Towards a Theory of Actor Computation

(Extended Abstract)

Gul Agha
University of Illinois
agha@cs.uiuc.edu

Ian A. Mason
Stanford University
iam@cs.stanford.edu

Scott Smith
Johns Hopkins University
scott@cs.jhu.edu

Carolyn Talcott
Stanford University
clt@sail.stanford.edu

Abstract

In this paper we present preliminary results of a rigorous development of the actor model of computation. We present an actor language which is an extension of a simple functional language, and provide a precise operational semantics for this extension. Our actor systems are open distributed systems, meaning we explicitly take into account the interface with external components in the specification of an actor system. We define and study various notions of equivalence on actor expressions and systems. We show that the usual tripartite family of testing equivalence relations collapses to two in the presence of fairness. We define a notion of operational bisimulation as a tool for establishing equivalence under fairness assumptions, and illustrate its use.

1. Introduction

The actor model of computation was originally proposed by Hewitt [5]. Actors are self-contained, concurrently interacting entities of a computing system. They communicate via message passing which is asynchronous and fair. Actors can be dynamically created and the topology of actor systems can change dynamically. The actor model is a primitive model of computation, but nonetheless easily expresses a wide range of computation paradigms. It directly supports encapsulation and sharing, and provides a natural extension of both functional programming and object style data abstraction to concurrent open systems [1, 2].

The main features of an open distributed system are that new components can be added, existing components can be replaced, and interconnections can be changed, largely without disturbing the functioning of the system. Components have no control over the components with which they might be connected. The behavior of a component is locally determined by its initial state and the history of its interactions with the environment through its interface. The internal state of a component must only be accessible through operations provided by the interface. Since the actor model has a built-in notion of local component and interface, it is a natural model to use as a basis for a theory of open distributed computation.

The semantics we define combines the message passing features of the primitive actor model with an applicative functional language for describing individual actor behaviors. This provides a simple yet expressive language with a manageable formal semantics. In our formalization we make explicit the notion of open system. An actor system is a collection of individually named concurrently executing actors,

plus two collections of actor names that define the interface to the environment. The *receptionists* ρ are names of actors within the system that external components may freely interact with; all other actors in the system are local and thus inaccessible from outside. The *external* actors χ are names of actors that are outside this actor system but to which messages may be directed. Each system is a self-contained entity, and we define operations for composing systems to form larger systems.

Most of the research in the area of parallel language design has either been practical but with very limited formal basis, or has been formal and theoretical but at the expense of realism. Our choices and approach are motivated by a desire to bridge the gap between theory and practice. The theory we develop is intended to be useful for justifying program transformations for real languages, and to formalize intuitive arguments and properties used by programmers.

Following the tradition of [9, 7] we adopt an operational interpretation of actor systems. Actor behavior is defined by a transition relation on configurations. Each configuration is a symbolic instantaneous representation of an actor system with respect to some idealized observer [1], and a transition on configurations maps configurations to possible future ones. Two actor expressions/systems are said to be observationally equivalent if they give rise to the same observations, suitably defined, inside all observing contexts. This notion is closely related to testing equivalence [4]. Observational equivalence provides a semantic basis for developing sound transformation rules.

Some highlights of this paper include the following. The operational semantics extends that of the embedded functional language in such a way that the equational theory of the functional language is preserved. We define a notion of open actor configuration which makes explicit the interface to the environment. As a first step towards an algebra of operations on open system components, we define a composition operator on configurations. An important aspect of the actor model is the fairness requirement: message delivery is guaranteed, and individual actor computations are guaranteed to progress. This makes the computation model more realistic: many intuitively correct equations fail in the absence of fairness. Although fairness makes some aspects of reasoning more complicated, it simplifies others. We prove that in the presence of fairness, the three standard notions of observational equivalence collapse to two. Finally, we give a simple bisimulation principle that allows equivalences to be established in the presence of fairness.

The remainder of this abstract is organized as follows. §2 gives the syntax and operational semantics of our actor language. §3 define various notions of equivalence and state their basic properties. §4 defines a notion of operational bisimulation as a sound approximation to operational equivalence and illustrates its use. §5 contains some concluding remarks.

Notation.

We use the usual notation for set membership and function application. Let Y, Y_0, Y_1 be sets. We specify meta-variable conventions in the form: let y range over Y, which should be read as: the meta-variable y and decorated variants such as y', y_0, ..., range over the set Y. Y^n is the set of sequences of elements of Y of length

n. Y^* is the set of finite sequences of elements of Y. $\bar{y} = [y_1, \ldots, y_n]$ is the sequence of length n with ith element y_i. (Thus [] is the empty sequence.) $u * v$ denotes the concatenation of the sequences u and v. $\mathbf{P}_\omega[Y]$ is the set of finite subsets of Y. $\mathbf{M}_\omega[Y]$ is the set of (finite) multi-sets with elements in Y. $[Y_0 \to Y_1]$ is the set of total functions, f, with domain Y_0 and range contained in Y_1. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. $Fmap[Y_0, Y_1]$ is the set of finite maps from Y_0 to Y_1.

2. A Simple Lambda Based Actor Language

Actors are self-contained, components of a computing system that communicate by asynchronous message passing. Message delivery is guaranteed (fairness). In response to a message an actor can send messages to actors that it knows about, and create new actors. It can also change its state/behavior. This change will be in effect when the next message is received by the actor, and the only time an actor's local state changes is when the actor changes it in response to a message. (Local cause for local effect principle).

Our actor language is an extension of the call-by-value lambda calculus that includes (in addition to arithmetic primitives and structure constructors, recognizers, and destructors) primitives for creating and manipulating actors. An actor's behavior is described by a closure which embodies the code to be executed when a message is received, and the local store (values bound to free variables). The actor primitives are: `send` (for sending messages); `become` (for changing behavior); and `newadr` and `initbeh` (for actor creation). `send`(a, v) creates a new message with receiver a and contents v and puts the message into the message delivery system. `become`(b) clones an anonymous actor to carry out the rest of the current computation, alters the behavior of the actor executing the become to b, and frees that actor to accept another message. The cloned actor may send messages or create new actors in the process of completing its computation, but will never receive any messages as its address can never be known. `newadr`() creates a new (uninitialized) actor and returns its address. `initbeh`(a, b) initializes the behavior of a newly created actor with address a to be b. An uninitialized actor can only be initialized by the actor which created it. Without this restriction composability of actor systems is problematic, as it would permit an external actor to initialize an internally created actor. The allocation of a new address and initialization of the actor's behavior have been separated in order to allow an an actor to know its own address. This is a weak form of synchronization and would not be necessary if message sending were synchronous. An alternative would be to have built into the semantics that every actor knows its own name, as is done in many actor and object-oriented languages. See [1] for intuitions behind these constructs.

2.1. Syntax

We take as given countable sets $\mathbb{X}$ (variables) and $\mathbb{A}\mathrm{t}$ (atoms). $\mathbb{F}_n$ is the set of primitive operations of rank n and $\mathbb{F} = \bigcup_{n \in \mathbb{N}} \mathbb{F}_n$. We assume $\mathbb{A}\mathrm{t}$ contains `t`, `nil`

for booleans, as well as integers. $\mathbb{F}$ contains arithmetic operations, branching `br` (rank 3), pairing $\texttt{ispr}, \texttt{pr}, \texttt{1}^{\texttt{st}}, \texttt{2}^{\texttt{nd}}$ (ranks 1, 2, 1, 1), and actor primitives `newadr`, `initbeh`, `send`, and `become` (ranks 0, 2, 2, 1). The sets of expressions, $\mathbb{E}$, value expressions, $\mathbb{V}$, and contexts (expressions with holes), $\mathbb{C}$, are defined inductively as follows.

Definition ($\mathbb{E}$, $\mathbb{V}$, $\mathbb{C}$):

$$\mathbb{V} = \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \lambda\mathbb{X}.\mathbb{E} \cup \texttt{pr}(\mathbb{V}, \mathbb{V})$$

$$\mathbb{E} = \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \lambda\mathbb{X}.\mathbb{E} \cup \texttt{app}(\mathbb{E}, \mathbb{E}) \cup \mathbb{F}_n(\mathbb{E}^n)$$

$$\mathbb{C} = \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \lambda\mathbb{X}.\mathbb{C} \cup \texttt{app}(\mathbb{C}, \mathbb{C}) \cup \mathbb{F}_n(\mathbb{C}^n) \cup \{\varepsilon\}$$

We let x, y, z range over $\mathbb{X}$, v range over $\mathbb{V}$, e range over $\mathbb{E}$, and C range over $\mathbb{C}$. $\lambda x.e$ binds the variable x in the expression e. We write $\mathrm{FV}(e)$ for the set of free variables of e. We write $e\{x := e'\}$ to denote the expression obtained from e by replacing all free occurrences of x by e', avoiding the capture of free variables in e'. Contexts are expressions with holes. We use ε to denote a hole. $C[\![e]\!]$ denotes the result of replacing any holes in C by e. Free variables of e may become bound in this process. `let`, `if` and `seq` are the usual syntactic sugar, `seq` being a sequencing primitive.

A simple actor behavior b that expects its message to be an actor address, sends the message 5 to that address, and becomes the same behavior, may be expressed using a definable call-by-value fixed-point combinator `rec` (cf. [7]) as follows.

$$b = \texttt{app}(\texttt{rec}, \lambda y.\lambda x.\texttt{seq}(\texttt{become}(y), \texttt{send}(x, 5)))$$

An expression that would create an actor with this behavior and send it some other actor address a is

$$e = \texttt{let}\{x := \texttt{newadr}()\}\texttt{seq}(\texttt{initbeh}(x, b), \texttt{send}(x, a)).$$

The behavior of a sink, an actor that ignores its messages and becomes this same behavior, is defined by

$$\texttt{sink} = \texttt{rec}(\lambda b.\lambda m.\texttt{become}(b)).$$

2.2. Reduction Semantics for Open Configurations

We give the semantics of actor expressions by defining a transition on open configurations. Open configurations describe actor systems in which addresses of some (but not necessarily all) of the actors are known to the outside world. These actors are called receptionists. An open configuration may also know addresses of some actors in the outside world. These actors are called external actors. The sets of receptionists and external actors are the interface of an actor system to its environment. They specify what actors are visible and what actor connections must be provided for the system to function. The set of receptionists may grow and the set of required external connections may shrink as the system evolves. In addition, an open configuration contains an actor map and a multi-set of pending messages. An actor map is a finite map from actor addresses to actor states. An actor state is

either uninitialized (having been newly created by an actor, a) written $(?_a)$; ready to accept a message, written (b) where b is its behavior, a lambda abstraction; or busy executing e, written $[e]$, here e represents the actor's current (local) processing state. A message contains the address of the actor to whom it is sent and the message contents. The contents can be any value constructed from atoms and actor addresses using constructors.

Lambda abstractions and constructions containing lambda abstractions are not allowed to be communicated in messages. There are two reasons for this restriction. Firstly, allowing lambda abstractions to be communicated in values violates the actor principle that only an actor can change its own behavior, because a `become` in a lambda message may change the receiving actor behavior. Secondly, if lambda abstractions are communicated to external actors, there is no reasonable way to control what actor addresses are actually exported. This has unpleasant consequences in reasoning about equivalence, amongst other things. This restriction is not a serious limitation since the address of an actor whose behavior is the desired lambda abstraction can be passed in a message. Thirdly, if lambda abstractions can be communicated in messages then syntactic extensions to the language that involve transformations such as CPS can not be done on a per actor basis, since it would requires transformation of code that might arrive in a message. We classify transitions as internal or external. The internal transitions of a configuration are:

(1) an actor executing a step of its current computation;

(2) an actor initializing the behavior of a newly created actor; and

(3) acceptance of a message by an actor not currently busy computing.

The transitions of class (1) involve a single actor. They may be purely internal (a λ-transition), or messages may be sent, or a new actor may be created. The transitions of class (2) involve two actors, and the initialized actor becomes ready to accept a message. The transitions of class (3) involve an actor and a message. The message is consumed and the actor becomes busy. In addition to the internal transitions of a configuration, there are transitions that correspond to interactions with external agents:

(4) arrival of a message to a receptionist from the outside; and

(5) passing a message out to an external actor.

We assume that we are given a countable set $\mathbb{A}\mathrm{d}$ of actor addresses. To simplify notation, we identify $\mathbb{A}\mathrm{d}$ with $\mathbb{X}$. This pun is useful for two reasons: it allows us to use expressions to describe actor states and message contents; and it allows us to avoid problems of choice of names for newly created actors by appealing to an extended form of alpha conversion. (See [7] for use of this pun to represent reference cells.)

Definition ($\mathrm{c}\mathbb{V}$, $\mathbb{A}\mathrm{s}$, $\mathbb{M}$): The set of communicable values, $\mathrm{c}\mathbb{V}$, the set of actor states, $\mathbb{A}\mathrm{s}$, and the set of messages, $\mathbb{M}$, are defined as follows.

$$\mathrm{c}\mathbb{V} = \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \mathbf{pr}(\mathrm{c}\mathbb{V}, \mathrm{c}\mathbb{V}) \qquad \mathbb{A}\mathrm{s} = (?_{\mathbb{X}}) \cup (\mathbb{L}) \cup [\mathbb{E}] \qquad \mathbb{M} = \langle \mathbb{X} \Leftarrow \mathrm{c}\mathbb{V} \rangle$$

We let cv range over $\mathrm{c}\mathbb{V}$.

Definition (Actor Configurations): An actor configuration with actor map, α, multi-set of messages, μ, receptionists, ρ, and external actors, χ, is written

$$\left\langle\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho}$$

where $\rho, \chi \in \mathbf{P}_{\omega}[\mathbb{X}]$, $\alpha \in \mathit{Fmap}[\mathbb{X}, \mathbb{A}\mathrm{s}]$, and $\mu \in \mathbf{M}_{\omega}[\mathbb{M}]$. Further, it is required that, letting $A = \mathrm{Dom}(\alpha)$, the following constraints are satisfied:

(0) $\rho \subseteq A$ and $A \cap \chi = \emptyset$,

(1) if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$, and if $\alpha(a) = (?_{a'})$, then $a' \in A$,

(2) if $<a \Leftarrow v> \in \mu$, then $\mathrm{FV}(v) \cup \{a\} \subseteq A \cup \chi$.

We let κ range over actor configurations. A configuration in which both the receptionist and external actor sets are empty is said to be *closed*. For closed configurations we may omit explicit mention of the empty sets. The actor map portion of a configuration is presented as a list of actor states each subscripted by the actor address which is mapped to this state. $\alpha, (b)_a$ denotes the map α' such that $\mathrm{Dom}(\alpha') = \mathrm{Dom}(\alpha) \cup \{a\}$, $\alpha'(a) = (b)$, and $\alpha'(a') = \alpha(a)$ if $a' \neq a$. Similarly for other states subscripted with addresses. We use _ to denote a fresh address whose actual name we do not care about. Such addresses refer to actors not known to any other actors (anonymous actors). In a configuration, there may be multiple occurrences of actor states with address represented by _. These are in fact distinct, and simply reflect that the choice of address is irrelevant.

The set of possible computations of an actor configuration is defined in terms of the transition relation $\mapsto$ on configurations. To describe the internal transitions other than message receipt, an expression is decomposed into a reduction context filled with a redex. Reduction contexts are expressions with a unique hole, that play the role of continuations in abstract machine models of sequential computation. We have defined the decomposition to correspond to a left-most, outer-most, call-by-value evaluation order, thus preserving the semantics of the embedded functional language. Decomposition of non-value expressions is unique. Thus, locally computation is deterministic.

Definition ($\mathbb{E}_{\mathtt{rdx}}$,$\mathbb{R}$): The set of redexes, $\mathbb{E}_{\mathtt{rdx}}$, and the set of reduction contexts, $\mathbb{R}$, are defined by

$$\mathbb{E}_{\mathtt{rdx}} = \mathtt{app}(\mathbb{V}, \mathbb{V}) \cup (\mathbb{F}_n(\mathbb{V}^n) - \mathtt{pr}(\mathbb{V}, \mathbb{V}))$$

$$\mathbb{R} = \{\varepsilon\} \cup \mathtt{app}(\mathbb{R}, \mathbb{E}) \cup \mathtt{app}(\mathbb{V}, \mathbb{R}) \cup \mathbb{F}_{n+m+1}(\mathbb{V}^n, \mathbb{R}, \mathbb{E}^m)$$

We let R range over $\mathbb{R}$.

Redexes can be split into two classes: purely functional and actor redexes. Reduction rules for the purely functional case are given by a relation $\overset{\lambda}{\mapsto}$ on expressions. They correspond to the usual operational semantics for the purely functional fragment of our actor language and we omit them from this abstract. The actor redexes are: $\mathtt{newadr}()$, $\mathtt{initbeh}(a, b)$, $\mathtt{become}(b)$, and $\mathtt{send}(a, v)$.

Definition ($\mapsto$): The single-step transition relation $\mapsto$, on actor configurations is the least relation satisfying the following conditions.

f(a) $\qquad e \overset{\lambda}{\mapsto} e' \;\Rightarrow\; \left\langle\!\left\langle\, \alpha, [e]_a \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho} \mapsto \left\langle\!\left\langle\, \alpha, [e']_a \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho}$

$\mathbf{n}(a,a)$ $\quad \left\langle\!\!\left\langle \alpha, [R[\![\texttt{newadr()}]\!]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[\![a']\!]]_a, (?_a)_{a'} \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$ $\quad a'$ fresh

$\mathbf{c}(a)$ $\quad \left\langle\!\!\left\langle \alpha, [R[\![\texttt{initbeh}(a', b)]\!]]_a, (?_a)_{a'} \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[\![\texttt{nil}]\!]]_a, (b)_{a'} \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$

$\mathbf{b}(a)$ $\quad \left\langle\!\!\left\langle \alpha, [R[\![\texttt{become}(b)]\!]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[\![\texttt{nil}]\!]]_{_}, (b)_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$

$\mathbf{s}(a, a', cv)$ $\quad \left\langle\!\!\left\langle \alpha, [R[\![\texttt{send}(a', cv)]\!]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[\![\texttt{nil}]\!]]_a \mid \mu, <a' \Leftarrow cv> \right\rangle\!\!\right\rangle^{\rho}_{\chi}$

$\mathbf{r}(a, cv)$ $\quad \left\langle\!\!\left\langle \alpha, (b)_a \mid <a \Leftarrow cv>, \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [\texttt{app}(b, cv)]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$

$\mathbf{o}(a, cv)$ $\quad \left\langle\!\!\left\langle \alpha \mid <a \Leftarrow cv>, \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha \mid \mu \right\rangle\!\!\right\rangle^{\rho'}_{\chi}$

where $\rho' = \rho \cup (\mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha))$ and $a \in \chi$

$\mathbf{i}(a, cv)$ $\quad \left\langle\!\!\left\langle \alpha \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha \mid \mu, <a \Leftarrow cv> \right\rangle\!\!\right\rangle^{\rho}_{\chi \cup (\mathrm{FV}(cv) - \mathrm{Dom}(\alpha))}$

provided $a \in \rho$ and $\mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha) \subseteq \rho$

Note that in the last four rules the message contents are restricted to be communicable values. $\stackrel{*}{\mapsto}$ is the transitive reflexive closure of $\mapsto$. The configurations reachable from a given configuration κ are those configurations κ' such that $\kappa \stackrel{*}{\mapsto} \kappa'$.

The transitions are labelled to facilitate some technical definitions. We write $\kappa_0 \stackrel{l}{\longrightarrow} \kappa_1$ if $\kappa_0 \mapsto \kappa_1$ according to the rule encoded by l. We say l is enabled in configuration κ if there is some κ' such that $\kappa \stackrel{l}{\longrightarrow} \kappa'$.

Definition (Computation trees and paths): If κ is a configuration, then we define $\mathcal{T}(\kappa)$ to be the set of all finite sequences of labeled transitions of the form $[\kappa_i \stackrel{l_i}{\longrightarrow} \kappa_i' \mid i < n]$ for some $n \in \mathbb{N}$ such that $\kappa_0 = \kappa$ and $\kappa_i' = \kappa_{i+1}$ for $i < n-1$. We call such sequences *nodes* and let ν range over nodes. We order nodes of a tree by the subtree relation: $\nu_0 < \nu_1$ iff ν_0 is below (properly extends) ν_1. A computation path for κ is a maximal linearly ordered set of nodes in $\mathcal{T}(\kappa)$. Note that a computation path can also be regarded as a (possibly infinite) sequence of transitions. We let π range over computation paths and use $\mathcal{T}^{\infty}(\kappa)$ to denote the set of all such π for $\mathcal{T}(\kappa)$.

We now rule out those computations that are unfair, i.e. those that either starve out a particular actor computation, or keep a message queued forever when the receiving actor is either external or has infinitely often been ready to receive a message.

Definition (Fair computation paths): A computation path $\pi = [\nu_i \stackrel{l_i}{\longrightarrow} \nu_{i+1} \mid i \in I]$ in the computation tree $\mathcal{T}(\kappa)$ is fair if each enabled transition eventually happens or becomes permanently disabled. That is, if l is enabled in κ_i then $\kappa_j \stackrel{l}{\longrightarrow} \kappa_{j+1}$ for some $j \geq i$, or l has the form $\mathbf{r}(a, cv)$ and for some $j \geq i$ a is busy and never again becomes ready to accept a message. For a configuration κ we define $\mathcal{T}_{\mathrm{f}}^{\infty}(\kappa)$ to be the subset of $\mathcal{T}^{\infty}(\kappa)$ that are fair.

Note that finite computation paths are fair, since all of the enabled transitions must have happened.

Actor systems compose well, as indicated by the following definition and theorem.

Definition (Composition of Open Configurations): Two open configurations $\kappa_i = \left\langle\!\!\left\langle\, \alpha_i \mid \mu_i \,\right\rangle\!\!\right\rangle_{\chi_i}^{\rho_i}$, $i < 2$ are composable if $\mathrm{Dom}(\alpha_0) \cap \mathrm{Dom}(\alpha_1) = \emptyset$. The composition $\kappa_0 \cup \kappa_1$ is defined by

$$\kappa_0 \cup \kappa_1 = \left\langle\!\!\left\langle\, \alpha_0 \cup \alpha_1 \mid \mu_0 \cup \mu_1 \,\right\rangle\!\!\right\rangle_{(\chi_0 \cup \chi_1) - (S_0 \cup S_1)}^{(\rho_0 \cup \rho_1) - (S_0 \cup S_1)}$$

where $S_0 = \chi_0 \cap \rho_1$ and $S_1 = \chi_1 \cap \rho_0$.

Theorem (Composition of Open Configurations): There exists a binary operation $\mathcal{M}$ on computation trees such that if κ_i are composable configurations then

$$\mathcal{T}(\kappa_0 \cup \kappa_1) = \mathcal{M}(\mathcal{T}(\kappa_0), \mathcal{T}(\kappa_1))$$

where S_0, S_1, and $\kappa_0 \cup \kappa_1$ are as above.

In brief, the operation $\mathcal{M}$ merges pairs of computations that have matching i/o transitions for those external actors of one system that are (identified with) receptionists of the other system. Note that this theorem fails if arbitrary actors are allowed to initialize the behavior of newly created actors.

3. Notions of Equivalence for Actors

Two forms of equivalence are given, one for expressing the equivalence of actor expressions, and another for expressing the equivalence of actor configurations. We base our notion of equivalence on the now classic *operational equivalence* of [9]. For the deterministic functional languages of the sort Plotkin studied, this equality is defined as follows. Two program expressions are said to be equivalent if they behave the same when placed in any observing context, where an observing context is some complete program with a hole, such that all of the free variables in the expressions being observed are captured when the expressions are placed in the hole. The notion of "behave the same" is (for deterministic functional languages) typically equi-termination, i.e. either both converge or both diverge.

3.1. Equivalence of actor expressions

We first define equivalence of actor expressions, the equivalence of actor configurations will be defined later. The first step is to find proper notions of "observing context" and "behave the same" in an actor setting. For actor expressions, the analogue of observing context is an observing actor configuration that contains an actor with a hole in which the expression to be observed is placed. Since termination is not particularly relevant for actor configurations, we instead introduce an observer

primitive, **event** and observe whether or not in a given computation, **event** is executed. This approach is similar to that used in testing equivalence for CCS [4]. Since the language is nondeterministic, three different observations may be made instead of two: either **event** occurs for all possible executions, it occurs in some executions but not others, or it never occurs.

Formally, the language of observing contexts is obtained by introducing a new 0-ary primitive operator, **event**. We extend the reduction relation $\mapsto$ by adding the following rule.

$$\mathbf{e}(a) \quad \left\langle\!\!\left\langle\, \alpha,\, [R[\![\texttt{event()}]\!]]_a \;\middle|\; \mu \,\right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle\, \alpha,\, [R[\![\texttt{nil}]\!]]_a \;\middle|\; \mu \,\right\rangle\!\!\right\rangle^{\rho}_{\chi}$$

For an expression e, the observing configurations are configuration contexts of the form $\left\langle\!\!\left\langle\, \alpha,\, [C[\![\ ]\!]]_a \;\middle|\; \mu \,\right\rangle\!\!\right\rangle$ over the extended language, such that filling the hole in $C[\![\ ]\!]$ with e results in a closed configuration. (Let $\mathbb{K}$ be the set of configuration contexts (configurations with holes), and let K range over $\mathbb{K}$.

We observe **event** transitions in the fair computation paths. We say that a computation path succeeds (**s**) if an **event** transition occurs in it, otherwise it fails (**f**). $obs(\pi)$ is the **s**/**f** observation of a single complete computation π, and $Obs(\kappa)$ is the set of observations possible for a closed actor configuration.

Definition (observations): Let κ be a configuration of the extended language, and let $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i \in I]$ be a fair computation path, i.e. $\pi \in \mathcal{T}_{\mathrm{f}}^{\infty}(\kappa)$. Define

$$obs(\pi) = \begin{cases} \mathbf{s} & \text{if } (\exists \kappa_0, \kappa_1, a)(\kappa_0 \xrightarrow{\mathrm{e}(a)} \kappa_1 \in \pi) \\ \mathbf{f} & \text{otherwise} \end{cases}$$

$$Obs(\kappa) = \begin{cases} \mathbf{s} & \text{if } (\forall \pi \in \mathcal{T}_{\mathrm{f}}^{\infty}(\kappa))(obs(\pi) = \mathbf{s}) \\ \mathbf{sf} & \text{if } (\exists \pi \in \mathcal{T}_{\mathrm{f}}^{\infty}(\kappa))(obs(\pi) = \mathbf{s}) \text{ and } (\exists \pi \in \mathcal{T}_{\mathrm{f}}^{\infty}(\kappa))(obs(\pi) = \mathbf{f}) \\ \mathbf{f} & \text{if } (\forall \pi \in \mathcal{T}_{\mathrm{f}}^{\infty}(\kappa))(obs(\pi) = \mathbf{f}) \end{cases}$$

The natural notion of operational equivalence is that equal observations are made in all closing configuration contexts. It may be desirable in some cases to consider using a weaker equality, however. An **sf** observation may be considered as good as an **s** observation, and a new equivalence arises if these observations are equated. Similarly, an **sf** observation may be as bad as an **f** observation. We may thus define the following three equivalences.

Definition ($\cong_{1,2,3}$):

(1) $e_0 \cong_1 e_1$ (testing or convex or Plotkin or Egli-Milner) iff $Obs(K[\![e_0]\!]) = Obs(K[\![e_1]\!])$ for all closing configuration contexts K

(2) $e_0 \cong_2 e_1$ (must or upper or Smyth) iff $Obs(K[\![e_0]\!]) =_{(\mathrm{sf=f})} Obs(K[\![e_1]\!])$ for all closing configuration contexts K

(3) $e_0 \cong_3 e_1$ (may or lower or Hoare) iff $Obs(K[\![e_0]\!]) =_{(\mathrm{sf=s})} Obs(K[\![e_1]\!])$ for all closing configuration contexts K

where $x =_{(o=o')} y$ iff $x = y$ or $x, y \in \{o, o'\}$.

Note that may-equivalence ($\cong_3$) depends only on the computation trees, not on the choice of paths admitted as computations, because all **events** are observed after

some finite amount of time. This means it is independent of whether or not fairness is required. Since fairness sometimes makes proving equivalences more difficult, it is useful that may-equivalence can always be proved ignoring the fairness assumption. The other two equalities are sensitive to choice of paths admitted as computations, in particular when fairness is required, as in our model, $\cong_2$ is in fact the same as $\cong_1$. In models without the fairness requirement, they are distinct. In either case, $\cong_3$ is distinct from $\cong_1$ and $\cong_2$.

Theorem (partial collapse):

(1=2) $e_0 \cong_2 e_1$ iff $e_0 \cong_1 e_1$

(1-3) $e_0 \cong_1 e_1$ implies $e_0 \cong_3 e_1$

(3-1) $e_0 \cong_3 e_1$ does not imply $e_0 \cong_1 e_1$

Proof (partial collapse):

2-1 $\cong_1$ implies $\cong_2$ follows from the definitions. The key to showing that $\cong_2$ implies $\cong_1$ is the observation that if $Obs(K[\![e_0]\!]) = \mathbf{f}$ and $Obs(K[\![e_1]\!]) = \mathbf{sf}$ it is always possible to construct a K^* such that $Obs(K^*[\![e_0]\!]) = \mathbf{s}$, and $Obs(K^*[\![e_1]\!]) = \mathbf{sf}$. To see this, suppose that K satisfies the hypothesis. Form K' by replacing all occurrences of `event()` in K by `send(`a`,nil)` for some fresh variable a. Let K^* be obtained by adding to K' a message <$a \Leftarrow$ `t`> and an actor a where a has the following behavior: If a receives the message `t`, it executes `event()` and becomes a sink, and if a receives the message `nil`, it just becomes a sink. Recall that a sink is an actor that ignores its message and becomes a sink. We claim K^* is the desired observing context. If $K[\![e_0]\!]$ never executes `event()`, then in any fair complete computation, the `t` message will be received by a, so $K^*[\![e_0]\!]$ will always execute `event()`. If $K[\![e_1]\!]$ executes `event()` in some computation, then in the corresponding computations for $K^*[\![e_1]\!]$, sometimes `nil` will be received by a before `t` is received and sometimes it won't, hence $K^*[\![e_1]\!]$ will execute `event()` in some computations, but not in all. $\square_{2-1}$

1-3 from the definitions. $\square_{1-3}$

3-1 We construct expressions e_0, e_1 such that $e_0 \cong_3 e_1$, but $\neg(e_0 \cong_2 e_1)$. Let e_0 create an actor that sends a message (say `nil`) to an external actor a and becomes a sink, and let e_1 create an actor that may or may not send a message `nil` to a depending on a coin flip (there are numerous methods of constructing coin flipping actors), and also then becomes a sink. Let K be an observing configuration context that with an actor a that executes `event` just if `nil` is received. Then $Obs(K[\![e_0]\!]) = \mathbf{s}$ but $Obs(K[\![e_1]\!]) = \mathbf{sf}$, so $\neg(e_0 \cong_2 e_1)$. To show that $e_0 \cong_3 e_1$, show for arbitrary K that some path in the computation of $K[\![e_0]\!]$ contains an event iff some path in the computation of $K[\![e_1]\!]$ contains an event. This is easy, because when e_1's coin flip indicates `nil` is sent, the computation proceeds identically to e_0's computation. $\square_{3-1}$

$\square$

Hereafter, $\cong$ (operational equivalence) will be used as shorthand for either $\cong_1$ or $\cong_2$. A possibly useful analogy is that $\cong_3$ corresponds to partial correctness and $\cong$ corresponds to total correctness.

The fairness requirement is critical in the proof of (**2-1**). For example in CCS, where fairness is not assumed, no such collapse of $\cong_2$ to $\cong_1$ occurs. So, although fairness complicates some aspects of the theory, it simplifies others. If we omitted the fairness requirement we could make more $\cong$-distinctions between actors. For example, let a_0 be a sink. Let a_1 be an actor that also ignores its messages and becomes the same behavior, but it continues executing an infinite loop. The infinite looping actor could starve out the rest of the configuration, but in the presence of fairness no such starvation can occur, so the two are equal.

Since our reduction rules preserve the evaluation semantics of the embedded functional language, many of the equational laws for this language (cf. [11]) continue to hold in the full actor language. For example, operational equivalence is a congruence and the laws concerning lambda abstraction and application continue to hold.

Theorem (lambda laws):

(cong) $e_0 \cong e_1 \Rightarrow C[\![e_0]\!] \cong C[\![e_1]\!]$

(betav) $\texttt{let}\{x := v\}e = (\lambda x.e)(v) \cong e\{x := v\}$

(app) $e_0(e_1) \cong (\lambda f.f(e_1))(e_0) = \texttt{let}\{f := e_0\}f(e_1)$

(cmps) $f(g(e)) \cong (\lambda x.f(g(x)))(e) = (f \circ g)(e)$

(id) $\texttt{let}\{x := e\}x = (\lambda x.x)(e) \cong e$

The proof of this theorem uses the notion of bisimulation (cf. the next section).

3.2. Equivalence of actor configurations

Equivalence is now defined for open actor configurations $\left\langle\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho}$. As with actor expressions, we wish to close the open configuration by adding observers. This produces a notion of equivalence for actor configurations that is closely connected with equivalence of actor expressions.

Definition (Closing an Actor Configuration): A closing of an actor configuration $\kappa = \left\langle\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho}$ is defined to be an actor configuration $\kappa' = \left\langle\!\left\langle\, \alpha' \mid \mu' \,\right\rangle\!\right\rangle_{\rho}^{\chi}$, in the extended language, composable with κ.

Definition ($\cong_s$): $\kappa_0 = \left\langle\!\left\langle\, \alpha_0 \mid \mu_0 \,\right\rangle\!\right\rangle_{\chi}^{\rho} \cong_s \left\langle\!\left\langle\, \alpha_1 \mid \mu_1 \,\right\rangle\!\right\rangle_{\chi}^{\rho} = \kappa_1$ iff $Obs(\kappa_0 \cup \kappa') = Obs(\kappa_0 \cup \kappa')$ for all actor configurations κ' closing κ_j, $j < 2$.

Theorem ($\cong$ / $\cong_s$): If $e_0 \cong e_1$ and $e_0' \cong e_1'$, then

$$\left\langle\!\left\langle\, \alpha, [C[\![e_0]\!]]_a, (\lambda x.C'[\![e_0']\!])_{a'} \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho} \cong_s \left\langle\!\left\langle\, \alpha, [C[\![e_1]\!]]_a, (\lambda x.C'[\![e_1']\!])_{a'} \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho}.$$

Note that while two closed configurations (configurations that have no receptionists and no external actors) cannot be distinguished by any external observation, two closed expressions can be distinguished by a behavior context that makes use of the values returned.

4. Operational Bisimulations

Given two computation trees $\mathcal{T}(\kappa_0)$ and $\mathcal{T}(\kappa_1)$ of actor configurations κ_0 and κ_1, we define the notion of an operational bisimulation, $R \subseteq \mathcal{T}(\kappa_0) \times \mathcal{T}(\kappa_1)$, in such a way as to ensure that if two computation trees, $\mathcal{T}(\kappa_0)$ and $\mathcal{T}(\kappa_1)$, are operationally bisimilar, then $\kappa_0 \cong_s \kappa_1$. We view operational bisimulation as a proof technique, not as an alternative notion of equivalence. To keep the notation somewhat under control, we shall treat a simple case in this paper. A computation tree is *non-expansive* iff the set of receptionists never increases. We say a configuration is *non-expansive* iff its computation tree is. In what follows we restrict our attention to non-expansive trees. Extending the results to expansive configurations poses only notational complications (e.g identifying newly created actors), and the non-expansive case suffices to prove operational equivalences.

The definition of an operational bisimulation requires a little notation. Firstly, an $R \subseteq \mathcal{T}(\kappa_0) \times \mathcal{T}(\kappa_1)$, naturally extends to an $R \subseteq \mathcal{T}^\infty(\kappa_0) \times \mathcal{T}^\infty(\kappa_1)$ as follows. For $\pi_i \in \mathcal{T}^\infty(\kappa_i)$ for $i < 2$

$$\pi_0 R \pi_1 \quad \text{iff} \quad (\forall \nu_0' \in \pi_0)(\forall \nu_1' \in \pi_1)(\exists \nu_0'' \in \pi_0)(\exists \nu_1'' \in \pi_1)(\nu_0'' < \nu_0' \wedge \nu_1'' < \nu_1' \wedge \nu_0'' R \nu_1'')$$

(Recall that $\nu < \nu'$ is the subtree relation on nodes.) Secondly, for $\nu = [\kappa_i \xrightarrow{l_i} \kappa_i' \mid i < n]$, the `i/o` restriction $(\nu)_{\mathrm{io}}$, is defined to be $(l_0)_{\mathrm{io}} * (l_1)_{\mathrm{io}} * \ldots * (l_{n-1})_{\mathrm{io}}$, where

$$(l)_{\mathrm{io}} = \begin{cases} [\mathbf{o}(a, cv)] & \text{if } l = \mathbf{o}(a, cv) \\ [\mathbf{i}(a, cv)] & \text{if } l = \mathbf{i}(a, cv) \\ [\mathbf{e}()] & \text{if } (\exists a)(l = \mathbf{e}(a)) \\ [] & \text{otherwise} \end{cases}$$

Definition (operational bisimulation): Given two *non-expansive* actor configurations, κ_0 and κ_1, we say that a relation, $R \subseteq \mathcal{T}(\kappa_0) \times \mathcal{T}(\kappa_1)$, is an operational bisimulation iff the following conditions hold:

(1) $[]R[]$

(2) $(\forall \nu_0 \nu_1)(\nu_0 R \nu_1 \Rightarrow (\forall \nu_0' < \nu_0)(\exists \nu_1')(\nu_0' R \nu_1' \wedge \nu_1' \leq \nu_1))$

(3) $(\forall \nu_0 \nu_1)(\nu_0 R \nu_1 \Rightarrow (\forall \nu_1' < \nu_1)(\exists \nu_0')(\nu_0' R \nu_1' \wedge \nu_0' \leq \nu_0))$

(4) $(\forall \nu_0' \nu_1')(\nu_0' R \nu_1' \Rightarrow (\nu_0')_{\mathrm{io}} = (\nu_1')_{\mathrm{io}})$

(5) If $\pi_0 R \pi_1$, then π_0 is *fair* iff π_1 is *fair*.

A few simple consequences of this definition should be made clear. Suppose that

$$\kappa_i = \left\langle\!\!\left\langle\, \alpha_i \;\middle|\; \mu_i \,\right\rangle\!\!\right\rangle_{\chi_i}^{\rho_i} \qquad i < 2$$

are operationally bisimilar. Then $\rho_0 = \rho_1$. The same can not quite be said for χ_0 and χ_1. Since there may be members of these sets that never ever receive any messages. Modulo this sort of garbage, these two sets must be the same.

Lemma (bisimilar composition): Suppose that κ_i are are operationally bisimilar via R, and κ is a composable configuration (i.e its actors are disjoint from those in κ_i). Then there is a relation $R^J \subseteq \mathcal{M}(\mathcal{T}(\kappa_0), \mathcal{T}(\kappa)) \times \mathcal{M}(\mathcal{T}(\kappa_1), \mathcal{T}(\kappa))$ which makes $\mathcal{M}(\mathcal{T}(\kappa_0), \mathcal{T}(\kappa))$ operationally bisimilar to $\mathcal{M}(\mathcal{T}(\kappa_1), \mathcal{T}(\kappa))$.

The following is the most important consequence of bisimulation.

Theorem (opeq): If two non-expansive computation trees, $\mathcal{T}(\kappa_0)$ and $\mathcal{T}(\kappa_1)$, are operationally bisimilar, then $\kappa_0 \cong_s \kappa_1$.

Proof (opeq): Suppose that $\kappa_i = \left\langle\!\!\left\langle\, \alpha_i \mid \mu_i \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$, $i < 2$ are operationally bisimilar via R and that κ is a closing configuration. Then by the composition theorem we have that $\mathcal{T}(\kappa_i \cup \kappa) = \mathcal{M}(\mathcal{T}(\kappa_i), \mathcal{T}(\kappa))$, and by the bisimilar composition theorem we have that $\mathcal{T}(\kappa_0 \cup \kappa)$ is operationally bisimilar to $\mathcal{T}(\kappa_1 \cup \kappa)$ via R^J. This is easily seen to imply that $Obs(\kappa_0 \cup \kappa) = Obs(\kappa_1 \cup \kappa)$ $\square_{\mathbf{opeq}}$

4.1. Example: Removal of Message Indirection

To illustrate how bisimulations can be used to prove two open configurations equivalent, we apply a transformation that removes indirection in message transmission. We begin with system 0, a two actor system with one receptionist r and one reference to an external actor x. The receptionist r takes requests for transforming data, applies some operation, f, and sends the result to the other internal actor a. This actor applies a second operation g and sends the result back to the receptionist, who passes it on, unchanged, to the external actor. This system is transformed into system 1, in which the second actor returns its results directly to the external actor. Equivalence is proved by establishing an operational bisimulation between the two systems, in which related nodes of the computation trees are constrained to be 'in step'. This approach also works for transformations such as fusion or splitting of internal actors [12]. We outline the construction of the bisimulation informally. We call the original system S_0 and the transformed system S_1. We let K_0, K_1 be the initial configurations of the S_0, S_1, respectively. The actors r and a each have only two "interesting" states:

- B_r – the initial behavior of r,
- $B_r[m]$ – the state in which r is processing message m,
- B_a^i – the initial behavior of a, in S_i,
- $B_a^i[m]$ – the state in which a is processing message m, in S_i.

The difference between the behaviors of a in the two systems is the address for sending replies (r in S_0 and x in S_1). When r receives a request or reply message m it sends a message and returns to its initial state. The message sent is $<a \Leftarrow f(v)>$ if m is a request with data v, and $<x \Leftarrow m>$ if m is a reply. When a receives a request message m with data v it sends a reply message with data $g(v)$ and returns to its initial state. In S_0 the reply is sent to r and in S_1 the reply is sent to x.

Now we describe when a S_0 configuration is in step with aS_1 configuration. To do this we define a redirection map X on messages. If m is a reply message with receiver r then $X(m)$ is a reply message with the same data as m and receiver x. Configurations κ_j reachable from K_j are said to be in step if a is in the same state in both configurations and one of the following four cases holds.

(1) the state of r is the same in both configurations, and the pending messages of κ_1 are the image under X of the pending messages of κ_0.

(2) r is processing a reply message m in κ_0 and waiting in κ_1, and the pending messages of κ_1 are the image under X of the pending messages of κ_0 plus a reply m to x.

(3) r is waiting in κ_0 and is processing a request message m in κ_1, and the pending messages of κ_1 are the image under X of the result of removing m from the pending messages of κ_0.

(4) r is processing a reply message m_0 in κ_0 and is processing a request message m_1 in κ_1, and the pending messages of κ_1 are the result of adding a reply m_0 to x to the image under X of the result of removing m_1 from the pending messages of κ_0.

The bisimulation $\mathcal{R}$ is now essentially determined by the clauses (1–4) of the definition and the requirement that $\mathcal{R}$ related configurations must be in step.

5. Discussion

This extended abstract outlines a first step towards a general theory for specification, interconnection, and transformation of components of actor systems. The full paper fills in details including a full set of transition rules and proofs of theorems. Our next task is to develop a logic for specifying components of actor systems, methods for verifying that programs implementing components meet their specifications, and methods for refining specifications into implementations. In addition, we plan to develop methods for modularizing specifications and combining components to build complex systems from simpler systems.

We contrast our work with three related efforts: CSP / Occam [6], the π-calculus [8], and Concurrent ML [3]. The CSP / Occam model is very restrictive. Occam assumes a fixed interconnection topology of processes, supports only static storage allocation, and disallows recursive procedures.

Many of the aims Milner and others had in developing the π-calculus are the same as ours, namely to formulate a language for concurrent computation that allows treatment of data channels as first-class objects, and furthermore for which an algebraic theory may be developed. Equally important, however, are how our aims differ. We aim for a model that can be regarded as a realistic model of a real language, not just an abstract calculus.

We believe realistic models must incorporate fairness assumptions, otherwise the model is impoverished by a collection of starving processes that have been enabled and in any realistic implementation would not be starving. Realistic models also must account for the inherently open nature of distributed systems, but the π-calculus only partially accounts for this: it is impossible to extrude a local port name to an external process. In addition, both CCS and the π-calculus treat senders and receivers uniformly, meaning there can be multiple receivers. However, this means a local receiving process can be corrupted by an external process that also receives on the same port. Lack of locality in these model may cause problems similar to those encountered in denotational models of reference/block structure in higher order languages.

We also contrast our work with more practical development of π-calculus-style communications primitives found in Berry, Milner, and Turner's effort [3], and Reppy's CML [10]. For our purposes we equate these presentations. Even though these theories are more realistic because the languages include other constructs such as functions and atomic data, they still do not incorporate fairness, they still have no theory of open systems, and they still suffer from the drawbacks of a unified treatment of senders and receivers alluded to above. Furthermore, neither of these presentations makes any attempt at developing an equational theory and reasoning principles as we do here.

Aknowledgements

This research was partially supported by DARPA contracts N00039-84-C-0211 and NAG2-703, and NSF grants CCR-8917606, CCR-8915663, and CCR-91-090070 by DARPA and NSF joint contract CCR 90-07195, by ONR contract N00014-90-J-1899, and by the Digital Equipment Corporation.

6. References

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

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

[3] D. Berry, R. Milner, and D.N. Turner. A semantics for ML concurrency primitives. In *Conference record of the 19th annual ACM symposium on principles of programming languages*, pages 105–129, 1992.

[4] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[5] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3), 1977.

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

[7] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[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] G. Plotkin. Call-by-name, call-by-value and the lambda-v-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[10] J. H. Reppy. An operational semantics of first-class synchronous operations. Technical Report TR 91-1232, Cornell University, 1991.

[11] C. L. Talcott. A theory for program and data specification. In *Design and Implementation of Symbolic Computation Systems, DISCO'90*, volume 429 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. full version to appear in TCS special issue.

[12] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT Press, Cambridge Mass., 1990.