for CCS). If we prefix a process P by a private "trigger", $z.$, and then send z to $\mathit{Exec}(x)$ on x, then the resulting system behaves like P:

$$\begin{array}{l} (x)((z)(\overline{x}z.\mathbf{0}|z.P)|\mathit{Exec}(x)) \\ \rightarrow\; (x)(z)(z.P|\overline{z}.\mathbf{0}) \\ \rightarrow\; (x)(z)P \;\equiv\; P \end{array}$$

What is happening here is that we are passing the *trigger* z of P to the term $\mathit{Exec}(x)$ which then *fires* z, thus executing P. In some hypothetical higher order calculus we might instead have

$$\mathit{Exec}'(x) \;=\; x(p).p$$

and written the system as

$$(x)(\overline{x}(P).\mathbf{0}|\mathit{Exec}'(x))$$

This rather simple example does not really demonstrate the utility of this method, so let's look at a slightly more complex example. Suppose we had a system consisting of a process which transmits the process P on channel x and then executes the process Q, and another process which receives a process on the channel x and executes it in parallel with R. In our hypothetical higher order calculus we would write this system as

$$\begin{array}{l} (x)(\overline{x}P.Q|x(p).(p|R)) \\ \rightarrow\; (x)(Q|(P|R)) \end{array}$$

We could achieve a similar effect using tigger passing in the π-calculus by

$$(x)((z)(\overline{x}z.Q|z.P)|x(y)(\overline{y}.\mathbf{0}|R))$$

As we stated earlier, it is not desirable to have agent passing as primitive in our calculus because of the problems caused by potential replication of agents, though there are many phenomenon which can be expressed most naturally by using such higher order devices. Using names as triggers or *pointers* to agents and passing just names allows us to achieve most of the expressivity of a higher order calculus while avoiding passing agents themselves.

Representing Complex Data Structures

For our final example we will show how π-calculus agents can be used to represent complex data structures. In particular we will show how to model lists and operations on lists. In addition this example will give a more substantive illustration of the techniques described in the previous two examples.

We will introduce some additional abbreviations for compound prefixes:

$$\begin{array}{rcl} \overline{x}y_1y_2\ldots y_n & \text{means} & \overline{x}y_1.\overline{x}y_2.\ldots.\overline{x}y_n \\ x(y_1)(y_2)\ldots(y_n) & \text{means} & x(y_1).x(y_2).\ldots.x(y_n) \end{array}$$

and, as with CCS, we will omit the finishing inaction operators, $\mathbf{0}$, from agents.

We will assume a fairly standard formalism for lists using the nullary constructor *Nil* and the binary constructor *Cons*, so the list $[a, b, c]$ would be represented as

$$\mathit{Cons}(a, \mathit{Cons}(b, \mathit{Cons}(c, \mathit{Nil})))$$

We will take the elements stored in lists to be names. Each list-representing agent will have a name, which is said to *point to* the agent. We will write $[\![l]\!](x)$ for the agent representing the list l and pointed to by x.

We will assume two special names $\textsc{Nil}$ and $\textsc{Cons}$ which we will not use as communication channels. Our first attempt to implement lists will work as follows: an agent representing the list *Nil* will output $\textsc{Nil}$ on its pointer; an agent representing some other list will first output $\textsc{Cons}$, then the first item in the list, and then a (private) name pointing to an agent representing the remainder of the list. So the translation of lists into agents is defined by:

$$\begin{array}{rcl} [\![\mathit{Nil}]\!](x) & \stackrel{\text{def}}{=} & \overline{x}\textsc{Nil} \\ [\![\mathit{Cons}(v, l)]\!](x) & \stackrel{\text{def}}{=} & (y)(\overline{x}\ \textsc{Cons}\ v\ y \mid [\![l]\!](y)) \end{array}$$

There are some problems with this representation of list. The first is that it's *ephemeral*: that is the agents do not continue to exist after the list is read once. We would like a *persistant* representation of lists, so that they can be read many times.

The second problem occurs if more than one agent has access to a list, and two or more agents attempt to access the list simultaneously. In this case the two or more agents would both be taking their inputs from the same channel leading to what can best be described as "a muddle". We use the same approach as in our first example to avoid this problem: suppose the list-agent we wish to read is pointed to by the name x; first pass a new name, say y, to the list agent on channel x; the list-agent then uses the name y for the remainder of the communication, first outputting its type, and then, if necessary, the first element of the list and a pointer to the remainder of the list. Note that, as soon as a name has been sent to the list agent on channel x, the list agent is ready to receive another name on x even if the first read is not yet finished. The definition of this implementation of lists is:

$$\begin{aligned} \llbracket \mathit{Nil} \rrbracket(x) &\overset{\text{def}}{=} x(y).(\overline{y}\textsc{Nil} \mid \llbracket \mathit{Nil} \rrbracket(x)) \\ \llbracket \mathit{Cons}(v,l) \rrbracket(x) &\overset{\text{def}}{=} x(y).((z)(\overline{y}\ \textsc{Cons}\ v\ z \mid \llbracket l \rrbracket(z)) \mid \llbracket \mathit{Cons}(v,l) \rrbracket(x)) \end{aligned}$$

We would like to describe an agent which appends two lists. The protocol for using the append agent will be as follows: suppose the agent is pointed to by x; first use the channel x to set up a new, private channel y to use for the remainder of the communication; then send the pointers to the list agents being appended to the append agent on the channel y; the append agent then outputs a name on y pointing to the appended list. The append agent works by first reading from the first list; then, if it is a *Nil* list, it outputs a (new) pointer to the second list; otherwise it creates a new list with the same first element as the first list, and the remainder the result of appending the second list to the remainder of first list, and outputs a pointer to this list.

There is a problem here in that the execution of the append agent is governed by what names it receives on certain input channels. We need some kind of conditional operators in order to express this. We could introduce an implementation of Boolean values simillar to that used for the lambda calculus (see [Bar81]). However we will instead embed the necessary apparatus into our definition of list agents.

For our third attempt at defining list agents we will drop the use of the names Nil and Cons. Instead we will take the approach that, after receiving the name of a channel for further communication, the list agents will then receive two names on that channel. If the agent represents a *Nil* list it will then output the first of the two names, otherwise it will output the second of the two names followed by the first element of the list and a pointer to the remainder of the list.

$$\begin{aligned} \llbracket \mathit{Nil} \rrbracket(x) &\overset{\text{def}}{=} x(y)(y(u)(v).\overline{y}u \mid \llbracket \mathit{Nil} \rrbracket(x)) \\ \llbracket \mathit{Cons}(a,l) \rrbracket(x) &\overset{\text{def}}{=} x(y)((z)(y(u)(v).\overline{y}vaz \mid \llbracket l \rrbracket(z)) \mid \llbracket \mathit{Cons}(a,l) \rrbracket(x)) \end{aligned}$$

The point here is that we are allowing the agent that accesses the list structure to instantiate the names that are used to distinguish the list types, possibly to private names. We could implement an agent that accessed a list pointed to by the name x and then executed either the agent P_1 or P_2 depending on whether x pointed to an empty list or not, as

$$(y)(u)(v)(\overline{x}y.\overline{y}uv.y(w).\overline{w} \mid (u.P_1|v.P_2))$$

We could not have a choice operator like this using non-private names (such as NIL and CONS) because of the possibility of other agents making use of those names and interfering.

In particular we can implement our append agent, pointed to by x, by:

$$\begin{aligned}
\mathit{Append}(x) &\overset{\text{def}}{=} x(y).(y(z_1)(z_2).(w)(\overline{y}w|\mathit{App}(z_1,z_2,w)) \mid \mathit{Append}(x)) \\
\mathit{App}(z_1,z_2,w) &\overset{\text{def}}{=} (u)(n)(c)(\overline{z_1}u.\overline{u}nc.u(t).\overline{t} \mid (n.\mathit{Copy}(w,z_2) \mid \\
&\qquad c.u(a)(z_3).(v)(\mathit{Cons}(a,v,w)|\mathit{App}(z_3,z_2,v)))) \\
\mathit{Copy}(x,y) &\overset{\text{def}}{=} x(z).(\overline{y}z|\mathit{Copy}(x,y)) \\
\mathit{Cons}(a,v,w) &\overset{\text{def}}{=} w(x).x(n)(c).\overline{x}cav
\end{aligned}$$

Here $\mathit{App}(z_1,z_2,w)$ forms a list pointed to by w consisting of (the list pointed to by) z_2 appended to z_1; $\mathit{Copy}(x,y)$ copies the input on x to y, thus if y points to a list then x points to the same list; and $\mathit{Cons}(a,v,w)$ forms a list agenti, pointed to by w, with first element a and remainder pointed to by v.

4.4 Limitations of the π-calculus

As we can see from the example above, the π-calculus can express lists, conditional operators and, since they can be expressed by a list-like structure, natural numbers. However such constructions are somewhat complex and unnatural, and so the question arises of whether the calculus can be extended in such a way as to make such representations easier. We can see that it is not sufficient just to add new names to represent various values, since we could not construct the necessary operators for manipulating those names in the calculus. The version of the π-calculus originally presented in [MPW89] was better in this respect, since it was equipped with summation and conditional guard operators. (Summation was defined in a similar manner to that in CCS, while conditional guards were guards of the form $[x = y]$, such that an agent $[x = x].P$ would behave like P, while $[x = y].P$ would reduce to $\mathbf{0}$ if $x \not\equiv y$). However summation and τ-actions produce semantic difficulties, and so it might be worth investigating some other external choice operator. Even with summation and conditional guards we could not build the infinite functions and operators necessary to manipulate the natural numbers if introduced as names. The question of how best to extend the calculus in order to make it more useful therefore remains open.

5 Conclusions

The use of formal models for concurrent systems is a comparatively recent development and it remains to be seen whether they will yield the same benefits that they have for sequential programming. It is still very much an open question what features a model should support in order to make itself useful, and how these features are best represented.

Of the models discussed here, only CCS has demonstrated its usefulness in various real world examples. The ability to restrict ourselves to an extremely small and simple subset of the calculus for theoretical purposes, and to then add various useful extensions for practical use, makes it an attractive proposition.

The chemical abstract machine provides an intuitive and inherently parallel way of modelling concurrent systems. These models are easy to reason about due to their lack of dependence on architectural constraints, and are well suited to providing semantics for languages and calculi for concurrency. In addition they may potentially help us to develop a theory of computational complexity for concurrent systems, similar to that established for sequential systems.

The π-calculus provides the ability to express mobility which may well be important in some systems, and offers a very general and expressive basis for a calculus for concurrent systems. It does this without resorting to higher order constructs, and instead exhibits a formal model for the use of pointers to objects in computation. However, in its basic form, it is not rich enough to be useful for specificying and developing real systems, and it remains to be seen how best to extend it in order to form a practical calculus.

References

[Bar81] H.P. Barrendregt. *The Lambda Calculus. Studies in Logic*, North-Holland, 1981.

[BB90] G. Berry and G Boudol. The chemical abstract machine. In *Proc. 17th Annual Symposeum on Principles of Programming Languages*, 1990.

[Bou89] G. Boudol. Towards a lambda calculus for concurrent and communicating systems. In *TAPSOFT 1989*, Springer Verlag, 1989.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

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

[Mil90] R. Milner. *Functions as Processes.* Technical Report, INRIA, 1990.

[MPW89] R. Milner, J. Parrow, and D. Walker. *A Calculus of Mobile Processes, Parts 1 and 2*. Technical Report ECS-LFCS-89-85 and -86, Laboratory for the Foundations of Computer Science, Edinburgh University, 1989.