Specification Diagrams for Actor Systems

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

May 15, 1998

1 Introduction

We propose here a new form of graphical notation for specifying open distributed object systems. The primary design goal is to make a form of notation for defining message-passing behavior that is expressive, intuitively understandable, and that has a formal underlying semantics.

Specification diagrams are graphical structures. Many specification languages that have achieved widespread usage have a graphical foundation: engineers can understand and communicate more effectively by graphical means. Popular graphical specification languages include Universal Modelling Language (UML) and its predecessors [RJB98], and StateCharts [Har87]. UML is the now-standard set of object-oriented design notations; it includes several different forms of graphical specification notation. Our aim here is a language with similar intuitive advantage but significantly greater expressivity and formal underpinnings. The language is also designed to be useful throughout the development lifecycle, from an initial sketch of the overall architecture to detailed specifications of final components that may serve as documentation of critical aspects of their behavior. Its design was inspired by concepts from actor event diagrams [Cli81], process algebra [Mil80, Hoa85], and UML Sequence Diagrams [RJB98]. In this particular presentation the underlying communication assumptions we use are those taken from the actor model: object- and not channel-based naming, and asynchronous fair message passing.

1.1 Actor Concepts

We provide here a very brief overview of actor concepts; see [Agh86, AMST97, Tal97] for more complete descriptions.

Actors are distributed, object-based message passing entities. Since actors are object-based, they each have a unique *name*, and actors may dynamically create other actors. Individual actors independently compute in parallel, and actors only communicate by message passing. Messages are sent asynchronously, so the sender may continue computing immediately after a message send. Messages are of the form $a \triangleleft M$, indicating message M is sent to actor a. Officially, we have

Definition 1.1 ($\mathbf{A}, \mathbf{M}, \mathbf{MP}$): Define $a \in \mathbf{A}$ to be a fixed countably infinite set of actor names; $M \in \mathbf{M}$ to be a set of message expressions; and,

$$mp \in \mathbf{MP} = \mathbf{A} \triangleleft \mathbf{M} \cup \mathbf{A} \triangleleft \mathbf{M} : \kappa$$

to be the set of message packets for $\kappa \in \mathbf{Key}$ a countable set of keys (e.g. $\mathbf{Key} = \mathbf{Nat}$). We will write $\mathbf{A} \triangleleft \mathbf{M}\{ : \kappa\}$ for a message packet that may or may not have a key. The keys serve a technical purpose which is described below.

All messages must eventually arrive at their destination, but with arbitrary delay. At this point they may be queued if the destination actor is busy. Additionally, individual actor computations must never starve. These two guarantees of progress are the fairness assumptions of actor computation. There is no programming language for actors; one possible language is defined in [MT97] but others are possible. A fixed semantic framework for actors has been developed [AMST97, Tal97]. We will use this framework as the basis for the developments here.

Actor systems are intended to model open distributed computation. This means that the whole system will not be present, and the framework must assume some external actors are interacting with the local system. Additionally, of the local actors, only some of their names may be known by external entities; these are the receptionists. The external actors are notated as the set χ, and the receptionists the set ρ. These sets may grow over time: the external actors will grow based on names received in messages from the outside, and receptionists may grow if new local names are sent out in messages.

Actor systems may be modeled by the set of possible sequences of inputs and outputs they may perform over time. We call one such sequence, possibly infinite, an *interaction path*, and model actor system behavior by a set of such paths. The technical details of this model are summarized in section 3.

2 Specification Diagram Notation

We begin with the graphical notation and an informal idea of its meaning. Figure 1 presents the basic diagram components. Vertical lines indicate progress in time going down, expressing abstract causal ordering on events, with events above

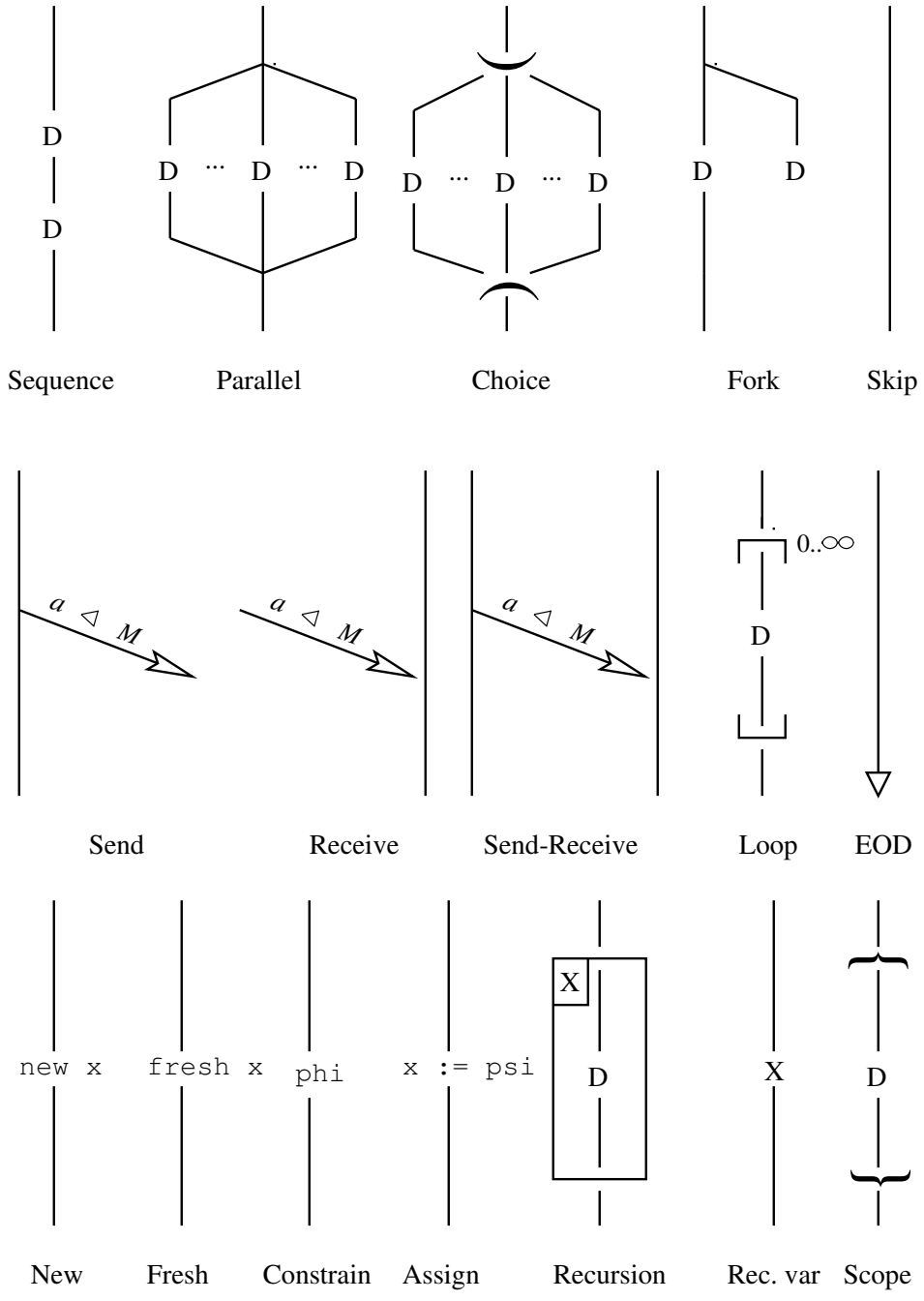

Figure 1: Specification Diagram Components

necessarily leading to events below. This causal ordering will be termed a *causal thread*. Note there is no necessary connection between these "threads" and actors or processes, they exist only at the semantic level: a single thread of causality may be multiple actors, and a single actor may have multiple threads of causality. The components listed in Figure 1 are composed to form specification diagrams. Each form of diagram component has a single in-edge and out-edge—the figure serves as a grammar for the diagrams.

sequencing Vertical lines (causal threads) represent necessary temporal sequencing of events.

parallel Events in the parallel diagrams have no causal ordering between them, but are after events above and before events below.

choice One of the possible choices is taken.

fork A diagram is forked off which hereafter will have no direct causal connection to the current thread (however, messages could indirectly impose some causality between the two).

skip Nothing.

send A message is sent.

receive A message is received by actor a, possibly binding pattern variables in the message M, which can be used below in the diagram.

send-receive A message is sent from one component of the diagram to another, producing a causal cross-connection in what could have been causally unrelated segments.

loop The diagram is iterated some number n times, where n is nondeterministically chosen from the interval $0 \ldots \infty$. The case $n = \infty$ means it loops forever.

EOD Denotes the end of a causal thred in the diagram.

scope Brackets demarcate static scoping of diagram variables.

recursion A boxed diagram fragment, may refer to itself by name, X.

new A fresh diagram variable x is allocated, with arbitrary contents.

fresh A fresh diagram variable x is allocated, with contents an actor name not currently in use.

constraint An arbitrary constraint is placed on the current thread of causality, which must be met. There is no direct analogue to this notion of constraint in programming languages: the constraint may be any mathematical expression. And, a constraint failing does not indicate an error, it indicates that such a computation path will not arise. So, it is significantly different from Hoare-style program assertions.

assign A variable is dynamically assigned a new value. The assignment body, ψ, can be any sensible mathematical expression.

Specification diagrams are truly a *specification* language—for internal messages, both source and target are shown, meaning there is a nonlocal constraint on both sender and receiver of message delivery. On this point they differ from existing forms of concurrent language.

There is no requirement that the choice be fair, in the sense that for a particular actor computation the same branch could always be taken. However, there is a requirement that message delivery be fair, in the sense that any message sent must eventually arrive at its destination.

The parallel and fork operators are similar, but parallel threads must eventually merge, while forked threads are asymmetrical in that the forked threads need never merge.

A *top-level* specification diagram includes an interface, notated D^{ρ}_{χ}. Top-level diagrams are modules which may be directly given semantic meaning. We will not always include the phrase "top level" but meaning should be clear from context.

In this brief paper we will use a textual notation and not the diagrammatic notation.

2.1 Textual Notation for Diagrams

We give an equivalent textual form for diagrams. This will also be considered the "official" syntax, and full details will be given. Diagrams may be easily mapped into the textual syntax.

The set of variables $\mathbf{X}_{\mathrm{d}}$ is the set of diagram variables x, y, z, ... used in diagrams. These variables may take on values in some mathematical universe **U** which we keep open-ended; it could be instantiated with a set theory or type theory. We assume the basic collections of the actor theory are contained in **U**: $\mathbf{A}, \mathbf{M}, \mathbf{MP}, \mathit{true}, \mathit{false} \subseteq \mathbf{U}$. To avoid circular mathematical definitions, we assume **U** is fixed before meaning of specification diagrams is assigned. And, we assume $\mathit{acq}(s)$ for $s \in \mathbf{U}$ is defined and constrain it to be a finite subset of **A**.

The notion of message packet **MP** defined earlier included an optional *key*. Keys are necessary to give a textual representation to the send-receive edges in di-

agrams, constraining a send to be connected to its corresponding receive by sharing the same key. The interaction paths will not contain keys, they are used for local synchronization only.

Specification diagram message packets $\mathbf{MP}_\mathrm{d}$ are packets of **MP** parameterized by diagram variables $\mathbf{X}_\mathrm{d}$.

Definition 2.1 ($\mathbf{MP}_\mathrm{d}, \mathbf{M}_\mathrm{d}, \mathbf{A}_\mathrm{d}$): Let $\mathbf{MP}_\mathrm{d}$ be $\mathbf{A}_\mathrm{d} \triangleleft \mathbf{M}_\mathrm{d} \cup \mathbf{A}_\mathrm{d} \triangleleft \mathbf{M}_\mathrm{d} : \kappa$ where $\mathbf{A}_\mathrm{d} = (\mathbf{X}_\mathrm{d} \to \mathbf{U}) \to \mathbf{A}$ and $\mathbf{M}_\mathrm{d} = (\mathbf{X}_\mathrm{d} \to \mathbf{U}) \to \mathbf{M}$.

The parameter $\varepsilon \in \mathbf{X}_\mathrm{d} \to \mathbf{U}$ is an environment interpreting free diagram variables. We use the informal convention that message expression $M_\mathrm{d} = x + 1$ is an abbreviation for $M_\mathrm{d}(\varepsilon) = \varepsilon(x) + 1$.

Specification diagrams may now be defined officially by the following grammar.

Definition 2.2 (Specification Diagram Grammar): The specification diagrams, $D \in \mathbb{D}$, are defined by the following grammar.

name	textual notation
skip	`skip`
sequence	$D_1; D_2$
parallel	$D_1 \parallel D_2$
choice	$D_1 \oplus D_2$
scope	$\{D\}$
fork	`fork`(D)
receive	`receive`(mp_d)
send	`send`(mp_d)
constraint	`constrain`(ϕ)
new variable	`new`(x)
fresh actor name	`fresh`(x)
assignment	x `:=` ψ
recursion	`rec`$X.D$
recursion variable	X

where $D_i \in \mathbb{D}$ are diagrams, and $X \in \mathbf{X}_\mathrm{r}$ is a countable set of recursion variables. A *top level* diagram is a diagram with an interface, D^ρ_χ.

Sequencing is right-associative, and binds most tightly, followed by choice and then parallel composition binding most loosely. We use an implicit static variable binding convention: `new`(x), `fresh`(x), and `receive`$(a_\mathrm{d} \triangleleft M_\mathrm{d}\{ : \kappa\})$, with x informally occurring in M_d, all denote implicit *bindings* (declarations) of variable x. $\{\ldots\}$ denotes a scope boundary, giving the static end of any implicit declarations contained immediately within. The extent of a variable binding is, intuitively, all points which causally must follow the binding, and that are at the same or deeper

scope boundary. Thus for instance in $(\texttt{new}(x) \parallel \texttt{skip}); x := 5$, the assignment to x is within the scope of the `new` since this assignment causally must follow the declaration of x. If $\parallel$ were replaced by $\oplus$ in this example, the assignment x would not be bound (in fact it would be partially bound: if the left choice were taken it would be bound, but not if the right choice were taken).

The assignment expression ψ is informally a function on $\mathbf{U}$ which may refer to variables in $\mathbf{X}_\mathrm{d}$; formally, $\psi \in \mathbf{P}_\omega[\mathbf{A}] \times \mathbf{P}_\omega[\mathbf{A}] \times (\mathbf{X}_\mathrm{d} \to \mathbf{U}) \to \mathbf{U}$: it takes as parameter a tuple $(\rho, \chi, \varepsilon)$ where the environment ε gives mathematical meaning to the free variables, and ψ may refer to actors in ρ and χ. Predicates on $\mathbf{U}$ are notated ϕ, and are formally predicates on $\mathbf{P}_\omega[\mathbf{A}] \times \mathbf{P}_\omega[\mathbf{A}] \times (\mathbf{X}_\mathrm{d} \to \mathbf{U})$. Informally, we will write e.g. "$x \in \rho$" to mean a predicate ϕ where $\phi(\rho, \chi, \varepsilon)$ iff $\varepsilon(x) \in \rho$. $acq(D)$ is defined as $\{acq(s) \mid s \in \mathbf{U} \text{ occurs as an } mp_\mathrm{d}, \phi \text{ or } \psi \text{ in } D\}$.

We use the convention that $\texttt{ExampleMacro}(x, y, z) = D$ defines a macro. Macros are just functions: we will be careful not to define self-referential macros. Certain syntax is easily encodable via macros and so is not defined in the core grammar.

Definition 2.3 (Diagram Macro Library): nondeterministic iteration: $[D]^{0\ldots\infty} = \texttt{rec}X.((D;X) \oplus \texttt{skip})$

interval iteration: $[D]^{i\ldots j} =$

$$\texttt{new}(x); \texttt{constrain}(i \le x \le j);$$
$$\texttt{rec}X.\texttt{constrain}(x = 0) \oplus \texttt{constrain}(x > 0); x := x - 1; D; X$$

for fresh variable x

if-then: $\texttt{if } \phi \texttt{ then } D_1 \texttt{ else } D_2 =$

$$(\texttt{constrain}(\phi); D_1) \oplus (\texttt{constrain}(\neg\phi); D_2)$$

while-do: $\texttt{while } \phi \texttt{ do } D =$

$$\texttt{rec}X.(\texttt{constrain}(\phi); D; X) \oplus \texttt{constrain}(\neg\phi)$$

end of diagram: $\texttt{eod} = \texttt{rec}X.(\texttt{skip}; X)$

abort path: $\texttt{abort} = \texttt{constrain}(\mathit{false})$

initialized new: $\texttt{new}(x = s) = \texttt{new}(x); \texttt{constrain}(x = s)$

constrained new: $\texttt{new}(x \in S) = \texttt{new}(x); \texttt{constrain}(x \in S)$

constrained receipt: $\texttt{receive}(a_\mathrm{d} \triangleleft M_\mathrm{d} \in S) =$

$$\texttt{receive}(a_\mathrm{d} \triangleleft M_\mathrm{d}); \texttt{constrain}(M_\mathrm{d} \in S)$$

Translation from informal diagrams into textual notation is straightforward from the above in all cases except the internal message edges. With these internal edges, a diagram can take on an arbitrary graph structure, and this cannot be placed in purely textual notation. The purpose of keyed message packets is precisely to capture the 1-1 relation implied by an internal edge: the sender and receiver must be paired with no other send/receive to this message. Thus, for each internal edge in a diagram, a fresh key κ is assigned to it, and message $a_d \triangleleft M_d$ on the message edge translated into a $\texttt{send}(a_d \triangleleft M_d : \kappa)$ action by the sender and a $\texttt{receive}(a_d \triangleleft M_d : \kappa)$ action by the receiver.

2.2 Small Examples

We now give a series of small examples to illustrate use of the language. None of the examples attempts to seriously illustrate the usefulness of the language for specifications, as the examples are too small; the goal here is just to give informal clarification of the syntax and semantics.

Message Passing Semantics Message passing in diagrams differs from actor programming. Diagram messages are specifications that a message was sent on one end and actively processed on the other. For instance, consider a sink:

$$\texttt{Sink}(a) = [\texttt{receive}(a \triangleleft x)]^{0\ldots\infty}$$

A sink behavior in an actor is a behavior that does nothing; messages will automatically queue up. At the specification level, we need to specify the receipt of those messages that are forever ignored. The diagrams do implicitly account for the asynchronous nature of actor communication: a message edge only constrains the send to be before the receive, not for the two to happen simultaneously. The use of "$0\ldots\infty$" iteration models all possible environment behaviors. The environment may send 0, an arbitrary number, or infinitely many messages.

The interaction path semantics of the top-level diagram $\texttt{Sink}(a)^{\{a\}}_{\emptyset}$ have paths of the following form. Each path consists of a sequence of input messages $a \triangleleft M$. Since the system is reactive, the environment could send $0, 1, 2, \ldots, n, \ldots$ or even infinitely many such messages. No messages are sent out.

So, a point about message passing in diagrams is that messages which will *never* be processed must be specified as arriving. Consider

$$[\texttt{receive}(a \triangleleft x);\ \texttt{constrain}(x > 100)\ \ldots]^{0\ldots\infty}$$

This specification may at first look analogous to a synchronization constraint that only processes messages with contents larger than 100. However, a synchronization constraint of this form will implicitly forever ignore messages with contents

less than 100. The above specification instead constrains the environment: it requires that such messages will in fact never be sent. Specifications which constrain the environment are *partial*.

Ticker A ticker is a simple actor which increments a counter upon receipt of a `tick` message, and replies to `time` messages with the current time. First, a partial specification of a ticker is given which only specifies that time replies are numerical, not that the number of tick inputs is counted. This shows how specifications may underconstrain the program space.

$$\begin{array}{l}\texttt{PartialTicker}(a) = \\ \quad [\texttt{receive}(a \triangleleft \texttt{tick})]^{0\ldots\infty} \parallel \\ \quad [\texttt{receive}(a \triangleleft \texttt{time@}x);\ \texttt{new}(y \in \mathbf{Nat});\ \texttt{send}(x \triangleleft \texttt{reply}(y))]^{0\ldots\infty}\end{array}$$

The possible interaction paths for the top-level diagram $\texttt{PartialTicker}^{\{a\}}_{\emptyset}$ may be informally described as any path satisfying the following. For each $a \triangleleft \texttt{time@}c$ input in the path, there is later a $c \triangleleft \texttt{reply}(n)$ output for arbitrary $n \in \mathbf{Nat}$, and all outputs are `reply` messages so paired. There also may be any number, including infinitely many, $a \triangleleft \texttt{tick}$ input messages in any order.

Next we give a high-level specification of the full ticker: it gives a sequence of replies to `time` messages in a non-decreasing sequence.

$$\begin{array}{l}\texttt{Ticker}(a) = \\ \quad [\texttt{receive}(a \triangleleft \texttt{tick})]^{0\ldots\infty} \parallel \\ \quad \texttt{new}(\texttt{count} \in \mathbf{Nat}); \\ \quad [\texttt{receive}(a \triangleleft \texttt{time@}x);\ \texttt{new}(y \in \mathbf{Nat}); \\ \quad \texttt{count} := \texttt{count} + y;\ \texttt{send}(x \triangleleft \texttt{reply}(\texttt{count}))]^{0\ldots\infty}\end{array}$$

The actual implementation of a ticker we are specifying sends `tick` messages to itself to increment the counter: every time it receives a `tick`, it increments the counter and sends itself another `tick`. A low-level specification which is more close to an actual implementation is

$$\begin{array}{l}\texttt{IntensionalTicker}(a) = \\ \quad \texttt{send}(a \triangleleft \texttt{tick});\ \texttt{new}(\texttt{count} = 0); \\ \quad [(\texttt{receive}(a \triangleleft \texttt{tick}); \\ \quad \texttt{count} := \texttt{count} + 1;\ \texttt{send}(a \triangleleft \texttt{tick})) \\ \quad \oplus \quad (\texttt{receive}(a \triangleleft \texttt{time@}x);\ \texttt{send}(x \triangleleft \texttt{reply}(\texttt{count})))]^{0\ldots\infty}\end{array}$$

Note that `tick` messages could in theory be sent by the environment. We then desire to establish equivalences on top-level diagrams such as

$$\texttt{Ticker}(a)_{\emptyset}^{\{a\}} \cong \texttt{IntensionalTicker}(a)_{\emptyset}^{\{a\}}$$

to show a high-level specification is equivalent to a low-level one. Extensional equivalence $\cong$ is defined in Section 5.

The above diagrams have a restricted acceptable message input set: messages must be in $\{\texttt{tick},\texttt{time}\}$, and so the diagrams are partial. It is not difficult to extend a diagram to a diagram without acceptable message restrictions:

$$\begin{array}{l}\texttt{CompleteTicker}(a) = \\ \quad \texttt{fork}(\,[\texttt{receive}(a \triangleleft x);\ \texttt{constrain}(\mathit{msgMeth}(x) \notin \{\texttt{tick},\texttt{time}\})]^{0\ldots\infty});\\ \quad \texttt{Ticker}(a)\end{array}$$

Ticker Factory This example shows how a fresh actor may be dynamically generated.

$$\begin{array}{l}\texttt{TickerFactory}(a) = \\ \quad [\texttt{receive}(a \triangleleft \texttt{new}@c);\\ \quad \texttt{fresh}(x);\\ \quad \texttt{fork}(\texttt{Ticker}(x));\\ \quad \texttt{send}(c \triangleleft \texttt{reply}(x))]^{0\ldots\infty}\end{array}$$

Function Composer We assume as given two mathematical functions F and G which are defined over all possible messages. First we specify an abstract function-computer behavior.

$$\begin{array}{l}\texttt{Function}(a, f) = \\ \quad [\texttt{fork}(\texttt{receive}(a \triangleleft \texttt{compute}(x)@c);\ \texttt{send}(c \triangleleft \texttt{reply}(f(x))))]^{0\ldots\infty}\end{array}$$

If the `fork` above was not present, the specification would implicitly order the function calls. However, since the function call order is irrelevant, the forkless specification would show causal ordering that was not necessary. The forking and forkless versions are in fact equivalent, however, in the sense that they specify the same set of interaction paths.

An actor which computes $G \circ F$ is then specified as $\texttt{Function}(a, G \circ F)$. A hypothetical implementation may use two distinct actors $\texttt{Function}(a_F, F)$ and $\texttt{Function}(a_G, G)$ to compute F and G respectively, and a `Composer` actor used to put together the composition. The system with these three actors should meet the above specification. The `Composer` is specified as follows.

$$\begin{array}{l}\texttt{Composer}(a,f,g,a_f,a_g) = \\ \quad [\texttt{receive}(a \triangleleft \texttt{compute}(x)@c); \\ \quad \texttt{fresh}(x_f);\ \texttt{send}(a_f \triangleleft \texttt{compute}(x)@x_f);\ \texttt{receive}(x_f \triangleleft \texttt{reply}(x)); \\ \quad \texttt{fresh}(x_g);\ \texttt{send}(a_g \triangleleft \texttt{compute}(x)@x_g);\ \texttt{receive}(x_g \triangleleft \texttt{reply}(x)); \\ \quad \texttt{send}(c \triangleleft \texttt{reply}(x))]^{0\ldots\infty}\end{array}$$

The complete low-level specification is then the parallel composition of these three:

$$\begin{array}{l}\texttt{FunctionComposer}(a,F,G,a_F,a_G) = \\ \quad \texttt{Composer}(a,F,G,a_F,a_G) \parallel \texttt{Function}(a_F,F) \parallel \texttt{Function}(a_G,G)\end{array}$$

In an open distrubuted system, components may be designed as separate top-level diagrams and then composed. In particular for large systems for which the design is distributed across different political entities, object-based components are composed. For this example, the top-level composition is defined as follows.

$$\begin{array}{l}\texttt{FunctionComposerModule}(a,F,G,a_F,a_G)^{\{a\}}_{\emptyset} = \\ \quad \texttt{Composer}(a,F,G,a_F,a_G)^{\{a\}}_{\{a_F,a_G\}} \parallel \texttt{Function}(a_F,F)^{\{a_F\}}_{\emptyset} \parallel \texttt{Function}(a_G,G)^{\{a_G\}}_{\emptyset}\end{array}$$

where $\parallel$ is not the composition operator on diagrams, but the composition operator on top-level specifications. Theorem 5.3 below may be used to show the two methods for composition produce the same specification:

$$\texttt{FunctionComposer}(a,F,G,a_F,a_G)^{\{a\}}_{\emptyset} \cong \texttt{FunctionComposerModule}(a,G \circ F)^{\{a\}}_{\emptyset}$$

This specification is equivalent to the abstract $\texttt{Function}(a, G \circ F)$:

$$\texttt{FunctionComposer}(a,F,G,a_F,a_G)^{\{a\}}_{\emptyset} \cong \texttt{Function}(a,G \circ F)^{\{a\}}_{\emptyset}$$

We sketch how an argument to establish this equivalence would proceed. The first step is to perform a "zipping" transformation on `FunctionComposer` to connect send and receive to give send-receive cross-edges. This is a point where the diagrammatic semantics becomes significantly more readable than the textual form.

$$\begin{array}{l}\texttt{ZippedFunctionComposer}(a,f,g,a_f,a_g) = \\ \quad [\texttt{receive}(a \triangleleft \texttt{compute}(x)@c);\ \texttt{fresh}(x_f);\ \texttt{fresh}(x_g); \\ \quad \texttt{send}(a_f \triangleleft \texttt{compute}(x)@x_f : \kappa_f);\ \texttt{receive}(x_f \triangleleft \texttt{reply}(f(x)) : \kappa'_f); \\ \quad \texttt{send}(a_g \triangleleft \texttt{compute}(x)@x_g : \kappa_g);\ \texttt{receive}(x_g \triangleleft \texttt{reply}(g(x)) : \kappa'_g); \\ \quad \texttt{send}(c \triangleleft \texttt{reply}(x))]^{0\ldots\infty} \parallel \\ \quad [\texttt{fork}(\texttt{receive}(a_f \triangleleft \texttt{compute}(x)@x_f : \kappa_f);\ \texttt{send}(x_f \triangleleft \texttt{reply}(f(x) : \kappa'_f)))]^{0\ldots\infty} \parallel \\ \quad [\texttt{fork}(\texttt{receive}(a_g \triangleleft \texttt{compute}(x)@x_g : \kappa_g);\ \texttt{send}(x_g \triangleleft \texttt{reply}(g(x) : \kappa'_g)))]^{0\ldots\infty}\end{array}$$

The ZippedFunctionComposer is an intermediate stage in the incremental translation of FunctionComposer to Function. By proving

$$\texttt{FunctionComposer}(a,F,G,a_F,a_G)^{\{a\}}_{\emptyset} \cong$$
$$\texttt{ZippedFunctionComposer}(a,F,G,a_F,a_G)^{\{a\}}_{\emptyset} \cong$$
$$\texttt{Function}(a,G\circ F)^{\{a\}}_{\emptyset}$$

the desired equivalence is established.

Simple Memory Cell

$\texttt{Cell}(a) =$
  $\texttt{new}(x)$;(* cell value, initially arbitrary *)
  [($\texttt{receive}(a \triangleleft \texttt{set}(v)@c)$;(* c/v are pattern variables *)
  $x := v$;
  $\texttt{send}(c \triangleleft \texttt{ack})$)
  $\oplus$
  ($\texttt{receive}(a \triangleleft \texttt{get}@c)$;
  $\texttt{send}(c \triangleleft \texttt{reply}(x))$]$^{0..\infty}$

A top-level specification for a memory cell is then $\texttt{Cell}(a)^{\{a\}}_{\emptyset}$. Another powerful idea is to write a partial specification which only responds to message input patterns that are semantically sensible. For instance, for a cell, receipt of a get before any set messages have arrived is unreasonable; we thus make a specification which constrains the environment to never do this.

$\texttt{EnvironmentConstrainingCell}(a) =$
  $\texttt{new}(x = \texttt{undefined})$;(* cell initially undefined *)
  [($\texttt{receive}(a \triangleleft \texttt{set}(v)@c)$;
  $x := v$;
  $\texttt{send}(c \triangleleft \texttt{ack})$)
  $\oplus$
  ($\texttt{receive}(a \triangleleft \texttt{get}@c)$;$\texttt{constrain}(x \neq \texttt{undefined})$
  $\texttt{send}(c \triangleleft \texttt{reply}(x))$]$^{0..\infty}$

Environment constraints are a powerful aspect of specification diagrams. Such specifications are partial: they are not defined for all patterns of input. Composition with such specifications is also partial, as it may fail since the specification is not fully reactive. However, the concept of satisfaction of an environment-constrained specification is a difficult one and so now we will generally study complete specifications only.

3 A Path-Based Framework for Actors

In this section we briefly review the semantic framework used to model actor systems; this framework will then be used to model specification diagrams. See [Tal97] for details; here we provide a terse and simplified presentation. We use a path-based (trace-based) semantics: an open, nondeterministic system is interpreted as a set of *interaction paths*. Each path is a possibly infinite list of input and output actions. Interaction path semantics models an actor system in terms of the possible interactions (patterns of message passing) it can have with its environment. Interaction semantics does not let us use any information about internal computations or what actors may be initially present or known beyond those specificed in the interface. A specification diagram is given meaning as a set of interaction paths, so the meaning of a diagram D may be the same as the meaning of some actor program implementation. If this is in fact the case, we can assert that the implementation meets the specification D.

Interfaces An actor system interface is a pair $\begin{smallmatrix}\rho\\ \chi\end{smallmatrix}$ of disjoint finite sets of actor names. ρ specifies the receptionists and χ specifies the external actors known to the system.

We define parallel composition of interfaces by

$$\begin{smallmatrix}\rho_1\\ \chi_1\end{smallmatrix} \bowtie \begin{smallmatrix}\rho_2\\ \chi_2\end{smallmatrix} \quad \text{iff} \quad \rho_1 \cap \rho_2 = \emptyset$$

$$\begin{smallmatrix}\rho_1\\ \chi_1\end{smallmatrix} \parallel \begin{smallmatrix}\rho_2\\ \chi_2\end{smallmatrix} = \begin{smallmatrix}\rho_1 \cup \rho_2\\ (\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2)\end{smallmatrix}, \text{ provided } \begin{smallmatrix}\rho_1\\ \chi_1\end{smallmatrix} \bowtie \begin{smallmatrix}\rho_2\\ \chi_2\end{smallmatrix}$$

Events In order to distinguish differrent occurrences of message packets with the same contents we use a set, $\mathbf{E}$, of events. Each event, e, contains a message packet, $pkt(e) \in \mathbf{MP}$. We let e range over $\mathbf{E}$ and E range over $\mathcal{P}[\mathbf{E}]$. We write $\mathtt{in}(e)$ to distinguish the arrival of a message from outside the system from its delivery e. We also write $\mathtt{out}(e)$ to distinguish delivery of a message to the environment from from the actual delivery to the target actor. We let $\widetilde{\mathbf{E}}$ be the set of events extended by these input/output events

$$\widetilde{\mathbf{E}} = \mathbf{E} \cup \mathtt{in}(\mathbf{E}) \cup \mathtt{out}(\mathbf{E})$$

and we let $\tilde{e}$ range over $\widetilde{\mathbf{E}}$ and $\widetilde{E}$ range over $\mathcal{P}[\widetilde{\mathbf{E}}]$. We will use a simple theory of potentially infinite lists, notated $[x_1, x_2, \ldots, x_k, \ldots]$. List concatenation is written $[\ldots] * [\ldots]$, and for the case where the first list is infinite returns that as the result. Unit for concatenation is the empty list, $[\,]$. $S\,\mathbf{List}$ is the set of lists with elements from S.

Interaction Paths An interaction path is a possibly infinite list of input/output events,

$$\pi = [\tilde{e}_1, \tilde{e}_2, \ldots, \tilde{e}_k, \ldots] \in (\mathtt{in}(\mathbf{E}) \cup \mathtt{out}(\mathbf{E}))\,\mathbf{List}.$$

It represents a potentially infinite sequence of interactions of a system with its environment as observed by some hypothetical observer. π^{ρ}_{χ} represents an interfaced interaction path, a path with the indicated interface. All of the interaction paths constructed in this paper are constrained to obey the *EPLaw* of [Tal97]. *EPLaw*(π) requires inputs of π to be to receptionists or names sent in a previous output, and outputs to be to external actors or actors whose name was received in a previous input. *EPLaw* corresponds to the Baker-Hewitt locality laws governing how actors become acquainted with one another.

3.1 Interaction Path Sets and their Algebra

An interaction path models one possible way a system might interact with its environment. We model the behavior of a system by sets of interfaced interaction paths, *Ip*.

Parallel Composition We define composability and composition on interaction path sets. The basic operation for composing paths is dovetailing two interaction paths, $\pi_0 \;\mathsf{Z}\; \pi_1$. This operation is defined in terms of precursor $\pi_0 \;\mathsf{Z}^0\; \pi_1$, which is the greatest symmetric function closed under the following (in the following we abuse notation and write $[\ldots] * S$ to mean $\{[\ldots] * s \;\big|\; s \in S\}$).

$$\begin{array}{ll}
(0) & [\,] \;\mathsf{Z}^0\; \pi = \{\pi\} \\
(1) & [\mathtt{in}(e)] * \pi_0 \;\mathsf{Z}^0\; [\mathtt{out}(e)] * \pi_1 = \pi_0 \;\mathsf{Z}^0\; \pi_1 \\
(2) & [\tilde{e}_0] * \pi_0 \;\mathsf{Z}^0\; [\tilde{e}_1] * \pi_1 = \\
 & \quad \{[\tilde{e}_0] * (\pi_0 \;\mathsf{Z}^0\; [\tilde{e}_1] * \pi_1)\} \cup \\
 & \quad \{[\tilde{e}_1] * ([\tilde{e}_0] * \pi_0 \;\mathsf{Z}^0\; \pi_1)\}
\end{array}$$

Then, $\pi_0 \;\mathsf{Z}\; \pi_1$ is defined as $\pi_0 \;\mathsf{Z}^0\; \pi_1$ with the paths $\pi \in \pi_0 \;\mathsf{Z}^0\; \pi_1$ that after some point forever starve events from one of π_0 or π_1 removed.

Define composability and parallel composition for path sets Ip_0 and Ip_1 with interfaces $\overset{\rho_0}{\chi_0}$ and $\overset{\rho_1}{\chi_1}$ as

$$\begin{array}{l}
Ip_0 \bowtie Ip_1 \text{ iff } \overset{\rho_0}{\chi_0} \bowtie \overset{\rho_1}{\chi_1} \\
Ip_0 \parallel Ip_1 = \{\pi^{\rho}_{\chi} \;\big|\; (\exists\, \pi_0 \overset{\rho_0}{\chi_0} \in Ip_0, \pi_1 \overset{\rho_1}{\chi_1} \in Ip_1) \\
\qquad\qquad (\pi \in \pi_0 \;\mathsf{Z}\; \pi_1,\ \pi_0 \text{ and } \pi_1 \text{ share no events } \tilde{e}, \text{ and } \mathit{EPLaw}(\pi))\} \\
\quad \text{where } \overset{\rho}{\chi} = \overset{\rho_0}{\chi_0} \parallel \overset{\rho_1}{\chi_1}, \text{ provided } \overset{\rho_0}{\chi_0} \bowtie \overset{\rho_1}{\chi_1}.
\end{array}$$

Restriction The restriction of Ip with interface π^{ρ}_{χ} to ρ' is defined by

$$Ip\lceil\rho' = \{\pi^{\rho'}_{\chi} \mid \pi^{\rho}_{\chi} \in Ip \text{ and } \pi \text{ contains no } \mathtt{in}(a \triangleleft M) \text{ events for } a \in (\rho - \rho')\}$$

Renaming Renaming of interaction paths, $\widehat{\alpha}(\pi)$, is pointwise on each event in the path, and renaming on path sets $\widehat{\alpha}(Ip)$ is pointwise on each element of the set.

4 Operational Semantics of Diagrams

Diagrams are given meaning in this section via an operational semantics. The goal is to give a set of interaction paths defining the behavior of top-level diagrams D^{ρ}_{χ}. This is accomplished by defining a single-step relation mapping configurations to configurations, the transitive closure of which yields a set of computation paths. In this sense it is a standard presentation of operational semantics of actors [AMST97, Tal97]. The main difference with these previous works is more post-processing is required to remove paths that are not admissible. A configuration is of the form

$$D^{\rho}_{\chi}\,\mu$$

Where ρ, χ are the current receptionists and external actor names, and $\mu \subseteq \mathbf{E}$ is a set of messages in transit, either to be sent out of the system or to be received locally. Since each message is a unique event, it is possible to have two messages with identical target and contents in μ.

4.1 Preliminaries

Before presenting the operational semantics, we need to define two concepts. We use a small-step semantics based on factoring a diagram D expression into a redex D_{rdx} and reduction (a.k.a. evaluation) context R, $D = R[D_{\text{rdx}}]$. Notation is also needed for looking up, modifying, and extending variable bindings. The concepts of reduction context and environment are in fact intertwined: environments are local to particular points in reduction (so e.g. parallel threads may have differing environments) and so are spread around the reduction context. These local environments are functions $\gamma_i \in \mathbf{X}_{\text{d}} \to \mathbf{U}$ which hold the current state of diagram variables $\mathbf{X}_{\text{d}}$, and map only finitely many variables to non-$\perp$ values, $\perp \in \mathbf{U}$ being a special meta-value indicating undefined. A small addition to the language syntax is used to bind variables inside the executing diagram: $\{\!|\gamma\colon D|\!\}$ indicates a lexical scoping construct $\{D\}$ under which execution is actively occurring, with current local environment γ.

Reduction contexts R (a.k.a. evaluation contexts) are used to isolate the next redex to be reduced. Their grammar is

$$R = \bullet \text{ or } R \parallel D \text{ or } D \parallel R \text{ or } R;D \text{ or } \{\!|\gamma : R|\!\}$$

$R[D]$ denotes the act of replacing the (unique) $\bullet$ in R with diagram D. We will need notation for reduction contexts defined as above but without the $\{\!|\gamma : R|\!\}$ case; they will be notated R^-.

Notation is next defined for manipulation of the environment. The basic operations needed include $R@x$ to look up the value of x in the environments of R, $R@x := s$ to modify the value of already-declared variable x, and $R \oplus x := s$ to add a new definition of x in the innermost lexical level. $\varepsilon(R)$ extracts the environment from R in the form of a function from diagram variables to values.

Definition 4.1 ($R@x, R@x := s, R \oplus x := s$, $\varepsilon(R)$): Letting

$$R = R_0^-[\{\!|\gamma_1 : \ldots R_n^-[\{\!|\gamma_n : R_{n+1}^-|\!\}] \ldots |\!\}],$$

define

lookup $R@x$ is $\gamma_i(x)$ where i is the largest number less than or equal to n with $\gamma_i(x) \neq \perp$, or $\perp$ if all $\gamma_i(x) = \perp$.

modify $R@x := s$ is $R_0^-[\{\!|\gamma_1 : \ldots R_i^-[\{\!|\gamma_i' : \ldots R_n^-[\{\!|\gamma_n : R_{n+1}^-|\!\}] \ldots |\!\}] \ldots |\!\}]$ where i is the largest number less than or equal to n with $\gamma_i(x) \neq \perp$, and $\gamma_i' = \gamma_i$ at all points except $\gamma_i'(x) = s$. If no such i exists, R is unchanged.

extend $R \oplus x := s$ is $R_0^-[\{\!|\gamma_1 : \ldots R_n^-[\{\!|\gamma_n' : R_{n+1}^-|\!\}] \ldots |\!\}]$ for $\gamma_n' = \gamma_n$ at all points except $\gamma_n'(x) = s$.

extract $\varepsilon(R) = f$ where $f(x) = R@x$. $\mathrm{Dom}(\varepsilon(R)) = \{x \mid (\varepsilon(R))(x) \neq \perp\}$.

acquaintances $acq(D)$ is

$$\bigcup_{M_\mathrm{d} \in D} acq(M_\mathrm{d}) \cup \bigcup_{\phi \in D} acq(\phi) \cup \bigcup_{\psi \in D} acq(\psi) \cup \bigcup_{a_\mathrm{d} \in D} acq(a_\mathrm{d}) \cup \bigcup_{\gamma \in D} \bigcup_{x \in \mathbf{X}_\mathrm{d}} acq(\gamma(x))$$

where "$\cdot \in D$" here means occurrence as a subterm in D. $acq(R)$ is defined identically to $acq(D)$ except with the last clause being $\bigcup_{1 \leq i \leq n} \bigcup_{x \in \mathbf{X}_\mathrm{d}} acq(\gamma_n(x))$.

4.2 The Semantic Definition

In this section we define the semantic meaning function $[\![D_\chi^\rho]\!]$ mapping top-level diagrams to sets of interaction paths that describe the input-output behavior of the diagram.

$D^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{in}(e)} D^{\rho}_{\chi'}\,(\mu \cup \{e\})$

where $e \notin \mu$, $\chi' = \chi \cup (acq(e) - \rho)$, $pkt(e)$ is not keyed, $target(e) \in \rho$, and $(acq(D) \cup acq(\mu)) \cap acq(e) \subseteq \rho \cup \chi$

$D^{\rho}_{\chi}\,(\mu \cup \{e\}) \xrightarrow{\texttt{out}(e)} D^{\rho'}_{\chi}\,\mu$

where $e \notin \mu$, $\rho' = \rho \cup (acq(e)) - \chi)$, $pkt(e)$ is not keyed, and $target(e) \in \chi$

$R[\texttt{skip}; D]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{seq}} R[D]^{\rho}_{\chi}\,\mu$

$R[\texttt{skip} \parallel \texttt{skip}]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{par}} R[\texttt{skip}]^{\rho}_{\chi}\,\mu$

$R[D_l \oplus D_r]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{choose}(l)} R[D_l]^{\rho}_{\chi}\,\mu$

and similarly for $\texttt{choose}(r)$

$R[\texttt{fork}(D)]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{fork}} (R[\texttt{skip}] \parallel R'[D])^{\rho}_{\chi}\,\mu$

where for $R = R_0^-[\{\!|\gamma_1 : \ldots R_n^-[\{\!|\gamma_n : R_{n+1}^-|\!\}]\ldots|\!\}]$, $R' = \{\!|\gamma_1 : \ldots \{\!|\gamma_n : \bullet|\!\}\ldots|\!\}$

$R[\texttt{receive}(a_{\mathrm{d}} \triangleleft M_{\mathrm{d}}\{\,:\kappa\})]^{\rho}_{\chi}\,(\mu \cup \{e\}) \xrightarrow{\texttt{receive}(e)} R'[\texttt{skip}]^{\rho}_{\chi}\,\mu$

where $R' = R \oplus \overline{x := s}$, and $pkt(e) = a_{\mathrm{d}}(\varepsilon(R)) \triangleleft M_{\mathrm{d}}(\varepsilon(R'))\{\,:\kappa\}$

$R[\texttt{send}(a_{\mathrm{d}} \triangleleft M_{\mathrm{d}}\{\,:\kappa\})]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{send}(e)} R[\texttt{skip}]^{\rho}_{\chi}\,(\mu \cup \{e\})$

where $pkt(e) = a_{\mathrm{d}}(\varepsilon(R)) \triangleleft M_{\mathrm{d}}(\varepsilon(R))\{\,:\kappa\}$ and e is a fresh event

$R[\texttt{new}(x)]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{new}(s)} (R \oplus x := s)[\texttt{skip}]^{\rho}_{\chi}\,\mu$

where $acq(s) \subseteq \rho \cup \chi \cup acq(R) \cup acq(\mu)$

$R[\texttt{fresh}(x)]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{fresh}(a)} (R \oplus x := s)[\texttt{skip}]^{\rho}_{\chi}\,\mu$

where $a \notin \rho \cup \chi \cup acq(R[\texttt{skip}]) \cup acq(\mu)$

$R[\texttt{constrain}(\phi)]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{constrain}} R[\texttt{skip}]^{\rho}_{\chi}\,\mu$

where $\phi(\rho, \chi, \varepsilon(R))$ holds

$R[x := \psi]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{assign}} (R'[\texttt{skip}]^{\rho}_{\chi}\,\mu$

where $R' = R@(x := \psi(\rho, \chi, \varepsilon(R)))$

$R[\texttt{rec}\,X.D]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{recursion}} R[D[(\texttt{rec}\,X.D)/X]]^{\rho}_{\chi}\,\mu$

$R[\{D\}]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{scope-in}} R[\{\!|(\lambda x.\bot) : D|\!\}]^{\rho}_{\chi}\,\mu$

$R[\{\!|\gamma : \texttt{skip}|\!\}]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{scope-out}} R[\texttt{skip}]^{\rho}_{\chi}\,\mu$

Figure 2: Single-Step Computation for Diagrams

Definition 4.2: The single-step computation relation on diagram configurations is defined in Figure 2.

For each of the rules except `in`/`out`, the *redex* is the D_{rdx} for left-hand-side $R[D_{\mathrm{rdx}}]$.

Definition 4.3: Given a top-level diagram $D_{0\chi_0}^{\rho_0}$, define

raw event paths $[\![D_{0\chi_0}^{\rho_0}]\!]_{\mathrm{raw}} =$

$$\{[\mathtt{lab}_0, \mathtt{lab}_1, \ldots, \mathtt{lab}_n, \ldots] \mid D_{0\chi_0}^{\rho_0}\,\emptyset \xrightarrow{\mathtt{lab}_0} D_{1\chi_1}^{\rho_1}\,\mu_1 \xrightarrow{\mathtt{lab}_1} \ldots \xrightarrow{\mathtt{lab}_{n-1}} D_{n\chi_n}^{\rho_n}\,\mu_n \xrightarrow{\mathtt{lab}_n} \ldots\}$$

progress $[\![D_{0\chi_0}^{\rho_0}]\!]_{\mathrm{progress}} = \{\pi \mid \pi \in [\![D_{0\chi_0}^{\rho_0}]\!]_{\mathrm{raw}}$ and for all configurations $D_{i\chi_i}^{\rho_i}\,\mu_i$ arising in π, if $D_i = R[D]$ for some R, D, then there is a later configuration $R'[D]_{\chi_{i+j}}^{\rho_{i+j}}\,\mu_{i+j} \xrightarrow{\mathtt{lab}_{i+j}}$ with redex the same subterm occurrence $D\}$.

fair $[\![D_{0\chi_0}^{\rho_0}]\!]_{\mathrm{fair}} = \{\pi \mid \pi \in [\![D_{0\chi_0}^{\rho_0}]\!]_{\mathrm{progress}}$ and each event e placed in μ during π's computation is eventually removed from μ at some later point in the computation $\}$

interaction paths $[\![D_{0\chi}^{\rho}]\!]_{\mathrm{IP}} = \{\pi_\chi^\rho \mid \pi_0 \in [\![D_{0\chi}^{\rho}]\!]_{\mathrm{fair}}$, π is π_0 with all events not of the form $\mathtt{in}(e)/\mathtt{out}(e)$ removed, and the events $\mathtt{in}(e)/\mathtt{out}(e)$ in π are all unique $\}$.

The semantics of a top-level diagram, $[\![D_\chi^\rho]\!]$, is then $[\![\{D\}_\chi^\rho]\!]_{\mathrm{IP}}$.

4.3 Commentary on the Definition

The single-step computation rules themselves are for the most part familiar territory for operational semantics presentations, with a few different aspects. They represent a language with syntax something like CSP, and asynchronous message passing and name handling modelled on the actor approach. We provide some commentry on what are perhaps some of the nonstandard aspects of the rules.

The `par` rule could just as well map `skip` $\|\, D$ to D. The `receive` rule contains implicit pattern-matching. Diagram variables in M_{d} are considered pattern variables, and are matched against the event e. Note that the receiver a_{d} could itself be a diagram variable, but this is not considered part of the pattern: it is not possible to receive a message destined for an arbitrary actor. $acq(\overline{s}) \subseteq acq(e)$ holds by the pattern match. In both send and receive, the keys κ are not "κ_{d}"–they are simple constants and cannot be variables which are defined in the environment. The `new` rule allocates variable x at the current lexical scoping level, and gives it an

arbitrary value based on the names of actors currently known. `fresh`, on the other hand, assigns a single actor name to x which is not currently known. In either rule, if x were already declared within the current lexical level, the old value is replaced. `assign` updates a variable based on ψ. The side condition only fails for the case a variable is assigned to which was never declared; this will cause the computation to get stuck and thus ruled out by lack of progress. `constrain` only continues to compute when the constraint holds; if not the computation is stuck and is ruled out by the progress requirement.

The most unusual aspect of the semantics lies in the details of the progress requirement. This requirement is significantly stronger than standard fairness requirements, and makes the computation system unrealizable. In fact, even without requiring progress the computation system is unrealizable because predicates $\phi(\rho,\chi,\varepsilon)$ may be undecidable, and thus for instance a deterministic halting-problem solver may be defined. However, assuming predicates must be decidable, the progressing paths still may not be realized by some actor computation. An example of such a non-realizable diagram is:

$$
\begin{array}{l}
\texttt{new}(\texttt{nomorezeros} = \textit{false});\\
[\ (\texttt{receive}(a \triangleleft 0);\ \texttt{constrain}(\neg\texttt{nomorezeros});\\
\ \texttt{nomorezeros} := \textit{true};\ \texttt{send}(c \triangleleft 1))\\
\oplus\\
(\texttt{receive}(a \triangleleft x);\ \texttt{constrain}(\neg(x = 0 \wedge \texttt{nomorezeros}));\\
\ \texttt{send}(c \triangleleft 0))\ ]^{0..\infty}
\end{array}
$$

it replies 0 to all inputs, except the *last* 0 input *may* get a 1 reply. No realizable system can forsee the future to know when the last input of a particular form has arrived.

Progress rules out any computation path which contains a parallel computation that is stuck, *i.e.,* does not reduce. The phrase "subterm occurrence", in analogy to the concept of *residual* in the lambda-calculus, denotes a particular occurrence of a subterm, because the same syntax may in theory occur multiple times in a single term. Each ocurrence must progress. A fully formal definition may be obtained by decorating each subterm with a unique label.

Particular computation paths that progress rules out include

- paths with false constraints such as $R[\texttt{constrain}(x = 0)]^{\rho}_{\chi}\,\mu$ where $R@x = 1$;
- paths which attempt receipt of a message on an unused local actor name, such as $R[\texttt{receive}(a \triangleleft M_{\mathrm{d}}\{:\kappa\})]^{\rho}_{\chi}\,\emptyset$ for actor $a \notin \rho \cup \chi$ that is never sent out of the configuration and which is sent no messages locally;

- paths which attempt receipt of a message, $R[\texttt{receive}(a \triangleleft M_{\mathrm{d}}\{:\kappa\})]^{\rho}_{\chi}\emptyset$ where in this particular computation path the environment will `in` no such message and it will not be sent locally either;
- paths which attempt receipt of a message on an external actor name, or send of a keyed packet to an external actor;
- assignments to variables that do not exist in the environment, $R[x := 0]^{\rho}_{\chi}\mu$ for $R@x = \bot$.

The above cases (excepting the first) are often a product of an ill-conceived diagram design. However, There is no firm line that may be drawn between the well-conceived and ill-conceived diagrams: well-conceived diagrams, when composed with other diagrams, may appear ill-conceived. Nonetheless a simple conservative aproximation of ill-conceived diagrams is possible that detects many obvious errors.

Definition 4.4: A diagram D^{ρ}_{χ} is *ill-conceived* if $[\![D^{\rho}_{\chi}]\!] = \emptyset$. Other diagrams besides these may reasonably be classified as ill-conceived.

If a diagram has no paths, it cannot be sensible. There are other diagrams which are intuitively not sensible, but there is no firmer line that can be drawn between sensible and not. Consider D that contain a subterm occurrence $D_l \oplus D_r$ for which all computation paths in $[\![\{D\}^{\rho}_{\chi}]\!]_{\mathrm{fair}}$ either invariably reduce this occurrence redex by $\texttt{choice}(l)$, or invariably by $\texttt{choice}(r)$. These choice operators may thus be simplified away to the always-chosen case only, and this may be due to an ill-conceived expression in the case never taken. However, it could also be due to the fact that in the context the subterm occurs in, the path not taken is not used, because the specification is more general than the current usage. For this reason, such cases are not invariably classified as ill-conceived.

An example which shows a need for keys is

$$\begin{array}{l}\texttt{new}(x);((x := 1;\texttt{send}(a \triangleleft x : \kappa) \oplus x := 2;\texttt{send}(a \triangleleft x : \kappa')) \parallel \\ \quad (\texttt{receive}(a \triangleleft x : \kappa) \oplus \\ \quad (\texttt{receive}(a \triangleleft x : \kappa');(\texttt{skip} \oplus (\texttt{constrain}(x \neq 2);\texttt{send}(a \triangleleft \texttt{BAD})))))\end{array}$$

—if the keys were removed, a κ-κ' communication could occur and `BAD` could be sent to a.

5 Toward an Algebra of Diagrams

We now outline the algebra of diagrams; work remains to be done in this area. See [Tal97] for full definition of the algebra of interaction path sets; basic definitions

were given previously in section 3. The algebra on diagrams is directly lifted from the algebra on *Ip* sets vie the semantic meaning function for diagrams, $[\![D^{\rho}_{\chi}]\!]$. When performing algebraic reasoning on diagrams, we use the convention that D^{ρ}_{χ} in fact stands for its interaction path semantics, $[\![D^{\rho}_{\chi}]\!]$. The algebraic operations on diagrams are inherited from the algebraic operations on interaction path sets:

Definition 5.1 (Composition, Restriction, Renaming):

$$\begin{array}{rl} D_1{}^{\rho_1}_{\chi_1} \parallel D_2{}^{\rho_2}_{\chi_2} & \text{means } [\![D_1{}^{\rho_1}_{\chi_1}]\!] \parallel [\![D_2{}^{\rho_2}_{\chi_2}]\!] \\ D^{\rho}_{\chi}\lceil\rho' & \text{means } [\![D^{\rho}_{\chi}]\!]\lceil\rho' \\ \widehat{\alpha}(D^{\rho}_{\chi}) & \text{means } \widehat{\alpha}([\![D^{\rho}_{\chi}]\!]) \end{array}$$

The notion of equivalence desired for top-level diagrams is an *extensional* one: we are not interested in internal structure of the diagrams, only that an actor configuration satisfies one specification if and only if it satisfies another. Thus two diagrams are defined to be equivalent when their interaction paths are the same.

Definition 5.2 (Extensional Equivalence of Diagrams): $D_1{}^{\rho_1}_{\chi_1} \cong D_2{}^{\rho_2}_{\chi_2}$ iff $[\![D_1{}^{\rho_1}_{\chi_1}]\!] = [\![D_2{}^{\rho_2}_{\chi_2}]\!]$.

Note that we use $\cong$ for extensional equivalence, reserving $=$ for syntactic identity of diagrams.

The composition of top-level diagrams is achieved just by forming a new diagram which places the composed diagrams in parallel. This allows for modular construction of diagrams and modular reasoning about diagram properties.

Theorem 5.3:

$$D_1{}^{\rho_1}_{\chi_1} \parallel D_2{}^{\rho_2}_{\chi_2} \cong (\{D_1\} \parallel \{D_2\})({}^{\rho_1}_{\chi_1}\!\parallel {}^{\rho_2}_{\chi_2}),$$

provided ${}^{\rho_1}_{\chi_1} \bowtie {}^{\rho_2}_{\chi_2}$ and for each $\texttt{receive}(a_{\mathrm{d}}, mp_{\mathrm{d}})$ occurring in D_1, by inspection, $a_{\mathrm{d}} \notin \rho_2$, and similarly for D_2.

The "by inspection" condition perhaps needs elaborating. One conservative interpretation would be that either $a_{\mathrm{d}} = a$ for $a \in \rho_1$, or $a_{\mathrm{d}} = x$ and $\varepsilon(x) \in \rho_1$ for all ε arising in $[\![D_1{}^{\rho}_{\chi}]\!]$. This condition is needed to guarantee messages are destined for one component or the other, and not both. In section 2 the `FunctionComposer` uses this theorem to show its components may be defined as separate top-level diagrams and composed. After components are composed, `send` and `receive` messages between the two components may be matched to give `send-receive` edges. The function composer example also illustrates this transformation. A future goal is a fully rigorous justification of this operation.

Restriction is elementary if newly restricted receptionists were in fact sent no messages in the specification:

Lemma 5.4:

$$D_\chi^{\rho\cup\rho'}\lceil\rho \cong D_\chi^\rho$$

provided for each $\texttt{receive}(a_\mathrm{d}, mp_\mathrm{d})$ occurring in D, by inspection, $a_\mathrm{d} \notin \rho'$.

What is defined here is equivalence on top-level diagrams. In general it will be desirable to define equivalance on diagram fragments; the natural idea is to use some sort of contextual equivalence *a la* Plotkin. We leave that topic for future work.

6 Related Work

There are a wide variety of notations for concurrent/distributed system specification. Different forms of specification have different strengths and weaknesses, and for large systems a number of different techniques will probably be needed in parallel. We briefly review some of the current schools by way of background.

Process Algebras Process algebra notation may be used to formally specify the communication actions of concurrent systems, and this was in fact one of the original goals of CCS [Mil80]. Process algebra and specification diagrams in fact share some significant similarities. Parallel composition is of a similar sort in both; choice in specification diagrams could be viewed as a generalization of CCS' external choice operator. message send and receive is analogous to the related concepts in the π-calculus [MPW92, HT91], although the π-calculus restricts data passed to be a channel name, and is in the classical presentation, synchronous as opposed to asynchronous. Name-passing and dynamic name creation are important to distributed systems and are treated in specification diagrams as well as the π calculus. The trace-based semantic framework is a concept shared with CSP [Hoa85].

There are differences as well, and the most important ones are found beneath the surface in the semantics of operators and not their syntax. The object-based behavior of specification diagrams is enforced by the interfaces; for this there is no analogue in process algebra since it is not object-based. There is a subtle difference in the meaning given to specification diagrams in comparison to process algebra. Simply put, process algebra is given a purely operational, realizable, interpretation. Even though specification diagrams have an operational semantics, this semantics is not realizable. If a constraint fails during computation, the computation *never happened*; it disappears from the set of possible paths. Constraints themselves may not be decidable properties. Specifications written in process algebra notation admit the possibility of deadlock since the environment may not send a particular

desired message. Specification diagrams, on the other hand, constrain the environment so that deadlock implicitly cannot occur; instead, either a specification is ill-formed, or composition of specifications will fail. Deadlock can in fact be *specified* in specification diagrams, by actively ignoring all input. The `Sink` example earlier is such an example. Specification diagrams allow communication to be constrained both at send and receive by cross-edges, so operationally speaking, if a message is not received by its intended receiver, that computation path never happened. There are advantages and disadvantages of uncomputable specification languages. The main advantage of uncomputable languages is their expressivity. The main advantage of computable languages is they generally possess more decidable properties.

Choice in specification diagrams could be called "extremely external" if the nomenclature of internal/external choice of CCS is used. Internal choice is a random coin flip which irrevocably picks one of two paths. External choice in its general form is a guarded choice; the path chosen must have the guard condition holding. In specification diagrams, the constraints allow a choice to be "un-chosen" even after it had been started, not just at the beginning. This is not an operational notion, but is useful in certain cases to allow for succinct specification of concurrent object behavior.

$$\texttt{in}(a_{\mathrm{d}} \triangleleft x);\ (\texttt{out}(a'_{\mathrm{d}} \triangleleft x);\ \texttt{in}(a_{\mathrm{d}} \triangleleft y);\ \texttt{constrain}(y > 0)) \oplus \texttt{skip}$$

–This specification has odd behavior of only forwarding a message when the *next* message is a positive number.

A number of full specification languages based on process algebra have been developed; examples include LOTOS [BB87], which is based on CSP; it is now an an ISO standard. Esterel [BG92] is a process algebra based specification language with a synchronous execution semantics.

Temporal Logic Temporal logic formulae have been extensively used as a means for logical specification of concurrent and distributed systems [JM86, MP92, Lam94]. While logics may express an extremely broad collection of properties, a significant disadvantage is the need for large, complex formulae to specify nontrivial systems: readability of specifications becomes a serious issue even for small specifications, and users thus require more advanced training. Specification diagrams are not a logic; as such, they cannot logically assert global properties of programs, only local properties via constraints. The equational theory of specification diagrams will provide the basis for more abstract reasoning about actor system components.

Automata-Based Formalisms Finite automata are useful for specifying systems which have a strong state-based behavior. They lack expressivity, but make up

for this lack by their amenability to automatic verification by state-space search techniques.

The StateCharts formalism [Har87] has become particularly popular in industry. States of the automaton represent states of the system (where certain invariant properties hold), and state transitions represent actions. The StateCharts formalism has features beyond simple finite automata, including the ability to nest and compose automata. This syntactic sugar makes it feasible to write specifications. Automata are also graphical and so serve as good visual specifications. Their primary weakness is that a complex software system may not have a meaningful global state, and properties of such systems are more naturally expressed in terms of events and relations on events. UML notation includes a StateCharts-based style of diagram. A formal semantics of StateCharts has been defined [HPPSS87], but the tools are not sound with respect to a formal semantics and so the effort is not completely satisfactory.

Message-Passing Diagrams Message-passing diagrams are a common form of informal graphical specification. A message-passing diagram has a time-line showing the message-passing behavior between different components. Unlike the other approaches described above, message passing specifications are usually object-based and can be asynchronous. The UML Sequence Diagram [RJB98] (derived in turn from the event trace diagram of [*et al*91]) is a simple form of message passing diagram for rpc-style communication. Specfication diagrams can be viewed as a major extension of UML sequence diagram notation. In the actor model, event diagrams [Gre75, Hew77] model actor computation in terms of message-passing between actors. Clinger[Cli81] formalizes event diagrams as mathematical structures and defines a formal semantics mapping actor system descriptions to sets event diagrams. More generally sets of event diagrams can be thought of as abstract specifications. These have rich mathematical structure but, are in general highly undecidable. Specification diagrams were partly inspired by event diagrams, and can be viewed as a condensation of a possibly infinite set of possibly infinite-sized event diagrams to one, finite, representation.

Acknowledgements

Thanks to Carolyn Talcott for many discussions and for comments on several versions of this document. Also thanks to Gul Agha and Ian Mason for helpful comments.

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[AMST97] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[Cli81] W. D. Clinger. *Foundations of Actor Semantics*. PhD thesis, MIT, 1981. MIT Artificial Intelligence Laboratory AI-TR-633.

[*et al*91] J. Rumbaugh *et al*. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Gre75] I. Greif. Semantics of communicating parallel processes. Technical Report 154, MIT, Project MAC, 1975.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

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

[HPPSS87] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings* 2^{nd} *Annual Symposium on Logic in Computer Science*, Ithaca, New York, pages 54–64, 1987.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1991.

[JM86] Farnam Jahanian and Aloysius Mok. Safety analysis of timing properties in real-time systems. *IEEE Transaction on Software Engineering*, 12(9):890–904, 1986.

[Lam94] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

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

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MT97] I. A. Mason and C. L. Talcott. A semantically sound actor translation, 1997. submitted.

[RJB98] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Tal97] C. L. Talcott. Composable semantic models for actor theories. In T. Ito M. Abadi, editor, *Theoretical Aspects of Computer Science*, number 1281 in Lecture Notes in Computer Science, pages 321–364. Springer-Verlag, 1997.