Distributed CCS

Padmanabhan Krishnan*†
Department of Computer Science
Ny Munkegade Building 540
Aarhus University
DK 8000, Aarhus C Denmark
E-mail: paddy@daimi.aau.dk

July 1991

Abstract

In this paper we describe a technique to extend a process language such as CCS which does not model many aspects of distributed computation to one which does. The idea is to use a concept of location which represents a virtual node. Processes at different locations can evolve independently. Furthermore, communication between processes at different locations occurs via explicit message passing. We extend CCS with locations and message passing primitives and present its operational semantics. We show that the equivalences induced by the new semantics and its properties are similar to the equivalences in CCS. We also show how the semantics of configuration and routing can be handled.

1 Introduction

A number of different types of parallel and distributed machines have been built. These include vector machines [Rus78] data driven/demand driven ma-

*The author acknowledges support from the Danish Research Council
†To appear at CONCUR-91

chines [TBH82], shared memory systems (like the Sequent) and distributed memory systems ranging from a network of work stations to well structured architectures like the hypercube. More parameters such as heterogeneity, reliability for fault-tolerance etc. add to the variation. Despite the variety, a common factor is the presence of multiple processing elements and interconnection between them. For such machines to be useful, applications must use the multiple processors, which requires the decomposition of the application into partitions which can be executed in parallel. Any co-operation between the partitions (synchronization, communication etc.) must utilize the interconnections.

Given the wide variety of architectures, it is not surprising that a variety of programming languages have been proposed; each addressing a subset of the architectures. Many languages are based on the message passing between a set of parallel processors [INM88]. Other approaches include using a distributed data-structure (tuple space [Gel85]), using the parallelism available in functional programs, parallel logic languages [CG83]. [Mar89] gives an detailed description of the various approaches. Our aim is to develop a calculus in which the following can be studied: 1) Expressing parallelism, 2) Describing the effect of parallelism and 3) Mapping of the expressed parallelism to a physical environment.

Labeled transition systems have been used to describe concurrent systems [Mil80]. The syntax of CCS is simple and yet powerful enough to express parallelism, non-determinism and synchronization. However, it does not address issues to related to modeling the physical environment. Also the operational semantics of CCS reduces parallelism to non-determinism (or interleaved execution of actions: the expansion theorem for CCS). This is due to the presence of only one observer with the power to observe one action at a time. We provide a semantics for CCS which is "distributed". Given that there is no uniformity in the nature of distributed architecture, one needs to define "distributed". For that we consider the natural language meaning of the terms concurrent, distributed and parallelism. Concurrent is defined as "happening together in time or place"; distributed as "scattered or spread out over a surface" and parallelism as "without interactive causal relationship". In CCS concurrent has the interpretation of place or processor. Thus it is not surprising that parallelism, given one place, reduces to non-determinism. Our goal is to "scatter parallelism over a surface".

Given that parallelism is to be mapped over a surface, the question arises as whether to allow the process to contain some information regarding the mapping. If so, another level of abstraction can be created. The mapping information available in the process can be considered to a logical representation of the surface. One then studies the effect of mapping a logical surface to a physical surface. In distributed computation, one considers the representation of distribution also called binding and the mapping of binding to the physical surface, i.e., configuring. This view is indeed taken by the advocates of the virtual node approach to distributed computing [CM87]. Thus there are two stages in program construction, viz., 1) distributed surface with logical binding and 2) the configuration of distributed memory.

In section 2 a semantics for distributed memory systems is developed and in section 3 some of the issues related to program development are discussed. In section 4 a few examples are presented.

2 Distributed CCS

Distributed systems do not share memory and hence the location of a process in the system is important. Given a number of processors, a process can only use the processor associated with the memory unless it is explicitly relocated. As both binding and configuration are to be considered, an extension to CCS is necessary. A notion of location has been introduced in CCS [KHCB91] to study the distributed nature of processes. Their primary concern is the logical construction of processes without considering the architecture the process is executing on. The idea of location has also been used in other languages [Hud86]. We use the same syntax as in [KHCB91] but give a different semantics.

In the semantics for CCS, any two processes could synchronize. This cannot be permitted in the distributed case. A local transition depending on behavior at a remote site is unrealistic. Consider, for example, the CCS process $(\ (a+b) \mid (\bar{a}+\bar{b})\ ) \setminus \{a, b\}$. If the two parallel processes are physically distributed, the decision to execute a(or b) has to be taken in co-ordination with the $\bar{a}$(or $\bar{b}$). If a general CCS process is to be executed on a physically distributed environment a non-trivial protocol to effect these decisions is essential. Therefore, deriving a distributed implementation from the operational semantics is not straight forward.

For the distributed case, a protocol based on the send/receive primitives is more appropriate. This then requires a definition of buffers, where the messages sent are stored before actually 'used' by the process. Towards this, we assume special actions such as $\overset{<l,a>}{\rightsquigarrow}$, which indicates the sending of action a to location l. This message is buffered by a process on location n by creating a process which can engage only in action 'a'. This can be generalized to communicating an arbitrary process. As we do not limit the number of parallel terms in a process, unbounded buffering is modeled.

Definition: 1 *Define a finite set of locations $\mathcal{L}$, a set of local actions Lact (represented by a,b), a set of sending actions Sact (represented by $\overset{<l,a>}{\rightsquigarrow}$). The cardinality of Sact is less than or equal to the cardinality of $\mathcal{L}\times$ Lact. Let τ represent synchronization and δ represent idling (of a location). The set of actions is $Act = Lact \cup Sact \cup \{\tau, \delta\}$.*

The basic syntax for our language (with μ being any action, l any location, and H a subset of Act) is as follows:

$$\text{P} = \text{Nil} \mid \mu \text{ ;P} \mid (\text{P} \mid \text{P}) \mid \text{P} + \text{P} \mid (1 :: \text{P}) \mid (\text{P} \backslash \text{H}) \mid \text{rec}(\tilde{X}{:}\tilde{E})$$

Nil is a process which can exhibit no further action. μ ;P represents action prefix; i.e., after executing μ the processes behaves as P. (P | Q) is the parallel combination of P and Q, while P+Q represents choice between P and Q. (l :: P) restricts the execution of P to location l, (P \H) restricts the behavior of P to those actions not in H. rec($\tilde{X}$:$\tilde{E}$) is recursive process. We place the usual restrictions that terms must be closed and the recursion guarded.

Given that a process has location information, an observation now is a pair consisting of action and location. As processes on different locations can execute in parallel behavior is characterized by a set of observations with the restriction that processes can exhibit multiple actions if they are at different locations. Process can synchronize if the are at the same location. A structural operational semantics [Plo81] is defined as an generalization of the rules for CCS. We assume that the following black-box is a model of a distributed system which runs a process given a set of processing elements (locations). There is a step line which when toggled advances each processing element by one step. The observer first toggles the machine and then notes the behavior on each of the locations (which may appear at different times

with respect to a global clock) and the process continues. This is similar to the step semantics developed for Petri Nets [vGV87].

The following definitions are used in the operational semantics which is defined in figure 1.

Definition: 2 *Let $\mathcal{O}_L$ represent the function space from $\mathcal{L} \rightarrow Act$. This is used by the operational semantics to identify the action observed at any location.*

Definition: 3 *Define the transition relation $\longrightarrow_D \subseteq$ Processes $\times \mathcal{O}_L \times$ Processes.*

Also define two projection functions Location and Action, which return the location/action part of the observation.

Definition: 4 *Define S1+S2 as follows: S1+S2 = S where*

$$S(l) = \begin{cases} S1(l) & \textit{if} \quad S2(l) = \delta \\ S2(l) & \textit{if} \quad S1(l) = \delta \\ \tau & \textit{if} \quad S1(l) = \overline{S2(l)} \end{cases}$$

The idea is that the two behaviors (Sl and S2) can be exhibited in one step if the non-idling actions are on different locations. If a location can exhibit two actions, they must be synchronizable and the combined behavior will exhibit τ.

The distributed operational semantics is defined in figure 1. We label the transition with only the non-idle observations. All locations not present on the label are assumed to exhibit the idle action (δ).

An informal explanation of the semantics follows. A process with no location can be executed at any location. However, the remainder of the process is constrained to executed at that location. This can be thought of as the initial loading of a process. The sending of a message to a particular location results in creating a process at that location which can only engage in the action contained in the message. Furthermore, the message passing is visible to the observer. As we have not *named* processes, one cannot send a message to a particular process. This can be simulated by ensuring only 'one process' at a location, and the result of sending messages to it as the buffers. A process already assigned a location can exhibit actions on that location.

Lact Prefix	$a; P \xrightarrow{\{<l,a>\}}_D l :: P \forall\ l \in \mathcal{L}$
Sact Prefix	$\overset{<m,a>}{\rightsquigarrow}; P \xrightarrow{\{<l,\overset{\{<m,a>\}}{\rightsquigarrow}>\}}_D (l :: P) \mid (m :: a) \forall\ l \in \mathcal{L}$
Location	$\dfrac{\mathrm{P} \xrightarrow{\{<l,\mu>\}}_D \mathrm{P}'}{\mathrm{l} :: \mathrm{P} \xrightarrow{\{<l,\mu>\}}_D \mathrm{P}'}$
Interleaving (Asynchronous Evolution)	$\dfrac{\mathrm{P} \xrightarrow{S}_D \mathrm{P}'}{\begin{array}{c}\mathrm{P} \mid \mathrm{Q} \xrightarrow{S}_D \mathrm{P}' \mid \mathrm{Q} \\ \mathrm{Q} \mid \mathrm{P} \xrightarrow{S}_D \mathrm{Q} \mid \mathrm{P}'\end{array}}$
Parallelism	$\dfrac{\mathrm{P} \xrightarrow{S1}_D \mathrm{P}', \mathrm{Q} \xrightarrow{S2}_D \mathrm{Q}', \mathrm{S} = \mathrm{S1} + \mathrm{S2}}{\begin{array}{c}\mathrm{P} \mid \mathrm{Q} \xrightarrow{S}_D \mathrm{P}' \mid \mathrm{Q}' \\ \mathrm{Q} \mid \mathrm{P} \xrightarrow{S}_D \mathrm{Q}' \mid \mathrm{P}'\end{array}}$
Hiding	$\dfrac{\begin{array}{c}\mathrm{P} \xrightarrow{S}_D \mathrm{P}', \\ \mathrm{Action(S)} \cap \mathrm{H} = \emptyset \text{ and } \mathrm{Action(S)} \cap \overline{H} = \emptyset\end{array}}{\mathrm{P} \setminus \mathrm{H} \xrightarrow{S}_D \mathrm{P}' \setminus \mathrm{H}}$
Recursion	$\dfrac{\mathrm{E}_i(\mathrm{rec}\ \tilde{X} : \tilde{E}/X) \xrightarrow{S}_D \mathrm{P}'}{\mathrm{rec}_i \mathrm{X} : \mathrm{E} \xrightarrow{S}_D \mathrm{P}'}$

Figure 1: Operational Semantics

In the operational semantics of l :: P we do not restrict P' to location l. The restriction is introduced by the transition of P. Furthermore, P could have evolved to P' via a send (say to location m) in which case P' has the structure: l :: P" | m :: a. Then l :: P' cannot exhibit a.

In this paper we do not consider relocating the processes to the appropriate location. Processes can evolve 'independently' of other processes modeling asynchronous behavior (the first | rule). The parallel combination of two processes exhibit 'true concurrency' if their locations are disjoint. Processes can also synchronize in the usual way. The hiding operator restricts the actions a process can exhibit. The restriction applies to all locations.

2.1 Bisimulation

In this section we establish a equivalence relation between processes based on the well known notion of bisimulation [Par81]. The use of bisimulation is to identify processes which cannot be distinguished by an observer. For this draft we assume that the observer is equipped to observe a single action (including synchronization, idling) at every location.

Definition: 5 *Define a relation on processes $\mathcal{R}$ to be a bisimulation such that if $P\ \mathcal{R}\ Q$ then*

$$\mathrm{P} \xrightarrow{S}_D \mathrm{P}' \Rightarrow \exists\ Q' \textit{ such that } \mathrm{Q} \xrightarrow{S}_D \mathrm{Q}' \textit{ and } P'\ \mathcal{R}\ Q' \textit{ and}$$
$$\mathrm{Q} \xrightarrow{S}_D \mathrm{Q}' \Rightarrow \exists\ P' \textit{ such that } \mathrm{P} \xrightarrow{S}_D \mathrm{P}' \textit{ and } Q'\ \mathcal{R}\ P'$$

Definition: 6 $\sim_D = \bigcup\{\mathcal{R} \textit{ such that } \mathcal{R} \textit{ is a bisimulation}\}$

Proposition 1 $\sim_D$ *is a congruence*
$P + Q \sim_D Q + P$
$(P + Q) + R \sim_D P + (Q + R)$
$P + P \sim_D P$
$P \mid Q \sim_D Q \mid P$
$(P \mid Q) \mid R \sim_D P \mid (Q \mid R)$
$P \mid Nil \sim_D P$
$l :: (P \mid Q) \sim_D (l :: P) \mid (l :: Q)$
$l :: (P + Q) \sim_D (l :: P) + (l :: Q)$
$l :: (P \backslash H) \sum_D (l :: P) \backslash H$

Proposition 2 *If* $(l1 \neq l2)$, $l1 :: (l2 :: P) \sim_D Nil$.

Proposition 3 *If* P *and* Q *are processes without location,* $(l :: P) \sim_D (l :: Q)$ *implies* $P \sim_D Q$.

The above proposition does not hold if P and Q are allowed to contain locations. For example, if P = l :: nil and Q = m :: a, l :: P $\sim_D$ l :: Q but P is not $\sim_D$ Q.

As we have assumed only one processor (or observer) at any given location, the expansion theorem for CCS translates directly to the location case.

Proposition 4 *If* $P = \sum_I a_i; P_i$ *and* $Q = \sum_J b_j; Q_j$ *then* $l :: (P \mid Q) \sim_D l :: (\sum_I a_i; (P_i \mid Q)) + l :: (\sum_J b_j; (Q_j \mid P)) + l :: (\sum_{a_i = \overline{b_j}} \tau; (P_i \mid Q_j))$

Example 1 *Note that the expansion theorem in CCS is not ualid directly. In general,* $(a \mid b)$ *is not bisimilar to* $a; b + b; a$. *If there are two locations (l and m),* $(a \mid b)$ *can exhibit* $< m, a >, < l, b >$, *which* $a; b + b; a$ *cannot exhibit. But* $l :: (a \mid b)$ *is bisimilar to* $l :: (a; b + b; a)$.

However, $a; (b; c + c; b)$ *is bisimilar to* $a; (b \mid c)$. *This is because the execution of action a fixes the location for the remainder of the process. If a parallel execution of b and c is to be permitted the process can be coded as follows. Let* l_1 *and* l_2 *be two distinct locations.* $(a; \overset{<l_1,t>}{\rightsquigarrow}; \overset{<l_2,t>}{\rightsquigarrow} \mid (l_1 :: \bar{t}; b) \mid (l_2 :: \bar{t}; c) \backslash \{t\}$. *The termination of the action a is explicitly sent to the two locations. Processes at the two locations wait for this information before proceeding in parallel.*

Proposition 5 *If* $\mathcal{L} = \{l1, 12, \ldots ln\}$,

$a; P \sim_D \sum_{li \in \mathcal{L}} li :: (a; P)$

$P \mid Q \sim_D \sum_{li \in \mathcal{L}} (li :: P) \mid Q + \sum_{li \in \mathcal{L}} P \mid (li :: Q)$

Bisimilarity involving the hiding operator is more complex than in CCS. In CCS (b;P)\{a} is b isimilar to Nil if b is equal to a, and to b;(P\{a}) otherwise. While this holds in our case the relation involving send actions is not straight forward. For example, $l :: (\overset{<m,c>}{\rightsquigarrow}; b \mid m :: a)\backslash\{a\}$ is bisimilar to $l :: \overset{<m,c>}{\rightsquigarrow}; b$. This is because the a action on m cannot be executed and there is no action which sends $\bar{a}$ to location m. However, $(l :: \overset{<m,a>}{\rightsquigarrow})\backslash\{a\}$ is not bisimilar to $l :: \overset{<m,a>}{\rightsquigarrow}$ as the latter can exhibit a at location m; nor is it bisimilar to Nil as the former can exhibit a send action. Similarly $l :: (l :: (\overset{<m,\bar{a}>}{\rightsquigarrow}; b \mid m :: a)\backslash\{a\}$ is in 'normal form' as there is a causal link between the send action and the b action on l and the a action on m. These observations play an important role when considering the axiomatization of the bisimulation equivalence which will be reported in a forthcoming paper.

In the above definition of bisimulation equivalence, we assumed that the observer could observe the τ action. The equivalence constructed is referred to as strong bisimulation. Another equivalence which is well studied is weak equivalence. Towards that, a transition relation $\Longrightarrow$ as P $\overset{a}{\Longrightarrow}$ Q iff P($\overset{\delta}{\longrightarrow}$)* $\overset{a}{\longrightarrow}$ ($\overset{\delta}{\longrightarrow}$)* is defined. The generalization to the above definition to the distributed case is not straight forward. This is because we define a stepped semantics and evolutions from multiple locations is possible. For example, transitions such as P $\overset{<l,a>,<m,\tau>}{\longrightarrow_D}$ P$'$ have to be mapped to one without τ. Omitting the $< m, \tau >$ is not advisable. If P were placed in a parallel context (such as P | Q) Q would be able to exhibit an action at location m in the same step as P exhibits $< l, a >$. By disallowing step semantics one can use the original definition, but the presence of step semantics requires a reformulation. An appropriate reformulation of observational equivalence in the presence of step semantics is being studied.

2.2 Testing

In this section we look at the applicability of the ideas developed in [DH84] for testing distributed processes. CCS processes are tested by CCS processes equipped with a special action ω which indicates success. Two basic types of testing are defined as follows:

Definition: 7

A process p **may satisfy** *a test o if* $(o \mid p) \xrightarrow{\tau*} q$ *and* $q \xrightarrow{\omega}$.

A process p **must satisfy** *a test o if* $(o \mid p) \xrightarrow{\tau} (o_1 \mid p_1) \xrightarrow{\tau} \ldots (o_i \mid p_i)$ *then* $\exists\ n \geq 0$, $o_n \xrightarrow{\omega}$ *and if* $(o_k \mid p_k)$ *can diverge, then for some* $k' \leq k$, $o_{k'} \xrightarrow{\omega}$ *(also called the existence of a successful extension).*

A similar technique can be used for distributed processes. However, given the current definition of the operational semantics, permitting only τ's in the testing relation is not sufficient. In CCS all non-τ actions had a corresponding synchronization action. In our extension, elements of Sact cannot be synchronized to produce a τ action. Therefore, one cannot differentiate between l :: (a + $\overset{<m,b>}{\rightsquigarrow}$) and l :: (a). Therefore one has to permit send actions to be part of the testing process. If one considers τ as communication at a local site, then send actions are communications across sites and contribute to the testing process.

The semantics of the parallel combinator permits both 'asynchronous interleaving' and 'simultaneous evolution' of processes at different locations. The testing process must be permitted to test for transitions at different locations; otherwise it will not be able to distinguish l :: (a; $\overset{<m,b>}{\rightsquigarrow}$; c) and l :: (a; $\overset{<m,d>}{\rightsquigarrow}$; c). However the testing procedure may be restricted to observe only interleaving of actions across all locations.

Another issue is the observation of success. One may insist that a test is successful only if ω is observed at 'all locations'. However, it is not possible to know a priori all the locations involved. Thus an observation of ω at any location is considered to be a success.

Definition: 8

A process p **may satisfy** *a test o (written as p* **may** *o) if $(o \mid p) \xrightarrow{TO_1}_D q_1 \overset{TO_2}{\longrightarrow}_D q_2 \ldots q$ and $q \xrightarrow{\{<,\omega>\}}_D$ for some location l, where each TO_i is a partial function from $\mathcal{L} \rightarrow (\{\tau, \delta\} \cup Sact)$.*

A process p **must satisfy** *a test o (written as p* **must** *o) if euery evolution of $(o \mid p)$ is either successful (i.e., exhibits ω at some location) or has a successful extension.*

Definition: 9

$p \simeq_{may} q : \forall$ *tests o, p* **may** *o iff q* **may o**.

$p \simeq_{must} q : \forall$ *o, p* **must** *o iff q* **must o**.

$p \simeq q : p \simeq_{may} q$ *and* $p \simeq_{must} q$.

Example 2 *Given at least two distinct locations, l and m, $(a; b + b; a) \not\simeq (a \mid b)$. Consider the testing process $(l :: \bar{a}; \overset{<m,c>}{\rightsquigarrow} \mid m :: \bar{c}; b; \omega) \backslash \{c\}$. The first process cannot pass the test while the second may pass the test. Note that the synchronization between the two branches of the tester is essential. If one had only $m :: (\bar{b}; \omega)$ then both the process will pass the test. Thus we haue defined a 'non-interleaving' testing equivalence.*

Proposition 6 $\simeq$ *is substitutive with respect to* $|$

Proposition 7 $P \sim_D Q$ *implies* $P \simeq Q$.

Proposition 8 *If P and Q are CCS terms, $P \simeq_{\mathrm{CCS}} Q$ implies $l :: P = l :: Q$.*

Further study is necessary to characterize the differences between bisimulation and testing equivalences for the distributed processes.

3 Program Development

While a theory usually abstracts away issues related to implementation details, it should be possible to derive a implementation dependent semantics from the more abstract semantics. Distributed program development can be divided in two phases: 1) Binding and 2) Configuration. In the binding

phase the system is specified with an expected architecture. For example, the sorting algorithm used for hypercubes will be different than the algorithm used for trees. Thus the specification of the algorithm indicates the architecture. One can consider that CCS with location addresses the issue of binding. A specification may assume the availability of a certain number of processing elements. But the physical architecture may be smaller. The issue then is to map the expected set onto the physical set which is called configuration. In the following two subsections we show how a semantics related to mapping the logical architecture to the physical one can be derived from the distributed semantics.

3.1 Configuration

Configuration is the mapping of a logically distributed program to a physical network. Usually, the number of processing elements in the physical network is less than the number of logically distributed units. One can represent the physical network by a set of locations and define a function Config: $\mathcal{L} \rightarrow \mathcal{L}'$, where $\mathcal{L}'$ represents the physical sites. Define a translation of a process using $\mathcal{L}$ into a process using $\mathcal{L}'$ as follows. $\mathcal{L}$ represents the logical parallelism used in defining processes while $\mathcal{L}'$ is the physical parallelism permitted.

Trans(a;P) = a;Trans(P)

Trans($\overset{<l,a>}{\rightsquigarrow}$;P) = $\overset{<Config(l),a>}{\rightsquigarrow}$; P

Trans(P | Q) = Trans(P) | Trans(Q)

Trans(P + Q) = Trans(P) + Trans(Q)

Trans(l :: P) = Config(l) :: Anchor(P, l)

Anchor(a; P, m) = a; Anchor(P, m)

Anchor($\overset{<l,a>}{\rightsquigarrow}$; P, m) = $\overset{<Config(l),a>}{\rightsquigarrow}$; Anchor(P,m)

Anchor((P | Q), m) = Anchor(P, m) | Anchor(Q, m)

Anchor((P + Q), m) = Anchor(P, m) + Anchor(Q, m)

Anchor(l :: P, m) = Nil if $1 \neq m$

Anchor(l :: P, m) = l :: Anchor(P, m) if $1 = m$

Trans converts all the old locations to the appropriate new locations. Anchor is needed to ensure that processes which could not exhibit any action continue to remain so under configuration. If Trans(l :: P) were defined to be Config(l) :: Trans(P) this property is not retained as shown by the following example.

Example 3 *Consider* $l :: (m :: a)$ *with* $l \neq m$. *It is bisimilar to Nil under* $\longrightarrow_D$. *However, if* $Config(l) = Config(m) = m'$ $l :: (m :: a)$ *will be* $m' :: (m' :: a)$ *which can exhibit action a. The intuition is that a 'wrongly' constructed program cannot be rectified at the binding stage.*

Proposition 9 $(P \sim_D Q)$ *implies* $(Trans(P) \sim_D Trans(Q))$

The above result can be generalized to relate configurations.

Definition: 10 *Let Conf1, Conf2:* $\mathcal{L} \rightarrow \mathcal{L}'$.

Define Conf1 $\leq$ *Conf2 iff Conf2*(l) $=$ *Conf2(m) implies Conf1*(l) $=$ *Conf1(m)*

Thus, if Conf1 $\leq$ Conf2, processes under Conf2 can exhibit 'more parallelism', which gives the following proposition.

Proposition 10 *Conf2(P)* $\sim_D$ *Conf2(Q) implies Conf1(P)* $\sim_D$ *Conf1(Q)*

3.2 Routing

In the distributed semantics we tacitly assumed a fully connected topology. This is acceptable for logical purposes. However, when mapping onto a physical network, a transfer from one location to another could make a number of hops. This information will depend on the routing tables used.

Definition: 11 *Define a routing table as a function:* $\mathcal{L} \times \mathcal{L} \rightarrow \mathcal{L}$. *Route(l, m)* $= n$ *(and* $m \neq n$*) is to be read as the route from l to m uses the connection from l to n.*

A new location semantics using the routing information can be defined as follows.

$$\frac{\mathrm{P} \stackrel{\{<l,\overset{<n,a>}{\rightsquigarrow}>\}}{\longrightarrow_D} \mathrm{P}', (\mathrm{m} \neq \mathrm{n}), \mathrm{Route(l, n)} = \mathrm{m}}{\mathrm{P} \stackrel{\{l,\overset{<n,a>}{\rightsquigarrow}\}}{\longrightarrow_R} \mathrm{P}' \mid \mathrm{m} :: \overset{<n,a>}{\rightsquigarrow}; \mathrm{nil}}$$

$$\frac{\mathrm{P} \stackrel{\{<l,\overset{<n,a>}{\rightsquigarrow}>\}}{\longrightarrow_D} \mathrm{P}', \mathrm{Route(l, n)} = \mathrm{n}}{\mathrm{P} \stackrel{\{l,\overset{<n,a>}{\rightsquigarrow}\}}{\longrightarrow_R} \mathrm{P}' \mid \mathrm{n} :: a; \mathit{Nil}}$$

Proposition 11 *As expected* $(P \sim_R Q)$ *iff* $(P \sim_D Q)$

4 Examples

In this section we present a few examples of distributed processing.

Example 4 *The following is an encoding of RPC* [BN81]. *A caller process sends a request to the callee and waits for a response. The callee waits for a request, calls the procedure and sends an acknowledgement. The calling of the procedure is denoted by a and the response by b. The actual procedure (call-procedure) is modeled as an action.*

$$\mathit{caller} = \overset{<m,a>}{\rightsquigarrow}; \bar{b}; \mathit{caller}$$

$$\mathit{callee} = \bar{a}; \mathit{call\text{-}procedure}; \overset{<l,b>}{\rightsquigarrow}; \mathit{callee}$$

$$\mathit{System} = (\ (l :: \mathit{caller}) \mid (m :: \mathit{callee}))\ \backslash\{a, b\}$$

Though the above code assumes a fixed location to send the response the location of the caller can be coded to be part of the action.

$$\mathit{caller}_i = \overset{<m,a_i>}{\rightsquigarrow}; \bar{b}; \mathit{caller}$$

$$\mathit{callee} = \textstyle\sum_i \bar{a_i}; \mathit{call\text{-}procedure}; \overset{<l_i,b>}{\rightsquigarrow}; \mathit{callee}$$

$$\mathit{System} = (\ (\ \textstyle\prod_i l_i :: \mathit{caller}) \mid (m :: \mathit{callee}))\ \backslash\{a, b\}$$

Example 5 *The following is an encoding of call streams [LS88]. Here the caller does not wait for an the acknowledgement. If continues its local processing (local-processing) and is willing to accept an acknowledgement when it has arrived.*

$$caller = \overset{<m,a>}{\rightsquigarrow};\ (\bar{b};\ caller + local\text{-}processing:\ caller)$$

$$callee = \bar{a};\ call\text{-}procedure;\ \overset{<l,b>}{\rightsquigarrow};\ callee$$

$$System = (\ (l :: caller) \mid (m :: callee))\ \backslash\{a, b\}$$

Example 6 *Encoding of tuple space as in Linda [Gel85]. In this example, we assume that there are two process (out1 and out2) which output to the tuple space; one process which removes tuples from the tuple space (in) and a process which reads the tuple space (read). The tuple space receives an in request and then returns a tuple in its space. The removal of the tuple is 'tacit' due to the semantics of synchronization. The read request is handled similarly except that the tuple is 'regenerated' as the synchronization removed it. The tuple space (tup) can either accept either an in action, or a read action. In our example, ci is the location of the in process, cr the location of the read process and tup the location of the tuple space.*

$$out1 = \overset{<t,a>}{\rightsquigarrow};\ out1$$

$$out2 = \overset{<t,b>}{\rightsquigarrow};\ out2$$

$$in = \overset{<t,i>}{\rightsquigarrow};(\bar{a};\ in + \bar{b};\ in)$$

$$read = \overset{<t,r>}{\rightsquigarrow};(\bar{a};\ read + \bar{b};\ read$$

$$tup = \bar{a};\ \overset{<ci,a>}{\rightsquigarrow};\ tup + \bar{b};\ \overset{<cr,b>}{\rightsquigarrow};\ tup) + \bar{r};(\bar{a};((\overset{<cr,a>}{\rightsquigarrow};\ tup) \mid a;\ nil) + \bar{b};(\overset{<cr,b>}{\rightsquigarrow};tup) \mid b;\ nil)\ )$$

$$System = (out1 \mid out2 \mid c1 :: in \mid cr :: read \mid t :: tup)\ \backslash\{a, b\}$$

Example 7 *Consider a printer(Pr) which interacts with a generator(Gen) and a console (Cs). The generator issues the print command. The console*

is informed about the status by the printer. If the printer is fine (ok) it waits a request (req) and prints it (print). If it is out of paper, it informs the console (op) and waits for new paper to be added. The first set of definitions present the specification in CCS, while the second uses asynchronous message passing.

$Gen = pc;\ req;\ Gen$

$Pr = (ok;\ \overline{req};\ print;\ Pr) + (op;\ \overline{np};\ Pr)$

$Cs = (\overline{op};\ np;\ Cs) + (\overline{ok};\ Cs)$

$Sys = (Gen \mid Pr \mid Cs) \backslash \{req,\ op,\ np,\ ok\}$

The distributed process is as follows.

$Gen = pc;\ \overset{<prl,rp>}{\rightsquigarrow};\ Gen$

$Pr = (\overset{<csl,ok>}{\rightsquigarrow};\overline{req};\ print;\ Pr) + (\overset{<csl,op>}{\rightsquigarrow};\ \overline{np};\ Pr)$

$Cs = (\overline{op},\ \overset{<prl,np>}{\rightsquigarrow};\ Cs) + (\overline{ok}Cs)$

$Sys = ((gl :: Gen) \mid (prl :: Pr) \mid (csl :: Cs))\backslash\{req,\ op\ ,np,\ ok\}$

In the CCS version, the generator can proceed only when the printer is ready. However, in the asynchronous case, there is no limit on the number of items to be printed before printing occurs. The creating of the 'message' processes can be thought of as the spool area. Also notice that ordering of ok messages between the printer and the console need not be preserved. However, the printer cannot continue if it is out paper until the console has fixed the problem. The above observations are valid even if the configuration assigns the three processes to the same processing element. The difference is in the send actions and the number of actions observed in a step.

5 Related Work

CCS with location was introduced in [KHCB91]. The main operational rules in their semantics are as follows: $a; p \xrightarrow[u]{a} p \;\; u \in Loc^*$ and $\dfrac{p \xrightarrow[l]{a} p'}{u :: p \xrightarrow[vu]{a} v :: p'}$.

The main difference of our semantics, is that we do not allow the evolution of processes such as v :: u :: P with v $\neq$ u. The evolution is permitted by [KHCB91] as they are concerned with the structure of the process rather than implementing a process on a distributed architecture. A location string 'vu' indicates that it is a sub-location of 'v'. They also assume an infinite number of locations and distinguish between $a; (b \mid c)$ and $a; (b; c + c; b)$. Similarly [CH89] present a distributed bisimulation semantics. The motivation and the results is similar to [KHCB91]. A detailed comparison between [KHCB91] and [CH89] is presented in [KHCB91]. In short, both the approaches consider spatial issues in distributed computation at the logical level. They do not consider limitations imposed by a physical architecture. Our semantics is also similar to step semantics [vGV87] but we have bounded the number of processes that can evolve in one step by the number of locations present. In 'distributed semantics' such as [DDM88] the primary concern is causality and not physically distributed computing elements. In the Chemical Abstract Machine [BB89], synchronization in a distributed environment is achieved by 'moving' the processes 'next to each other'. This while resulting in a nice theory is not accurate for all systems. In actual systems, processes are usually static and communicate with messages. This is not surprising as the pro-gramming paradigm on which the Chemical abstraction Machine is based [BCM88] assumes that all data is available in shared storage.

6 Conclusion

This is very much a working paper. We have used of idea of locations to denote distribution and presented a semantics based on the restrictions imposed by distributed hardware. Much more work is essential to understand the effect of a distributed architecture on process behavior. We have outlined a few basic results for strong bisimulation and testing equivalences. Issues

such as weak equivalence, a complete axiomatization for these equivalences etc. need to be investigated. We have also outlined how configuration and routing can be handled. A few examples demonstrating the use of our location CCS has been presented. The applicability of this to more specific architectures such as cube, meshes etc. are currently under study.

Acknowledgement

The author is thankful to Uffe Engberg for many suggestions one of which was to use locations to characterize distribution. Thanks also to Peter Mosses and Jens Palsberg for taking a keen interest in this work.

References

[BB89] G. Berry and G. Boudol. The Chemical Abstract Machine. Technical Report 1133, INRIA-Sophia Antipolis, December 1989.

[BCM88] J. P. Banatre, A. Coutant, and D. Metayer. A Parallel Machine for Multiset Transformation and its Programming Style. *Future Generation Computer Systems*, 4:133–144, 1988.

[BN81] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(4):39–59, February 1981.

[CG83] K. L. Clark and S Gregory. PARLOG: A Parallel Logic Programming Language. Technical Report 5, Imperial College, May 1983.

[CH89] I. Castellani and M. Hennessy. Distributed Bisimulations. *Journal of the Association for Computing Machinery*, 36(4):887–911, October 1989.

[CM87] K. M. Chandy and J. Misra. Parallelsim and Programming: A Perspective. In *Foundations of Software Technology and Theoretical Computer Science, LNCS 287*, pages 173–194. Springer Verlag, 1987.

[DDM88] P. Degano, R. DeNicola, and U. Montanari. A Distributed Operational Semantics for CCS Based on Condition/Event Systems. *Acta Informatica*, 26:59–91, 1988.

[DH84] R. DeNicola and M. C. B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Language and Systems*, 7(1):80–112, Jan 1985.

[Hud86] P. Hudak. Parafunctional Programming. *IEEE Computer*, 19(8):60–71, 1986.

[INM88] INMOS Ltd. **occam-2** *Reference Manual*. Prentice Hall, 1988.

[KHCB91] A. Kiehn, M. Hennessy, I. Castellani, and G. Boudol. Observing Localities. In *Workshop on Concurrency and Compositionality: Goslar*, 1991.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 260–267, 1988.

[Mar89] S. T. March, editor. *ACM Computing Surveys*, volume 21,3. ACM, 1989.

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

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI Conference, LNCS-104.* Springer Verlag, 1981.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[Rus78] R. Russell. The CRAY-1 Computer System. *CACM*, January 1978.

[TBH82] P. Treleaven, D. Brownbridge, and R. Hopkins. Data Driven and Demand Driven Computer Architectures. *ACM Computing Surveys*, 14(1), 1982.

[vGV87] R. J. van Glabbeek and F. W. Vaandrager. Petri Net Models for Algebraic Theories of Concurrency. In J. W. deBakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE-II , LNCS 259.* Springer Verlag, 1987.