Interaction Semantics
for
Components of Distributed Systems

Carolyn Talcott
Stanford University
clt@sail.stanford.edu

Keywords: actors, components, distributed systems, semantics, interaction, composition

Abstract: The actor model is a natural starting point for a semantic theory that treats both heterogeneity and modularity (encapsulation and composability) in open distributed systems. This paper begins with some simple examples of actor systems that illustrate the essential features of actor based computation, and the interactions and combinations of components these systems. We then present a semantic theory that models these features. A notion of abstract actor structure (AAS) is introduced that characterizes the minimal semantic requirements for an actor language and allows for components to be defined using multiple languages. We model components as collections of actors accessed via a specified subset called receptionists. A transition system semantics for components is derived from the local rules for behavior of individual actors given in an AAS. This semantics accounts for both internal computation and interaction of a component with its environment. We abstract away from details of internal computation to define a notion of interaction semantics. This allows us to reason about equivalence of actor system components considered as black boxes. An algebra of interaction sets corresponding to the algebra of components is defined, verifying that the interaction semantics compositional. This provides a basis for modular reasoning about construction and transformation of components specified as abstract actor systems.

1. Introduction

We are interested in developing tools for reasoning about components of open distributed systems. This includes behavior specification; equivalence of components; interactions of components with their environment; and composition of components and their specifications. The modern computing environment is becoming increasingly open and distributed. The main characteristics of an open distributed system are that such systems allow the addition of new components, the replacement of existing components, and changes in interconnections between components, largely without disturbing the functioning of the system. Open distributed systems require a discipline in which a component may not have any direct control over other components with which it is connected. Instead, the behavior of a component is locally determined by its initial state and the history of its interactions with the environment. Moreover, interactions between components may occur only through their interfaces. Thus, the internal state of a component must only be accessible through operations provided by the interface.

The actor model of computation [6, 1, 2] has a built-in notion of local component and interface, and thus it is a natural model to use as a basis for a theory of open distributed computation. Actors are independent computational agents that interact solely via message passing. An actor can create other actors; send messages; and modify its own local state. An actor can only send messages to its acquaintances – actors whose addresses it was given upon creation, or received in a message, or actors it created. Actor semantics requires computations to be fair. We take two views of actors: as individuals and as elements of components. Individual actors provide units of encapsulation and integrity. Collecting actors into components provides for composability and coordination. Individual actors are described in terms of local transitions. Components are described in terms of interactions with their environment.

In [3] a higher-order actor language was defined, notions of program equivalence were studied, and methods were developed for establishing program equivalence. In this paper we abstract away from a specific choice of programming language and focus on the interactions of components. For this purpose we introduce abstract actor structures (AASs). An AAS provides an abstract set of actor state descriptions and functions that determine the local transitions of individual actors. This specializes and refines the two-level actor systems of [9]. Using techniques of concurrent rewriting semantics [7] the semantics of components is derived from the local semantics of individual actors. We note that the language and semantics of [3] can be easily reformulated in the AAS/rewriting framework.

The remainder of this paper is organized as follows. §2 presents a series of examples to illustrate the main concepts. In §3 abstract actor structures are presented. In §4 an algebra of actor components is defined, a computational semantics is given, and the interaction semantics of actor components is obtained by forgetting details of internal computations. In §5 an algebra of component behaviors is defined and correspondence to the component algebra is shown. This provides a method for computing the interaction semantics of compositions from that of the parts. §6 discusses related and future work.

2. Examples

In this section we give a series of examples that illustrate the main features of actor computation and the two complementary views: individual actors and components. A component is a collection of actors together with a designated subset, called the receptionists, that form the interface to the outside world. We indicate how local behavior of an individual actor is specified, and how the behavior of a component is derived from the local behaviors of the individual actors of the component. We also illustrate the construction of components from smaller parts.

2.1. Timer

The first example we consider is a timer. We write $(\,T(n))_c$ to indicate a timer actor with address c. The state of this actor, $T(n)$, indicates that the current elapsed time is n. The timer can tick (time passes) evolving to an actor with state $T(n+1)$. This is expressed by the local transition:

$$(\,T(n))_c \quad \xrightarrow{\texttt{tick}(c)} \quad (\,T(n+1))_c$$

The timer can also answer requests for the current elapsed time. This is expressed by a local transition involving the timer and a request message, $c \lhd \mathtt{time}(x)$.

$$\begin{matrix} (\,T(n))_c \\ c \lhd \mathtt{time}(x) \end{matrix} \quad \xrightarrow{\mathtt{a}(c,\mathtt{time}(x))} \quad \begin{matrix} (\,T(n))_c \\ x \lhd n \end{matrix}$$

A component consisting of a single timer $(\,T(n))_c$ with receptionist c can receive messages from actors in its external environment

$$(\,T(n))_c \quad \xrightarrow{\mathtt{in}(c \lhd v)} \quad \begin{matrix} (\,T(n))_c \\ c \lhd v \end{matrix}$$

and send messages to actors in its external environment.

$$\begin{matrix} (\,T(n))_c \\ x \lhd v \end{matrix} \quad \xrightarrow{\mathtt{out}(x \lhd v)} \quad (\,T(n))_c$$

Input and output interactions can occur asynchronously with internal activity,

$$(\,T(n))_c \quad \begin{matrix} \xrightarrow{\mathtt{in}(c \lhd \mathtt{time}(x))} \\ \xrightarrow{\mathtt{tick}(c)} \end{matrix} \quad \begin{matrix} c \lhd \mathtt{time}(x) \\ (\,T(n+1))_c \end{matrix} \quad \xrightarrow{\mathtt{in}(c \lhd \mathtt{time}(y))} \quad \begin{matrix} c \lhd \mathtt{time}(y) \\ (\,T(n+1))_c \\ c \lhd \mathtt{time}(x) \end{matrix}$$

and messages are not necessarily delivered to the target actor in the same order that they arrive at the component.

$$\begin{matrix} c \lhd \mathtt{time}(x) \\ (\,T(n+1))_c \\ c \lhd \mathtt{time}(y) \end{matrix} \quad \xrightarrow{\mathtt{a}(c,\mathtt{time}(y))} \quad \begin{matrix} (\,T(n+1))_c \\ c \lhd \mathtt{time}(x) \\ y \lhd n+1 \end{matrix} \quad \begin{matrix} \xrightarrow{\mathtt{out}(y \lhd n+1)} \\ \xrightarrow{\mathtt{a}(c,\mathtt{time}(x))} \end{matrix} \quad \begin{matrix} x \lhd n+1 \\ (\,T(n+1))_c \end{matrix}$$

The piles of actors and messages making up the configuration of a component are really multisets, also written linearly as lists with separating commas. Being multisets the order of elements of the list does not matter.

The *interaction semantics* of a component is the set of sequences of input/output interactions in which it might participate. This is completely determined by the local behaviors of the constituent actors (and the collection of messages initially present). The interaction semantics of a timer, $\mathcal{I}((\,T(n))_c)$, is the set of sequences of elements $\mathtt{in}(c \lhd v)$, $\mathtt{out}(x \lhd n')$ satisfying the following three conditions: (1) for each request, $\mathtt{in}(c \lhd \mathtt{time}(x))$ $(x \neq c)$, there is a corresponding reply, $\mathtt{out}(x \lhd n')$, for some $n' \geq n$; (2) each output is a reply to some earlier request; and (3) inputs not of the form $\mathtt{in}(c \lhd \mathtt{time}(x))$ are ignored.

2.2. Staged Calculation of a Function

Let ϕ be a function with domain W. We let $(\texttt{fun})_a$ denote a ϕ-computer – an actor that accepts requests to compute $\phi(w)$ for $w \in W$, and carries out the computation in stages. $\texttt{funx}(x, \texttt{init}(w))$ is the initial stage of a ϕ-computer responding to a request $\texttt{req}(x, w)$. At stage $\texttt{funx}(x, s)$ where s is not final, a ϕ-computer can do some increment of work, progressing to stage $\texttt{funx}(x, \texttt{next}(s))$. If s is final, the result, $\texttt{answer}(s)$, can be sent to x. The local behavior of a ϕ-computer is given by the following transitions.

$$(\texttt{fun})_a\,,\ a \triangleleft \texttt{req}(x,w) \quad \xrightarrow{\texttt{a}(a,\texttt{req}(x,w))} \quad (\texttt{funx}(x,\texttt{init}(v)))_a \qquad \text{if} \quad w \in W$$

$$(\texttt{funx}(x,s))_a \quad \xrightarrow{\texttt{ex}(a)} \quad \begin{cases} (\texttt{funx}(x,\texttt{next}(s)))_a & s \text{ not final} \\ (\texttt{fun})_a\,,\ x \triangleleft u & s \text{ final with } \texttt{answer}(s) = u \end{cases}$$

We assume that the computation always terminates for $w \in W$. Thus the interaction semantics of a ϕ-computer component, $\mathcal{I}((\texttt{fun})_a)$, consists of sequences of inputs to a and outputs to requestors such that there is exactly one output, $x \triangleleft \phi(w)$, corresponding to each valid input, $a \triangleleft \texttt{req}(x, w)$ with $w \in W$. Ill-formed inputs are ignored.

2.3. Dispatching

We want to build a component, $\texttt{Dispatch}(\vec{a})[d]$, with receptionist d and external acquaintances $\vec{a} = [a_1, \ldots, a_k]$ that dispatches each incoming message to d to some 'randomly' selected element of $\vec{a}$. Individual actors are deterministic, so how can we do this? One way is to define a pre-dispatcher, $(\texttt{dis}(\vec{a}, z))_d$, that asks a timer, z, for elapsed time and uses the reply to determine which actor to select for each incoming message. To separate timer replies from incoming requests, a helper actor is created for each incoming message. The helper's job is to receive the timer reply and to do the actual message dispatch. The pre-dispatcher $\texttt{dis}$ and the helper $\texttt{sel}$ are specified as follows.

$$(\texttt{dis}(\vec{a},z))_d\,,\ d \triangleleft v \quad \xrightarrow{\texttt{a}(d,v)} \quad ((\texttt{dis}(\vec{a},z))_d\,,\ (\texttt{sel}(\vec{a},v))_b\,,\ z \triangleleft \texttt{time}(b)) \lceil d$$

$$(\texttt{sel}(\vec{a},v))_b\,,\ b \triangleleft n \quad \xrightarrow{\texttt{a}(b,n)} \quad (?)_b\,,\ a_i \triangleleft v \qquad \text{where } i = n \bmod k$$

In the first rule b is a new actor address (from the point of view of the component), and the restriction, $\_ \lceil d$, indicates that only d can receive messages from the external environment. In particular, b is not visible externally until its address is explicitly exported. The ? state of b after doing the dispatch is a sink state that can not create any new actors or send any messages. Now the desired dispatcher can be constructed by composing a pre-dispatcher $(\texttt{dis}(\vec{a}, z))_d$ with a timer $(T(n))_c$. In order to make the connection we first rename z to c. To hide the timer we restrict the visible actors to d.

$$\texttt{Dispatch}(\vec{a})[d] \;\stackrel{\triangle}{=}\; ((\texttt{dis}(\vec{a},z))_d^{\{z \mapsto c\}}\,,\ (T(0))_c) \lceil d = ((\texttt{dis}(\vec{a},c))_d\,,\ (T(0))_c) \lceil d$$

The interaction semantics of $\texttt{Dispatch}(\vec{a})[d]$ consists of sequences of inputs to d and outputs to an element of $\vec{a}$ such that there is exactly one output for each input and that the corresponding inputs and outputs have the same message contents. We will see later how to compute this from the interaction semantics of the two subcomponents.

Two components are said to be *interaction equivalent*, written $C_0 \overset{i}{\cong} C_1$, if they have the same interaction semantics. We justify omitting the timer parameter from the parameters of the dispatcher component by the following equivalence.

(D1) $(\,(\mathtt{dis}(\vec{a},c))_{d}\,,\,(\,T(0))_{c})\lceil d \;\overset{i}{\cong}\; (\,(\mathtt{dis}(\vec{a},c))_{d}\,,\,(\,T(n))_{c})\lceil d \quad \text{for} \quad n \in \mathbf{Nat}$

One use of a dispatcher might be for a form of load balancing. This is accomplished by composing the dispatcher with a collection of actors having the same behavior. For example, let $\vec{a} = [a_1, \ldots, a_k]$ be a list of distinct actor addresses, and let $\mathtt{Fun}[\vec{a}] \overset{\triangle}{=} (\mathtt{fun})_{a_1}\,, \ldots, (\mathtt{fun})_{a_k}$ be a component consisting of k copies of $(\mathtt{fun})_{a}$. Behavioral correctness of this load balancing transformation is expressed by the following interaction equivalence:

(D2) $(\mathtt{Dispatch}(\vec{a},c)[d]\,,\,\mathtt{Fun}[\vec{a}])\lceil d \;\overset{i}{\cong}\; (\mathtt{fun})_{d}$

Note that without the assumption of termination for the computation of $\phi(w)$ the equivalence would not hold.

3. Abstract Actor Structures

In this section we define the notion of abstract actor structure. To make the ideas more concrete we interleave definitions with discussion showing how various aspects of the examples of §2 are expressed in the AAS framework.

An abstract actor structure (AAS) is a structure of the form

$$\langle \mathbf{A}, \mathbf{V}, \mathbf{S}; \mathit{En}_{\mathrm{d}}, \mathit{Deliv}, \#\mathit{new}, \mathit{Ex}, \mathit{acq}, \hat{\_} \rangle$$

where $\mathbf{A}$ is a countable set of actor addresses, $\mathbf{V}$ is a set of values that can be communicated between actors, and $\mathbf{S}$ is a set of actor states. The set of values includes the set of actor addresses. Actor states are intended to carry information traditionally contained in the script (methods) and acquaintances (instance variables), as well as the local message queue and the current processing state. We let a range over $\mathbf{A}$, v range over $\mathbf{V}$, s range over $\mathbf{S}$, using the convention that "x range over X" means that the meta-variable x and decorated variants such as x', $x_0, \ldots$ range over the set X. $(\,s\,)_{a}$ is an actor with address, a, in state, s. $a \triangleleft v$ is a message with target, a, and contents, v.

$\mathit{En}_{\mathrm{d}}(s)$ is a predicate on actor states that holds if the state, s, is enabled for message delivery. If s is enabled for delivery, then $\mathit{Deliv}(s,v)$ is an actor state in which, intuitively, a message with contents v has been added to the local message queue. Since new actors may be created and the addresses of these new actors can only be determined at 'runtime' we formulate local semantics, $\mathit{Ex}((\,s\,)_{a})$, as a function from lists of distinct actor addresses to simple actor system fragments – multisets of actors and messages in which no two actor occurrences have the same address. $\#new(s)$ is the number of new actors that will be created by actor a in state s executing a step and $Ex((s)_a)([a_1, \ldots a_{\#new(s)}])$ is the fragment produced. This fragment contains the actor with address a, possibly with a modified state, and an actor with address a_i for $1 \leq i \leq \#new(s)$. It may also contain messages to new actors or to actors previously known by a in state s. For an idle actor, $(s)_a$, we have $\#new(s) = 0$ and $Ex((s)_a)([\,]) = (s)_a$.

Timer states and steps We assume that the set of values contains numbers and requests $\mathtt{time}(x)$ for $x \in \mathbf{A}$. Our presentation of the local semantics of a timer combined the delivery and processing of messages. The AAS framework separates these aspects of actor computation. To account for this we introduce auxiliary states $T^\dagger(n,x)$ to represent the state of a timer that has received a request, $\mathtt{time}(x)$, but has not yet processed it. $T(n)$ is enabled for message delivery, while $T^\dagger(n,x)$ is not. Thus, for $n \in \mathbf{Nat}$, $x \in \mathbf{A}$

$$En_{\mathrm{d}}(T(n)) \qquad \neg(En_{\mathrm{d}}(T^\dagger(n,x))) \qquad Deliv(T(n),v) = \begin{cases} T^\dagger(n,x) & \text{if } v = \mathtt{time}(x) \\ T(n) & \text{otherwise} \end{cases}$$

$$\#new(T(n)) = \#new(T^\dagger(n,x)) = 0$$

$$Ex((T(n))_c)([\,]) = (T(n+1))_c \qquad Ex((T^\dagger(n,x))_c)([\,]) = (T(n))_c\,,\ x \triangleleft n$$

Both $(T(n+1))_c$ and $(T(n))_c\,,\ x \triangleleft n$ are simple fragments with actor domain $\{c\}$. The latter is the union of two 'atomic' fragments, one an actor, the other a message.

Since states and values are abstract entities, a means of determining the actor addresses occurring in a state or value is needed. This is met by the acquaintance function, acq, which gives the (finite) set of actor addresses occurring in a state or value. Returning to the timer example we have $acq(\mathtt{time}(x)) = acq(T^\dagger(x,n)) = \{x\}$ for $x \in \mathbf{A}$ and $n \in \mathbf{Nat}$.

Actor addresses can not be explicitly created by actors, and the semantics can not depend on the particular choice of addresses of a group of actors. A renaming mechanism is used to formulate this requirement. We let ρ range over bijections on $\mathbf{A}$ (renamings). For any such ρ, $\widehat{\rho}$ is a renaming function on states and values that agrees with ρ on actor addresses. Renaming is extended naturally to structures built from addresses, states, and values. For example, if $\rho(c) = a$ and $\rho(x) = y$ then $\widehat{\rho}(\mathtt{time}(x)) = \mathtt{time}(y)$, $\widehat{\rho}(c \triangleleft \mathtt{time}(x)) = a \triangleleft \mathtt{time}(y)$, and $\widehat{\rho}((T^\dagger(n,x))_c) = (T^\dagger(n,y))_a$. We also require the renaming mechanism to commute with function composition: $\widehat{\rho_0 \circ \rho_1} = \widehat{\rho_0} \circ \widehat{\rho_1}$.

An AAS must obey the fundamental acquaintance laws of actors [4, 5]. In particular, the axioms (**AR**) and (**Ex**) below must hold in an AAS. It is easy to check that the timer specified above satisfies these axioms .

Acquaintance and Renaming Axioms (AR)

(i) $\quad acq(\widehat{\rho}(x)) = \widehat{\rho}(acq(x)) \quad \text{for} \quad x \in \mathbf{S} \cup \mathbf{V}$

(ii) $\quad acq(Deliv(s,v)) \subseteq acq(v) \cup acq(s) \quad \text{if} \quad En_{\mathrm{d}}(s)$

(iii) $\quad \widehat{\rho}(Deliv(s,v)) = Deliv(\widehat{\rho}(s), \widehat{\rho}(v)) \quad \text{if} \quad En_{\mathrm{d}}(s)$

(iv) $En_{\mathrm{d}}(s) \Leftrightarrow En_{\mathrm{d}}(\widehat{\rho}(s))$

(v) $\#new(s) = \#new(\widehat{\rho}(s))$

(vi) $(\forall a \in acq(x))(\rho(a) = a) \Rightarrow \widehat{\rho}(x) = x$ for $x \in \mathbf{S} \cup \mathbf{V}$

(i) and (iii) say that renaming commutes with the delivery and acquaintance functions. (ii) says that acquaintances of an actor in state, s, after delivery of a message with contents, v, are among the acquaintances of s and of v. (iv) and (v) say that renaming does not change enabledness or the number of actors that will be created upon execution. (vi) says that if a renaming fixes the acquaintances of an object then applying it does not change the object. (vi) together with the composition property of the renaming mechanism imply that two renamings that agree on the acquaintances of an object have the same result when applied to the object, and that $\widehat{\rho}$ is a bijection on $\mathbf{S} \cup \mathbf{V}$.

Execution axioms (Ex) If $\vec{a}$ is a list of $\#new(s)$ distinct actor addresses disjoint from $a, acq(s)$, then $Ex((s)_a)(\vec{a}) = sF$ for some fragment, sF, such that

(i) $adom(sF) = \{a\} \cup \vec{a}$ the set of addresses of actors occurring in sF

(ii) $(s')_{a'} \in sF \Rightarrow acq(s') \subseteq acq(s) \cup adom(sF)$

(iii) $a' \triangleleft v' \in sF \Rightarrow \{a'\} \cup acq(v') \subseteq acq(s) \cup adom(sF)$

(iv) $Ex(\widehat{\rho}((s)_a))(\widehat{\rho}(\vec{a})) = \widehat{\rho}(sF)$

(ii) says that any acquaintance of an actor after executing a step or of a newly created actor either was an acquaintance of the actor before the step is taken, or is one of the newly created actors. (iii) says that the targets and contents of newly sent messages are similarly constrained. (iv) says that executing a step commutes with renaming – that is, the local semantics is uniformly parameterized by the set of locally occurring actor addresses.

In the remainder of this paper we let AAS be a fixed but unspecified abstract actor structure. For purposes of the examples, we assume that the set of values includes **Nat**, the natural numbers, and that **A** is disjoint from **Nat**.

4. Actor System Components

An abstract actor structure allows one to specify the behavior of individual actors. Now we show how individual actors can be combined to form components and how the local semantics of individual actors determines the semantics of components. Our treatment is guided by ideas from rewriting logic [7]. Here we merely present the resulting labelled transition system that serves as the operational semantics. Details of the full rewriting logic presentation will appear elsewhere. The actor semantics has two aspects: internal transitions and interaction steps. Internal transitions are combinations of execution and message delivery steps. Interaction steps model a components interaction with its environment by exchange of messages.

4.1. The Actor Component Algebra

Actor system components, $C \in \mathit{Cmpt}$, are multisets of actors, $(s)_a$, messages, $a \lhd v$, and restricted components, $C \lceil R$ where R is a finite set of actor addresses. (Note that simple fragments are a special case.) The receptionists, $\mathit{recep}(C)$, of a component are the actors defined in the component that are externally visible. The externals, $\mathit{extrn}(C)$, of a component are the addresses of actors occurring as acquaintances of actor states, message targets, or message contents, but not defined in the component. Thus $\mathit{recep}(C) \cap \mathit{extrn}(C) = \emptyset$.

Definition (*Cmpt*, *recep*, *extrn*):

(mt) $\diamond \in \mathit{Cmpt}$,

$\mathit{recep}(\diamond) = \mathit{extrn}(\diamond) = \emptyset$

(act) $(s)_a \in \mathit{Cmpt}$,

$\mathit{recep}((s)_a) = \{a\}$ and $\mathit{extrn}((s)_a) = \mathit{acq}(s) - \{a\}$

(msg) $a \lhd v \in \mathit{Cmpt}$,

$\mathit{recep}(a \lhd v) = \emptyset$ and $\mathit{extrn}(a \lhd v) = \mathit{acq}(v) \cup \{a\}$

(mun) $C_0\, ,\ C_1 \in \mathit{Cmpt}$ if $\mathit{recep}(C_0) \cap \mathit{recep}(C_1) = \emptyset$,

$\mathit{recep}(C_0\, ,\ C_1) = \mathit{recep}(C_0) \cup \mathit{recep}(C_1)$

$\mathit{extrn}(C_0\, ,\ C_1) = \mathit{extrn}(C_0) \cup \mathit{extrn}(C_1) - \mathit{recep}(C_0\, ,\ C_1$

(rstr) $C \lceil R \in \mathit{Cmpt}$ if $R \subseteq \mathit{recep}(C)$,

$\mathit{recep}(C \lceil R) = R$ and $\mathit{extrn}(C \lceil R) = \mathit{extrn}(C)$

Some example calculations of receptionists and externals are:

$$\mathit{recep}((\mathtt{dis}(\vec{a}, c))_d\, ,\ (T(n))_c) = \{d, c\} \qquad \mathit{extrn}((\mathtt{dis}(\vec{a}, c))_d\, ,\ (T(n))_c) = \vec{a}$$
$$\mathit{recep}(((\mathtt{dis}(\vec{a}, c))_d\, ,\ (T(n))_c) \lceil d) = \{d\} \qquad \mathit{extrn}(((\mathtt{dis}(\vec{a}, c))_d\, ,\ (T(n))_c) \lceil d) = \vec{a}$$

To simplify the transition rules we define a structural equivalence relation on components.

Definition (Structural equivalence): $C \sim C'$ is the least congruence on *Cmpt* satisfying the the multiset union axioms (_ , _ is associative, commutative, with identity $\diamond$) and the following (where in each case it is implicitly assumed that both sides of the equivalence is well-formed)

(top) $C \sim C \lceil \mathit{recep}(C)$

(erase) $(C_0 \lceil R_0\, ,\ C_1) \lceil R \sim (C_0\, ,\ C_1) \lceil R$ if $(\mathit{recep}(C_0) - R_0) \cap \mathit{extrn}(C_1) = \emptyset$

(alpha) $C \lceil R \sim \widehat{\rho}(C) \lceil R$ if ρ is the identity on R and $\mathit{extrn}(C)$

Examples of Structural Equivalence We can use the structural equivalence rules to justify the following equations for components taken from our examples.

(1) $(T(n))_c \sim (T(n))_c \lceil c$ by (top)

(2) $(((\mathtt{dis}(\vec{a}, c))_d\,,\ (T(n))_c) \lceil d,\ F(\vec{a})) \lceil d$

$\sim ((\mathtt{dis}(\vec{a}, c))_d\,,\ (T(n))_c\,,\ F(\vec{a})) \lceil d$ by (erase)

$\sim ((\mathtt{dis}(\vec{b}, c))_d\,,\ (T(n))_c\,,\ F(\vec{b})) \lceil d$ by (alpha)

Note that each component C has a structurally equivalent flat form, $(\alpha\,,\ \mu) \lceil R$ where α is the multiset of actors defined in C, and μ is the multiset of messages occurring in C.

4.2. Internal Computation

Definition (Internal transitions): The internal transition relation $\tau : C \Longrightarrow C'$ is defined by the following rules:

(exe) $(s)_a \Longrightarrow Ex((s)_a)(\vec{a}) \lceil a$

(del) $(s)_a, a \lhd v \Longrightarrow (Deliv(s, v))_a$ if $En_{\mathrm{d}}(s)$

(id) $C \Longrightarrow C$

(cmps) $C_0 \Longrightarrow C_2$ if $C_0 \Longrightarrow C_1$ and $C_1 \Longrightarrow C_2$

(mun) $C_0\,,\ C_1 \Longrightarrow C_0'\,,\ C_1'$ if $C_j \Longrightarrow C_j'$ for $j < 2$

(rcp) $C \lceil R \Longrightarrow C' \lceil R$ if $C \Longrightarrow C'$

The use of $\lceil a$ in the (exe) rule ensures that internal transitions do no change the set of receptionists. Combined with the (out) rule below, newly created actors become visible outside a component only when their addresses are explicitly sent out in a message.

4.3. Computational Semantics

Definition (Interaction steps): Interaction steps, $\gamma : C \Longrightarrow C'$ are of one of the following forms.

(silent) $\langle\,\rangle : C \Longrightarrow C'$ if $\tau : C \Longrightarrow C'$

(in) $\mathtt{in}(a \lhd v) : C \Longrightarrow C, a \lhd v$ if $a \in recep(C)$

(out) $\mathtt{out}(a \lhd v) : (C, a \lhd v) \lceil R \Longrightarrow C \lceil R \cup X$

if $a \notin recep(C)$, and $X = (acq(v) \cap recep(C)) - R$

X is the newly exported receptionists. [1]

[1] In the formulation of the input and output transitions we could easily allow concurrent (asynchronous) internal transitions to happen. This would not change the interaction semantics, and so we have chosen to use the simpler form here.

The computational behavior of a component is its set of fair computation paths.

Definition (Computation Paths, $Path(C)$): A computation path, π is an infinite sequence of interaction steps $\pi(i) = \gamma_i : C_i \Longrightarrow C'_i$ such that $C_{i+1} \sim C'_i$ for $i \in \mathbf{Nat}$. For technical reasons, we require that a path not use up the address space – there is a countable subset of $\mathbf{A}$ not occurring as a receptionist or external actor in the path. $Path(C)$ is the set of computation paths leading from C – computations paths, π, as above such that $C \sim C_0$.

Definition (Enabledness, Firing): An actor in C is always enabled (since an execution step is always possible). A message is enabled in C if its target is external, or if its target is internal and in a state that is enabled for delivery. An enabled actor is said to fire in an interaction step if an execution step for that actor occurs in the step. An enabled message is said to fire in an interaction step if a delivery or output step for that message occurs.

Definition (Observational Fairness, $OFair(C)$): A path, π, is observationally fair if an actor or message that becomes enabled at any stage C_i, either fires in some succeeding step or at some later stage it becomes permanently disabled. $OFair(C)$ is the subset of paths in $Path(C)$ that are observationally fair.

The reader may wonder why we call this *observational* fairness. There is a small subtlety having to do with the fact that we can not in general refer unambiguously to internal actors that are not receptionists, since the addresses are hidden and the actors may float around in the multiset (although in the examples we have considered this is not a problem). There are a number of ways to solve this problem. One is to first define computation at a less abstract level, omitting the structural equivalence rule (alpha). Having defined fairness of paths in this setting, we can reintroduce the alpha rule and use the induced notion of fairness. Details are omitted to saved space (and spare the reader).

4.4. Interaction semantics

The interaction semantics of a component hides the details of internal computation, replacing (fair) computation paths by interaction paths. An interaction path contains initial receptionist and external address sets together with an infinite sequence of interactions – silent, input, or output. In particular, component configurations can not be observed.

Definition (Interactions): The set of interactions, $Iact$, is the set of transition labels, $\langle\ \rangle$, $\mathtt{in}(a \triangleleft v)$, and $\mathtt{out}(a \triangleleft v)$. $Iact^\infty = [\mathbf{Nat} \to Iact]$ is the set of interaction sequences. We let γ range over $Iact$ and ζ range over $Iact^\infty$. An interaction path is a triple (R, E, ζ) where $\zeta \in Iact^\infty$ and $R, E \in \mathbf{P}_\omega[\mathbf{A}]$ (finite subsets of $\mathbf{A}$) with $R \cap E = \emptyset$.

Definition (Interaction Semantics): The interaction semantics of a component, $\mathcal{I}(C)$ is the abstraction of its fair computation paths.

$$\mathcal{I}(C) = \{ c2i(\pi) \bigm| \pi \in OFair(C) \}$$

The map, $c2i$, from computation paths to interactions paths, is defined as follows. If $\pi = [\gamma_i : C_i \Longrightarrow C'_i \bigm| i \in \mathbf{Nat}]$ and $\zeta(i) = \gamma_i$ for $i \in \mathbf{Nat}$, then

$c2i(\pi) = (recep(C_0), extrn(C_0), \zeta)$.

Definition (Interaction Equivalence): Two components are interaction equivalent just if they have the same interaction semantics.

$$C_0 \overset{i}{\cong} C_1 \Leftrightarrow \mathcal{I}(C_0) = \mathcal{I}(C_1)$$

Lemma (Congruence): $\overset{i}{\cong}$ is a congruence with respect to the component operations:

(mun) $C_0 \overset{i}{\cong} C_1 \Rightarrow (C\,,\; C_0) \overset{i}{\cong} (C\,,\; C_1)$

(rcp) $C_0 \overset{i}{\cong} C_1 \Rightarrow C_0 \lceil R \overset{i}{\cong} C_1 \lceil R$

4.5. Timer interaction semantics

We now give a more precise characterization of the timer interaction semantics, and indicate how it is derived from the definitions. First observe that the internal computations are of the form:

$$\tau : (\!\langle T(n) \rangle\!)_c\,,\; M_0, M_1 \Longrightarrow (\!\langle T(n+j) \rangle\!)_c\,,\; M_0, M_2$$

where M_1 has the form $\{c \triangleleft \mathtt{time}(x_i) \mid 1 \leq i \leq l\}$, M_2 has the form $\{x_i \triangleleft n_i \mid \leq 1 \leq i \leq l\}$, and $n \leq n_i \leq n + j$ for $j, l \in \mathbf{Nat}$.

Now, note that in any fair computation path, any message that is input will eventually be delivered, since the timer never becomes permanently disabled for message delivery. Thus, for every input of the form $\mathtt{in}(c \triangleleft \mathtt{time}(x))$, to the timer $(\!\langle T(n) \rangle\!)_c$, a message $x \triangleleft n'$ for some $n' \geq n$ will be generated, and by fairness each such message will eventually be output. Thus $\mathcal{I}((\!\langle T(n) \rangle\!)_c)$ is the set of interaction paths $(\{c\}, \emptyset, \zeta)$ such that no actors are exported (thus all inputs are to c), and there is a bijection F from indices of inputs of the form $\mathtt{in}(c \triangleleft \mathtt{time}(x))$ in ζ to indices of outputs in ζ such that

(F.1) if $\zeta(i) = \mathtt{in}(c \triangleleft \mathtt{time}(x))$, then $F(i) > i$ (requests arrive before replies are sent) and $\zeta(F(i)) = \mathtt{out}(x \triangleleft n')$ for some $n' \geq n$

(F.2) if $F^{-1}(i_1) > i_0$, $\zeta(F(i_0)) = \mathtt{out}(x_0 \triangleleft n_0)$, $\zeta(F(i_1)) = \mathtt{out}(x_1 \triangleleft n_1)$, then $n_1 \geq n_0$.

Clause (F.2) says that if the output at i_0 is of the form $\mathtt{out}(x_0 \triangleleft n_0)$ and the output, $\mathtt{out}(x_1 \triangleleft n_1)$, at i_1 is a reply to a request that arrived after the output at i_0, then the time reported at i_1 is at least as big at that reported at i_0.

A simple consequence of the above characterization is that if $(\{c\}, \emptyset, \zeta) \in \mathcal{I}((\!\langle T(n) \rangle\!)_c)$, such that $\zeta(2i) = \mathtt{in}(c \triangleleft \mathtt{time}(x_i))$ and $\zeta(2i+1) = \mathtt{out}(x_i \triangleleft \mathtt{time}(n_i))$ for $i \in \mathbf{Nat}$ (i.e. the interaction appears synchronized,) then $n_i \leq n_{i+1}$ for $i \in \mathbf{Nat}$.

5. Compositionality

We have observed that interaction equivalence is a congruence on the component algebra, thus we can think of the component operations acting on interaction equivalence classes. To show that interaction semantics is compositional, we need to define corresponding operations on sets of interaction paths. Here we focus on the composition operation. We first define the receptionists and externals effective at each stage of an interaction path in terms of the actors exported and imported by previous interactions. Then we define a composability relation on interaction paths and the composition operation on composables. Finally we define composition for sets of interaction paths and show the correspondence with composition of components. We illustrate compositionality by inferring the interaction semantics of the dispatcher from the semantics of its parts.

5.1. Composing Sets of Interaction Paths

Definition (Exports/Imports of interactions): For $R, E \in \mathbf{P}_\omega[\mathbf{A}]$ with $R \cap E = \emptyset$, we define the exports, $\mathit{Export}(R, E, \gamma)$ and imports, $\mathit{Import}(R, E, \gamma)$ of an interaction, γ, relative to receptionists, R, and externals, E, as follows.

$$\mathit{Export}(R, E, \langle\,\rangle) = \mathit{Import}(R, E, \langle\,\rangle) = \emptyset$$

$$\mathit{Export}(R, E, \mathtt{in}(a \lhd v)) = \mathit{Import}(\mathtt{out}(a \lhd v)) = \emptyset$$

$$\mathit{Import}(R, E, \mathtt{in}(a \lhd v)) = \mathit{Export}(R, E, \mathtt{out}(a \lhd v)) = \mathit{acq}(v) - (R \cup E)$$

Definition (Accumulated receptionists and externals): For $\zeta \in \mathit{Iact}^\infty$ and $R, E \in \mathbf{P}_\omega[\mathbf{A}]$ with $R \cap E = \emptyset$, we define the accumulated receptionists $\mathit{recep}((R, E, \zeta), i)$ and externals $\mathit{extrn}((R, E, \zeta), i)$ at stage i, by induction on i as follows:

$$\mathit{recep}((R, E, \zeta), 0) = R \quad \text{and} \quad \mathit{extrn}((R, E, \zeta), 0) = E$$

$$\mathit{recep}((R, E, \zeta), i+1) =$$

$$\mathit{recep}((R, E, \zeta), i) \cup \mathit{Export}(\mathit{recep}((R, E, \zeta), i), \mathit{extrn}((R, E, \zeta), i), \zeta(i))$$

$$\mathit{extrn}((R, E, \zeta), i+1) =$$

$$\mathit{extrn}((R, E, \zeta), i) \cup \mathit{Import}(\mathit{recep}((R, E, \zeta), i), \mathit{extrn}((R, E, \zeta), i), \zeta(i))$$

Lemma (ipath.acq): If π is a computation path, then $\mathit{c2i}(\pi)$ satisfies the following acquaintance laws.

(i) if $\zeta(i) = \mathtt{in}(a \lhd v)$, then $a \in \mathit{recep}((R, E, \zeta), i)$

(o) if $\zeta(i) = \mathtt{out}(a \lhd v)$, then $a \in \mathit{extrn}((R, E, \zeta), i)$

Definition (Composition of Interaction Paths): Two interaction paths (R_0, E_0, ζ_0), (R_1, E_1, ζ_1) are composable, written $(R_0, E_0, \zeta_0) \# (R_1, E_1, \zeta_1)$, with composition $(R, E, \zeta) = (R_0, E_0, \zeta_0) \parallel (R_1, E_1, \zeta_1)$ provided $R = R_0 \cup R_1$, $E = E_0 \cup E_1 - R$, and for $i \in \mathbf{Nat}$ the following hold:

(r.r) $\mathit{recep}((R_0, E_0, \zeta_0), i) \cap \mathit{recep}((R_1, E_1, \zeta_1), i) = \emptyset$

(r.e) $(\mathit{recep}((R, E, \zeta), i+1) - \mathit{recep}((R, E, \zeta), i)) - \mathit{extrn}(R, E, \zeta, i) = \emptyset$

and for $a \in \mathbf{A}$, $v \in \mathbf{V}$, ζ is defined by:

(tau) if $\zeta_0(i) = \langle\ \rangle$ and $\zeta_1(i) = \langle\ \rangle$, then $\zeta(i) = \langle\ \rangle$.

(out.0) if $\zeta_0(i) = \mathtt{out}(a \triangleleft v)$, then either

- $a \notin \mathit{recep}((R_1, E_1, \zeta_1), i)$, $\zeta_1(i) = \langle\ \rangle$, and $\zeta(i) = \mathtt{out}(a \triangleleft v)$, or
- $a \in \mathit{recep}((R_1, E_1, \zeta_1), i)$, $\zeta_1(i) = \mathtt{in}(a \triangleleft v)$, and $\zeta(i) = \langle\ \rangle$.

(out.1) (out.0) with 0,1 interchanged

(in.0) if $\zeta_0(i) = \mathtt{in}(a \triangleleft v)$, then either

- $\zeta_1(i) = \langle\ \rangle$, $a \in \mathit{recep}(R, E, \zeta), i)$, and $\zeta(i) = \mathtt{in}(a \triangleleft v)$, or
- $\zeta_1(i) = \mathtt{out}(a \triangleleft v)$, and $\zeta(i) = \langle\ \rangle$.

(in.1) (in.0) with 0,1 interchanged

The clauses (r.r) and (r.e) ensure that receptionist addresses from the two components do not conflict with one another or with external addresses. The condition, $a \in \mathit{recep}((R, E, \zeta), i)$, in the first case of (in.0) is needed to insure that composition obeys the acquaintance laws. It could fail, even thought the components are interaction paths, since an address exported by a component will not be exported by the composition if all exporting outputs are to the partner component.

Lemma (compose.isem): If $(R_0, E_0, \zeta_0) \# (R_1, E_1, \zeta_1)$, $(R_j, E_j, \zeta_j) \in \mathcal{I}(C_j)$ for $j < 2$, and $(R, E, \zeta) = (R_0, E_0, \zeta_0) \,\|\, (R_1, E_1, \zeta_1)$, then $(R, E, \zeta) \in \mathcal{I}(C_0\ ,\ C_1)$.

Proof (sketch): The proof relies on properties of the a corresponding algebra of computation paths alluded to in the proof sketch of the theorem below. □

Definition (Composing Interaction sets): Let P_j be sets of interaction paths for $j < 2$. The composition of P_0 and P_1 is defined by

$$P_0 \,\|\, P_1 = \{(R_0, E_0, \zeta_0) \,\|\, (R_1, E_1, \zeta_1) \;\big|\; (R_0, E_0, \zeta_0) \in P_0 \,\wedge\, (R_1, E_1, \zeta_1) \in P_1 \,\wedge \\ (R_0, E_0, \zeta_0) \# (R_1, E_1, \zeta_1)\}$$

Theorem (Composing Paths): Let C_0, C_1 be composable components. Then

$$\mathcal{I}(C_0) \,\|\, \mathcal{I}(C_1) = \mathcal{I}(C_0\ ,\ C_1)$$

Proof (sketch): We define composability and composition for computation paths and sets of computation paths in a similar manner. We also define decomposition for computation paths. Both composition and decomposition preserve fairness and composition of computational behaviors is the behavior of the composition. These properties together with the connection between computation paths and interaction paths suffice. □

5.2. The Dispatcher Construction

We now show how to compose the interaction semantics of a pre-dispatcher with that of a timer to obtain the interaction semantics of a dispatcher. First we characterize the interaction semantics of a pre-dispatcher. $\mathcal{I}((\mathtt{dis}(\vec{a}, z))_d)$ is the set of interaction paths $(\{d\}, \{\vec{a}, z\}, \zeta)$ such that for some $Y \subset \mathbf{A} - \{d, z, \vec{a}\}$ the following hold:

(1) if $\zeta(i) = \mathtt{in}(x \lhd v)$, then $x \in \{d\} \cup Y$, and if $\zeta(i) = \mathtt{out}(x \lhd v)$, then $x \in \{z, \vec{a}\}$;

(2) There is a bijection, G, from indices of inputs in ζ to Y, a bijection, H, from indices of inputs in ζ to indices of outputs in ζ with target z, and an injection, F, from indices of outputs with target in $\vec{a}$ to indices of inputs with target in Y such that if $\zeta(i) = \mathtt{out}(a_j \lhd v)$, with $a_j \in \vec{a}$, and $\zeta(F(i)) = \mathtt{in}(y \lhd v')$, with $y \in Y$, $G(i') = y$, then

- $v' \in \mathbf{Nat}$ and $j = v' \bmod k$
- $\zeta(i') = \mathtt{in}(d \lhd v$ and $\zeta(H(i')) = \mathtt{out}(z \lhd \mathtt{time}(y)$
- $i > F(i) > H(i') > i'$

G associates each input with the helper created to accept the reply from the timer, and H associates each input with the corresponding request sent to the timer. F associates each output to an element of $\vec{a}$ with the reply from the timer that triggers that output. Since in an arbitrary environment we can't guarantee that z really is a timer, there may be no inputs to the helper, or there may be several, and some of the inputs may not be numbers. Thus F is only an injection, not a bijection. $\mathcal{I}((\mathtt{dis}(\vec{a}, c))_d)$ is just $\mathcal{I}((\mathtt{dis}(\vec{a}, z))_d)$ with z replaced by c.

To compute the interaction paths of the composition

$$\mathtt{Dispatch}(\vec{a})[d] = ((\mathtt{dis}(\vec{a}, z))_d\,,\ (T(n))_c) \lceil d$$

we compute the interactions of $(\mathtt{dis}(\vec{a}, c))_d\,,\ (T(n))_c$ that have no external inputs to c. That is, we consider composable pairs of interaction paths such that all inputs to the timer component are matched by outputs from the dispatcher fragment. Since the exports from the pre-dispatcher occur only due to outputs to c, no newly created actor addresses will be exported from the composition. Thus all inputs to newly created actors must be matched by an output from the timer. This means each output to c by the pre-dispatcher has a unique corresponding input to the associated helper. Furthermore, each pre-dispatcher interaction meeting these requirements gives rise to an associated composition interaction by omitting the outputs to c and the inputs to auxiliary actors. To summarize $(\{d\}, \vec{a}, \zeta) \in \mathcal{I}(\mathtt{Dispatch}(\vec{a})[d])$ just if there is some ζ_d for the $(\mathtt{dis}(\vec{a}, c))_d$ component such that the injection F is a bijection and ζ is the subsequence of inputs to d and outputs to $\vec{a}$. Note that this gives rise to a bijection $\lambda j.G(tgt(\zeta_d(F(j))))$ from outputs to inputs. Thus $\mathcal{I}(\mathtt{Dispatch}(\vec{a})[d])$ is the set of interaction paths $(\{d\}, \{\vec{a}\}, \zeta)$ such that the following hold:

(1.i) no actors are exported, hence all inputs are to d,

(1.o) each output is of the form $\mathtt{out}(a_j \lhd v)$ for some $a_j \in \vec{a}$

(2) There is a bijection, F, from indices of outputs to indices of inputs such that if $\zeta(i) = \mathtt{out}(a_j \lhd v)$, then $i > F(i)$, and $\zeta(F(i)) = \mathtt{in}(d \lhd v)$.

5.3. Checking Equivalence Claims

We indicate how the equivalence claims at the end of §2 can be justified.

Lemma (D1): $((\mathtt{dis}(\vec{a}, c))_d\,,\ (T(0))_c) \lceil d \overset{i}{\cong} ((\mathtt{dis}(\vec{a}, c))_d\,,\ (T(n))_c) \lceil d$

Proof : This follows easily from the form of the characterization. □

Lemma (D2): $(\mathtt{Dispatch}(\vec{a})[d]\,,\ \mathtt{Fun}[\vec{a}]) \lceil d \overset{i}{\cong} \mathtt{Fun}[d]$

Proof : We must show that the two components have the same sets of interaction paths. Interaction paths of $(\mathtt{Dispatch}(\vec{a})[d]\,,\ \mathtt{Fun}[\vec{a}]) \lceil d$ are composites of paths ζ_d of $\mathtt{Dispatch}(\vec{a})[d]$ and ζ_f of $\mathtt{Fun}[\vec{a}]$ such that the outputs of ζ_d match the inputs of ζ_f. For any path ζ_d of $\mathtt{Dispatch}(\vec{a})[d]$ there is a matching path ζ_f of $\mathtt{Fun}[\vec{a}]$. Let H be the bijection from inputs of ζ_d to outputs, restricted to well-formed inputs for $\mathtt{Fun}[d]$ and let G be the bijection from well-formed inputs of ζ_f to outputs. The composition $\zeta_d \parallel \zeta_f$ has a bijection from well-formed inputs to outputs given by $G \circ H$. This bijection satisfies the conditions for a path of $\mathtt{Fun}[d]$. Furthermore any path of $\mathtt{Fun}[d]$ can be decomposed into a composable pair of paths as above. □

6. Discussion

The algebra of components is suggestive of the algebra of π-calculus processes [8]. Indeed, the parallel composition and renaming operations are quite similar. Our restriction operator is dual to the π-calculus hiding operator making explicit what is visible rather that what is to be hidden. Also interactions are similar to actions. However the underlying semantics is rather different. In our model addresses (names) refer to actors with state, not to stateless channels, interaction is asynchronous, and computations are required to be fair. Another difference is our use of interaction equivalence rather than more traditional notions such as forms of bisimulation or failure sets. Direct comparison with notions of equivalence based on bisimilarity is not easy, as they usually do not account for fairness. This is a topic for future work. We note that the bisimulation defined in the obvious way is unsound for interaction semantics.

An important next step is to consider larger examples, especially examples coming from real systems. What kinds of properties are of interest to system developers and users? The fact that in the simple examples, a characterization of the interaction semantics of composites can be derived from the characterizations of the components is encouraging. But, we need to develop high level abstractions and structured characterizations as well as scalable methods for composing them. Another direction for future work is to develop a specification language and corresponding reasoning principles (proof rules).

Models such as event structures or pomsets are closer in spirit and level of abstraction to our interaction semantics. These models make the causal ordering between events explicit whereas this information is implicit in the structure of sets of interaction paths. Another direction for future work is to characterize the structure of those sets of interaction paths that are reasonable candidates for the semantics of components. This probably means making the ordering information explicit using ideas from the work on event diagram [5].

Acknowledgements. The author would like to thank José Meseguer and her collaborators in actor research, Gul Agha, Ian Mason, and Scott Smith for many fruitful discussions, and two anonymous referees for helpful criticisms of an earlier draft. She would also like to thank Egidio Astesiano, Gianna Reggio, and Renato Romano for pointing out several corrections to an earlier version. This research was partially supported by ARPA grant NAVY N00014-94-1-0775, ONR grant N00014-94-1-0857, and NSF grant CCR-9312580.

7. References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation, 199? to appear.

[4] Henry G. Baker and Carl Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, August 1977.

[5] W. D. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[6] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[7] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[8] R. Milner, J. G. Parrow, and D. J. Walker. A calculus of mobile processes, Parts I and II. Technical Report ECS-LFCS-89-85, -86, Edinburgh University, 1989.

[9] N. Venkatasubramanian and C. L. Talcott. Reasoning about Meta Level Activities in Open Distributed Systems. In *Principles of Distributed Computation*, 1995.