Specification Diagrams for Actor Systems

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

Carolyn L. Talcott (clt@cs.stanford.edu)
SRI International

Abstract. Specification diagrams (**SD**'s) are a novel form of graphical notation for specifying open distributed object systems. The design goal is to define notation for specifying message-passing behavior that is expressive, intuitively understandable, and that has formal semantic underpinnings. The notation generalizes informal notations such as UML's Sequence Diagrams and broadens their applicability to later in the design cycle. Specification diagrams differ from existing actor and process algebra presentations in that they are not executable *per se*; instead, like logics, they are inherently more biased toward specification. In this paper we rigorously define the language syntax and semantics and give examples that show the expressiveness of the language, how properties of specifications may be asserted diagrammatically, and how it is possible to reason rigorously and modularly about specification diagrams.

Keywords: specification, message passing behavior, actor systems, interaction semantics

1. Introduction

Our goal is to define a notation for specifying message-passing behavior that is expressive, intuitively understandable, and that has a rigorous underlying semantics. Many specification languages that have achieved widespread usage have a graphical presentation format, primarily because engineers can understand and communicate more effectively by graphical means. Popular graphical specification languages include The Unified Modeling Language (UML) and its predecessors [36], Petri nets, Statecharts [23], and SDL [28, 17]. Our aim here is to define a *specification diagram* (**SD**) language with similar intuitive advantages that combines greater expressivity with formal underpinnings. The language is also designed to be useful throughout the development process to give: initial sketches of the overall system structure, informal scenarios of possible behavior early in the design cycle, or detailed specifications of individual components that may serve as formal documentation of critical aspects of their behavior and support rigorous checking that an implementation meets a given specification. It may be used as a programming language, or for giving diagrammatic assertions of correctness that encompass safety and liveness properties that can be understood in terms of the semantics of the language without need to use a separate logic. There is a textual form which is useful for formal manipulations. The design of the **SD** language draws on concepts from actor event diagrams [12], UML

Sequence Diagrams [36], process algebra [32, 26], and Dijkstra-style weakest precondition calculus [34]. The underlying interaction model is that of the actor model: object- and not channel-based naming is used, open systems are treated explicitly, and message passing is asynchronous, fair, and with nondeterministic arrival order.

This section concludes with a brief introduction to the underlying actor interaction model. Section 2 introduces the diagram syntax, and section 3 illustrates the use of **SD**'s with a variety of examples. The operational and denotational (interaction) semantics of **SD** components is given in section 4. Section 5 illustrates the use of the semantic framework to reason about specifications.

1.1. Actor Concepts

We now provide a brief overview of the underlying actor model [24, 8, 2, 3]. Actors are independent computational agents that interact solely via asynchronous message passing. Actors may dynamically create other actors. Like objects in an object-based system, each actor has a unique *name* and a message can only be read (received) by the actor to whom it was sent. All actor computations must obey the actor locality laws [8]: the names an actor can know are those given to it at creation time, names it has received in a message, and names of actors it has created; an actor can only send messages to actors whose names it knows; messages can only contain names of actors known by the sender; and the names given to an actor at creation time must be among those known by the creator. Finally, all messages must eventually arrive at their destination, possibly with an arbitrarily long delay (fairness).

Individual actors are collected into open distributed components called *actor system components*. Each component specifies only part of a system; there will be some *external actors* that actors of the local system know about and may interact with. Additionally, of the local actors, only some of their names may be known by external entities; these are the *receptionists*. 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. We use ρ to denote the set of receptionists and χ to denote the set of external actors of an actor system component. The pair (ρ, χ) is called the system *interface* and we write S^{ρ}_{χ} to indicate an actor system component S with receptionists ρ and known external actors χ. The interface can be thought of as an abstract input/output specification. Since an actor can only send messages to actors it knows, actors outside a component can only send messages to actors in the component that are receptionists (input) and conversely, actors in the component can only send messages to actors outside that are in the set of externals (output). Note that the system may contain actors not in ρ; these actors are locally known only. Dually, there can be other actors external to the system not in χ.

What we observe about an actor system component is its patterns of interaction with its environment. To describe such interactions we need at least a set of *actor names* and a set of *messages*. Formally we have

Definition 1.1 (Actor Communication Basis): An actor communication basis provides a countably infinite set $a \in \mathbf{A}$ of actor names and a set $M \in \mathbf{Msg}$ of messages. Message packets mp are of the form $a \triangleleft M$, indicating message M is sent to actor a. We let $\mathbf{MP}$ be the set of message packets. We assume given a function $acq(M)$ that returns the finite set of actor names (acquaintances) communicated in a message.

An individual "run" of an actor system component is modeled by a possibly infinite sequence of inputs, $\mathtt{in}(a \triangleleft M)$, and outputs, $\mathtt{out}(a \triangleleft M)$, indicating the communication events and the order they came in and out of the system. We call one such infinite sequence (together with the system interface) an *interaction path*, and model the behavior of an actor system component by the set of all of its possible interaction paths. A component will also have internal computation actions, but these are not observable outside the component and so do not occur in the interaction paths. The *computation paths* of a component consist of its internal actions interleaved with input/output events. Interaction paths are defined by first determining the computation paths, and then projecting out the internal actions. We use this infinite trace form of model because it directly corresponds to the observable system behavior: watching a given component run, the input and output operations in sequence are exactly what can be observed. Other models including finite traces and bisimulation equivalences also may of course be used.

2. Syntax

In this section we present the syntax of **SD**'s, and an informal description of their meaning. We use two forms of notation for diagrams, one graphical and one textual. The graphical one is intended for use in practice: the graphical drawings are highly intuitive. However for mathematical study the textual form is perhaps easier to manipulate. Figure 1 presents the graphical diagram elements. Vertical lines indicate progress in time going down, expressing abstract causal ordering on events, with events above 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, as the threads exist only at the semantic level: a single thread of causality may involve multiple actors, and a single actor may appear to have multiple threads of causality. The components D in the figure may themselves be any diagram: the figure is a graphical grammar. The base case diagram elements

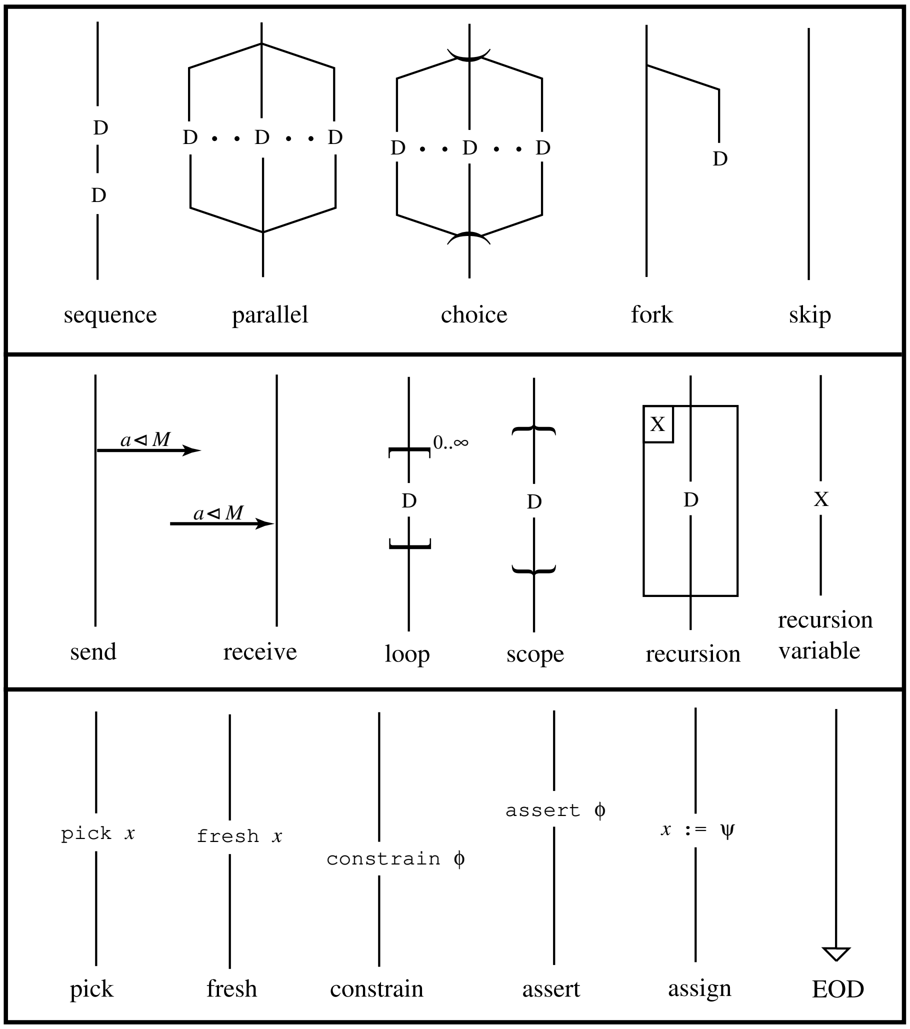

Figure 1. Specification Diagram Elements

such as `send`, `receive`, $:=$, etc., are executed atomically. The figures in the next section present several simple examples of **SD**'s.

Before describing the diagram elements one-by-one, we define the basic mathematical building blocks. We assume given a mathematical universe, $\mathbf{U}$, which contains (as subsets) at least $\mathbf{A}$, $\mathbf{Msg}$, $\mathbf{MP}$, $\mathbf{Nat}$ (the natural numbers), and $\mathbf{Bool}$ (the booleans $true$ and $false$). We use mathematical set theory to define operations and relations on $\mathbf{U}$. Diagrams have a state component represented using state variables which take on values in $\mathbf{U}$. $\mathbf{X}_{\mathrm{d}}$ is the set of diagram state variables and $x, y, z, \ldots$ denote elements of $\mathbf{X}_{\mathrm{d}}$. An environment γ is a finite function from $\mathbf{X}_{\mathrm{d}}$ to $\mathbf{U}$ used to model the assignment of state variables to their values. The remaining building blocks are expressions used in a variety of ways: assignments, constraints, message patterns. Rather than fix a syntax for expressions we assume given a set, $\mathbf{U}_{\mathrm{d}}$, of abstract expressions. Formally an abstract expression is a partial function from environments to values in the universe. We refer to the application of an expression to an environment as its evaluation in that environment. We let ψ range over $\mathbf{U}_{\mathrm{d}}$, M_{d} range over abstract expressions with value in $\mathbf{Msg}$, a_{d} range over abstract expressions with value in $\mathbf{A}$, and ϕ range over abstract expressions with value in $\mathbf{Bool}$. We say that the variable x *occurs in* ψ if $\psi(\gamma) \neq \psi(\gamma')$ for some γ, γ' which differ just at the variable x: $\mathrm{Dom}(\gamma) \cup \{x\} = \mathrm{Dom}(\gamma') \cup \{x\}$ and $\gamma(y) = \gamma'(y)$ for $y \in (\mathrm{Dom}(\gamma) - \{x\})$. The above conditions imply that x is in the domain of at least one of γ or γ' and if x is in the domain of both, then the two environments assign x different values. Our convention regarding equality of partial terms t, t' is that $t = t'$ means both are undefined, or both are defined and the values are equal. This is equivalent to extending $\mathbf{U}_{\mathrm{d}}$ with a new element to represent undefinedness. Abstractly, ψ is just a partial function. In practice it could be undefined on some environment for many reasons, for example: a variable whose value is needed is not defined; or a function used to compute the meaning of ψ is not defined for the particular valuation of variables.

Abstract message expressions M_{d} may be used as patterns in much the same way that algebraic terms serve as patterns in presenting conditions or rewrite rules. In the case of message patterns, the pattern variables are the state variables occurring in M_{d} and a message M matches M_{d} just if there is an environment γ whose domain is the set of pattern variables of M_{d} such that $M_{\mathrm{d}}(\gamma) = M$. In this case we say that γ binds the pattern variables to their matching values. For example if x is a diagram state variable, then $\mathtt{set}(x)$ is a message pattern and the message $\mathtt{set}(0)$ matches this pattern binding x to 0.

We use the informal convention that expression ψ written as $x + 1$ is an abbreviation for $\psi(\gamma)$ equal to $\gamma(x) + 1$, and "$x \in S$" means a predicate ϕ where $\phi(\gamma)$ iff $\gamma(x) \in S$.

The individual graphical elements are now informally described. With each element, the textual grammatical equivalent is given in parentheses.

sequence ($D_1; D_2$) Vertical lines (causal threads) represent necessary temporal sequencing of events in D_1 before those in D_2.

parallel ($D_1 \mid D_2$) Events in parallel diagrams have no causal ordering between them, but are after events above and before events below.

choice ($D_1 \oplus D_2$) One of the possible choices is taken. 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.

fork (`fork`(D)) A diagram is forked off which hereafter will have no direct causal connection to the future of the current thread (however, messages could indirectly impose some causality between the two). The parallel and fork operators are similar, but parallel threads must eventually merge, while forked threads are asymmetrical in that the forked threads never merge.

skip (`skip`) Does nothing.

send (`send`$(a_\mathrm{d} \triangleleft M_\mathrm{d})$) A message is sent to a_d with contents M_d. There is a requirement that message delivery be fair, in the sense that any message sent must eventually arrive at its destination.

receive (`receive`$(a_\mathrm{d} \triangleleft M_\mathrm{d})$) A message matching M_d is received by actor a_d, the pattern variables occurring in M_d are bound to the matching values in the scope of the `receive`. This statement blocks until a message arrives matching its pattern. If none arrives, the computation path is considered unfair and not admitted. Dually, if a message arrives but is never matched by any `receive`, that computation path is also considered unfair and is not admitted.

loop (`[`D`]`$^{0\ldots\infty}$) The diagram is iterated some number n times, where n is nondeterministically chosen from the interval $0\ldots\infty$. The case $n = \infty$ corresponds to loop-forever. The textual syntax here is not in the core language—it is defined in the macro library below, along with variants.

scope ($\{x_0, \ldots, x_n : D\}$) Brackets demarcate static scoping of state variables. In the official textual syntax explicit variable declarations must be given (and the variables initially given arbitrary values), but by an implicit convention discussed below, bracketing alone may be used to define variable extent.

EOD (`eod`) Denotes the end of a causal thread in the diagram. This is also not core syntax and is defined in the macro library.

pick (`pick`(x)) State variable x is assigned arbitrary contents.

fresh (`fresh`(x)) State variable x is given contents consisting of an actor name not currently in use.

constraint (`constrain`(ϕ)) A constraint ϕ is placed on the current state of the computation, which must be met. Otherwise the computation path is not admitted.

assertion (`assert`(ϕ)) An assertion ϕ is made. Unlike `constrain`, an `assert` that evaluates to false is an explicit signal of failure of some property, but otherwise, an well-formed assertion (that evaluates to a boolean value) has no computational effect.

assign($x := \psi$) A variable is dynamically assigned a new value given by evaluating the assignment body, ψ in the current environment.

recursion ($\texttt{rec}\, X.D$, X) A boxed diagram fragment may refer to itself by name, X, so X occurring inside the box refers to the whole box.

recursion variable (X) $\mathbf{X}_{\mathrm{r}}$ is a countable set of recursion variables, with $X \in \mathbf{X}_{\mathrm{r}}$.

In the textual language, sequencing is right-associative and binds most tightly, followed by choice and then parallel composition binding most loosely. Choice and parallel composition are also associative and in diagram notation we treat them as multi-ary constructs rather than iterating the binary construct.

We define some syntactic sugar which allows variable declarations at a scope boundary to be implicit: $\{D\}$ abbreviates $\{x_0, \ldots, x_n : D\}$ for state variables $x_0, \ldots, x_n$ all occurring directly in D (not in a deeper lexical level) as $\texttt{pick}(x_i)$, $\texttt{fresh}(x_i)$, or $\texttt{receive}(a_{\mathrm{d}} \triangleleft M_{\mathrm{d}})$, with x_i occurring in M_{d}.

We use the convention that $\mathrm{ExampleMacro}(s, t, u) = D$ defines a macro. Macros are just diagram producing functions: we will be careful not to define self-referential macros. Macro parameters (s, t, u in the example) are taken to be meta-variables ranging over $\mathbf{U}$ and not state variables. In order to avoid ambiguity between the two kinds of variable, all metavariables must be explicitly listed as the parameter to a macro. Certain syntax is easily encodable via macros and so is not defined in the core grammar. Here are some examples.

Definition 2.1 (Diagram Macro Library):

eod: $\texttt{eod} = \texttt{rec}\, X.(\texttt{skip}; X)$

initialized pick: $\texttt{pick}(x = u) = \texttt{pick}(x); \texttt{constrain}(x = u)$

constrained pick: $\texttt{pick}(x \in X) = \texttt{pick}(x); \texttt{constrain}(x \in X)$

loop: $[\,D\,]^{0 \ldots \infty} = \texttt{rec}\, X.((D; X) \oplus \texttt{skip})$

finite iteration: $[\,D\,]^{0 \ldots \omega} =$

$$\begin{array}{l} \texttt{pick}(x \in \mathbf{Nat}); \\ \texttt{rec}\, X.\texttt{constrain}(x = 0) \oplus \texttt{constrain}(x > 0); x := x - 1; D; X \end{array}$$

where x is a state variable not occurring in D

loop-forever: $[\,D\,]^{\infty} = \texttt{rec}\, X.(D; X)$

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

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

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

failure: $\texttt{fail} = \texttt{assert}(\mathit{false})$

initialized decl: $\{\ldots, x = u, \ldots : D\} =$
$\{\ldots, x, \ldots : \texttt{constrain}(x = u); D\}$

constrained decl: $\{\ldots, x \in S, \ldots : D\} =$
$\{\ldots, x, \ldots : \texttt{constrain}(x \in S) : D\}$

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 graphical diagrams into textual notation is obtained by inductively replacing the graphical syntax with the corresponding textual syntax listed above. Note that macros such as the `initial/constrained pick/decl` macros do not provide atomic execution of the sequence of actions in their expansion. Thus they have the "expected" semantics only if not placed in parallel with diagrams that read/write the assigned/constrained variables. Atomicity, if desired, could be obtained by using more complex macros. A *top-level* **SD** includes an interface, notated $\langle D \rangle^{\rho}_{\chi}$. Top-level diagrams are actor system components (§ 1.1) and are given meaning in terms of a transition system from which sets of interaction paths are derived. We will not always include the phrase "top level" but meaning should be clear from context.

SD's combine features typically found in concurrent object-based programming languages— notions of local name, variable, assignment, loop, if-then, and message send and receipt—with features more appropriate for specification language, including assertions, constraints, picking a value from a (possibly infinite) set of possible values. They allow name passing and the dynamic generation of fresh names, a feature shared with the π-calculus, while obeying the locality (acquaintance) laws of actor computation [8]. As we will see in section 4 the fairness properties of computations described by **SD**'s differ those of traditional actor languages. The notion of constraint is analogous to the **assume** predicate of Dijkstra's language [15], while the assertion primitive is analogous to Dijkstra's **assert** predicate. The constrain predicate in particular is an important element of the language, greatly increasing its power. A constraint may be any mathematical predicate, and a constraint failing does not indicate a run-time error, it indicates that such a computation path will not arise, i.e. the path is "cancelled in the middle of computing" as if it never happened. This predicate is thus impossible to implement in full generality, but it gives the specification language significant flexibility. Examples in the next section should make this clear. There are some surface similarities of `constrain` and `assert` with the ask and tell agents of concurrent constraint programming [37]. Tell constraints restrict the solution space, and ask constraints fire only when their precondition is in the solution space. `constrain` has some commonality with tell in that it restricts the solution space, but the case of unsatisfiability is handled very differently—in our system the entire computation vanishes as if it never happened, whereas in CLP, failure value is returned. The ask constraints are a guarded, blocking form of constraint which has no real analogue here. **SD** assertions are used to express properties of state variables at given points in the computation, and can be used to express both safety and liveness properties as the examples below will demonstrate. Assertions are a means of observing the system as it evolves. A well-formed assertion (one that always evaluates to a boolean) has no effect on the interaction semantics of an **SD**.

3. Using Specification Diagrams

In this section we illustrate the full range of functionality of the language via examples. Starting with simple examples that show how various patterns of message passing may be specified, we progress to examples of how different specifications may be related and how more advanced properties may be expressed.

3.1. Expressing patterns of message passing

We give here a series of examples illustrating how **SD** constructs may be used to specify component behaviors and scenarios, and give a rough idea of the meaning of diagrams as sets of interaction paths.

3.1.1. *Simple Memory Cell*

This simple cell holds a single value, and responds to `set` and `get` messages.

$\mathrm{Cell}(a) =$

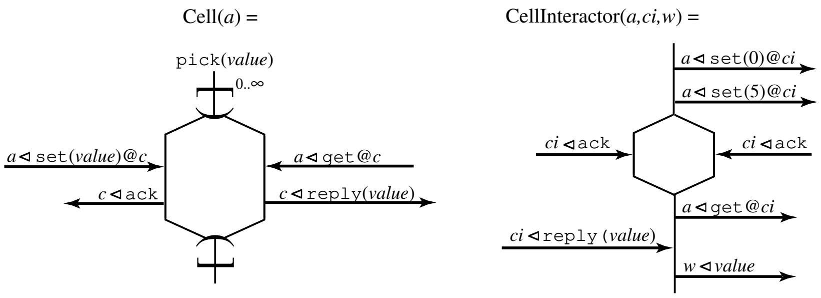

Figure 2. A Memory Cell and an Example Cell Interactor

$\{$ $\mathtt{pick}(value)$; `(* cell value, initially arbitrary *)`
`[` $(\mathtt{receive}(a \triangleleft \mathtt{set}(value)@c);\ \mathtt{send}(c \triangleleft \mathtt{ack}))$
$\oplus$
$(\mathtt{receive}(a \triangleleft \mathtt{get}@c);\ \mathtt{send}(c \triangleleft \mathtt{reply}(value)))$ `]`$^{0\ldots\infty}$ $\}$

The corresponding graphical **SD** appears on the left of Figure 2. The "$0 \ldots \infty$" iteration models all possible environment behaviors: the environment may send 0, an arbitrary finite number, or infinitely many $\mathtt{set}/\mathtt{get}$ requests. In the $a \triangleleft \mathtt{set}(value)@c$ message pattern, variables $value$ and c are implicitly declared pattern variables, bound by the receipt action. For example, the message $a \triangleleft \mathtt{set}(2)@\mathtt{sam}$ matches this pattern, binding $value$ to 2 and c to $\mathtt{sam}$. $\mathtt{send}(c \triangleleft \mathtt{ack})$ is an asynchronous send, and thus a receive of another $\mathtt{set}/\mathtt{get}$ request can immediately follow, even before the $\mathtt{ack}$ has arrived at its destination. The "$\{\ldots\}$" around the whole specification is the static scoping construct. By the implicit convention of the previous section, the notation without any variables listed (as above) indicates all variables are declared. So, the above is shorthand for "$\{\ value, a, c\ : \ldots\}$." The notation `(* ... *)` is used to add comments to textual **SD**'s.

A top-level (interfaced) specification for a memory cell is $\langle \mathrm{Cell}(a)\rangle^{a}_{\emptyset}$. The meaning of this top-level diagram, $[\![\langle \mathrm{Cell}(a)\rangle^{a}_{\emptyset}]\!]$, is a set Ip_{Cell} of interaction paths. The **SD** meaning function $[\![\cdot]\!]$ is formally defined in Definition 4.10 of Section 4. Interaction sequences in Ip_{Cell} include ones beginning as follows:

(1) $\mathtt{in}(a \triangleleft \mathtt{set}(0)@c), \mathtt{out}(c \triangleleft \mathtt{ack}), \mathtt{in}(a \triangleleft \mathtt{get}@c), \mathtt{out}(c \triangleleft \mathtt{reply}(0)), \ldots$

(2) $\mathtt{in}(a \triangleleft \mathtt{set}(0)@c), \mathtt{in}(a \triangleleft \mathtt{get}@c), \mathtt{out}(c \triangleleft \mathtt{reply}(x)), \mathtt{out}(c \triangleleft \mathtt{ack}), \ldots$
for x arbitrary

(3) $\mathtt{in}(a \triangleleft \mathtt{set}(0)@c), \mathtt{out}(c \triangleleft \mathtt{ack}), \mathtt{in}(a \triangleleft \mathtt{get}@c'), \mathtt{out}(c' \triangleleft \mathtt{reply}(0)), \ldots$

(4) $\mathtt{in}(a \triangleleft \mathtt{set}(0)@c), \mathtt{out}(c \triangleleft \mathtt{ack}), \mathtt{in}(a \triangleleft \mathtt{set}(1)@c'),$
$\mathtt{in}(a \triangleleft \mathtt{get}@c), \mathtt{out}(c \triangleleft \mathtt{reply}(1)), \mathtt{out}(c' \triangleleft \mathtt{ack}) \ldots$

(5) $\mathtt{in}(a \triangleleft \mathtt{set}(0)@c), \mathtt{in}(a \triangleleft \mathtt{set}(5)@c), \mathtt{out}(c \triangleleft \mathtt{ack}),$
$\mathtt{out}(c \triangleleft \mathtt{ack}), \mathtt{in}(a \triangleleft \mathtt{get}@c), \mathtt{out}(c \triangleleft \mathtt{reply}(0)), \ldots$

In the second example, since the `get` is sent before arrival of the `ack`, the value received may have been the original (arbitrary) value placed in the cell. The third example shows how customers may be different for each message. The fourth example shows how a `get` may in fact get a reply which incorporates a previous `set` that has not in fact been observably `ack`'ed yet, due to internal buffering delaying the final `ack`. Example five shows how multiple `set` requests will not necessarily be handled in the order they came in, since there is buffering inside the system which is independent of arrival order.

In the operational semantics for **SD**'s, each internal action appears in the *computation paths*, and these internal actions are then projected out to give the interaction paths. An example computation path which could produce interaction path (1) above could start as:

$$\mathtt{pick}, \mathtt{in}(a \triangleleft \mathtt{set}(0)@c), \mathtt{choose}(l), \mathtt{receive}(\mathtt{set}(0)@c),$$
$$\mathtt{send}(c \triangleleft \mathtt{ack}), \mathtt{out}(c \triangleleft \mathtt{ack}), \ldots$$

`pick` is the choice of arbitrary initial $value$, $\mathtt{choose}(l)$ indicates the left branch of the choice was taken, and the internal `receive`/`send` are also actions in the computation path. The internal actions are all projected out in forming the interaction path (1) above, since they are not part of the observable behavior of the system.

Choice, $\oplus$, in **SD**'s most closely corresponds to what is often called "internal" choice in process algebra: a coin is flipped and one branch is taken. However, it is not quite that simple, because the combination of choice and the constraints imposed by fairness of message delivery allow "external" forms of choice to be represented with the internal choice operator. In particular for the Cell, the choice between whether to try to receive a `set` or `get` is made internally by a coin flip, *but*, if the `get` branch was chosen and no `get` messages are forthcoming, the path is unfair and will not arise. This subtle interplay between choice and fairness is one of the features of the language that takes some getting used to, but it is an elegant and powerful mechanism. This topic is discussed further in Section 4.3.3 below, after the formal semantics have been given.

The interaction of path (5) may be expressed diagrammatically via this CellInteractor, also shown in Figure 2:

$$\mathrm{CellInteractor}(a, ci, w) =$$
$$\{\ \mathtt{send}(a \triangleleft \mathtt{set}(0)@ci);\ \mathtt{send}(a \triangleleft \mathtt{set}(5)@ci);$$
$$(\mathtt{receive}(ci \triangleleft \mathtt{ack}) \mid \mathtt{receive}(ci \triangleleft \mathtt{ack}));$$
$$\mathtt{send}(a \triangleleft \mathtt{get}@ci);\ \mathtt{receive}(ci \triangleleft \mathtt{reply}(value)); \mathtt{send}(w \triangleleft value)\ \}$$

The final $value$ sent to external w in combined top-level **SD**

$$\langle \mathrm{CellInteractor}(a, ci, w) \mid \mathrm{Cell}(a) \rangle_w^{\emptyset}$$

might be either 0 or 5. This is an example of using **SD**'s to describe possible scenarios. Note that if ci were a receptionist of the top-level diagram, the final $value$ sent could be arbitrary since the `ack` messages could have been sent in from the outside. A path with this behavior is

$$\mathtt{in}(ci \triangleleft \mathtt{ack}), \mathtt{in}(ci \triangleleft \mathtt{ack}), \mathtt{out}(w \triangleleft u), \ldots \text{ for } u \text{ any message}$$

If the CellInteractor had created a fresh actor name to serve as customer via $\mathtt{fresh}(c)$ as its first step, and used c instead of ci in the `set` and `ack` messages, the above anomalous behavior would not arise since the `ack` channel would not be externally visible.

3.1.2. *Ticker*

A ticker is a simple monotonically increasing counter which replies to `time` messages with its current count value. This example illustrates the unbounded nondeterminism present in actor computations. A high-level specification is as follows; see Figure 3 for a diagram.

$$\begin{array}{l}
\mathrm{Ticker}(a) = \\
\quad \{\ \mathtt{pick}(count \in \mathbf{Nat}); \\
\quad\quad \mathtt{[}\ \mathtt{[}\ \mathtt{receive}(a \triangleleft \mathtt{time}@x);\ \mathtt{send}(x \triangleleft \mathtt{reply}(count))\ \mathtt{]}^{0\ldots\omega}; \\
\quad\quad\quad count := count + 1\ \mathtt{]}^{0\ldots\infty}\ \}
\end{array}$$

This succinct specification expresses the fact that the count can stay constant for finitely many $(0 \ldots \omega)$ `time` requests, but then must increase. The count may also be repeatedly incremented without receiving any `time` requests, if the $0 \ldots \omega$ repeatedly chooses 0. However, if there is a `time` request waiting to be received, then a non-zero choice must eventually occur. A top-level ticker is $\langle \mathrm{Ticker}(a) \rangle_{\emptyset}^{a}$. The interaction paths in $[\![\langle \mathrm{Ticker}(a) \rangle_{\emptyset}^{a}]\!]$ consist of $a \triangleleft \mathtt{time}@c$ requests coming `in` and replies $c \triangleleft \mathtt{reply}(count)$ going `out` of the system. It is incorrect to state that the $count$ reply values sent out will be monotonically increasing, due to local buffering that may have happened; but, the observed $count$ values must eventually increase.

3.1.3. *Ticker Factory*

This example of a factory for producing tickers shows how fresh actors may be dynamically generated, and shows a use of `fork`. It also appears in Figure 3.

$$\mathrm{TickerFactory}(a) =$$

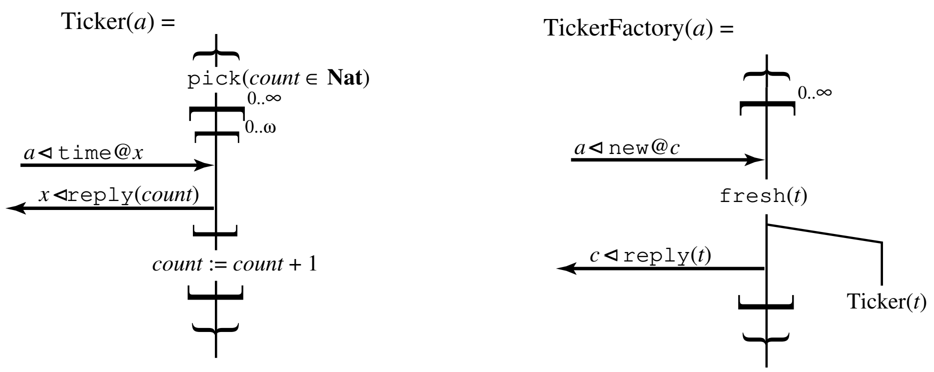

Figure 3. High-level Specification of a Ticker and a Ticker Factory

$$\{\ [\ \mathtt{receive}(a \triangleleft \mathtt{new}@c);\ \mathtt{fresh}(t);\ \mathtt{fork}(\mathrm{Ticker}(t));\ \mathtt{send}(c \triangleleft \mathtt{reply}(t))\ ]^{0\ldots\infty}\ \}$$

The interaction paths in $[\![\langle \mathrm{TickerFactory}(a)\rangle_{\emptyset}^{a}]\!]$ contain interactions with the ticker-factory actor: incoming requests $\mathtt{in}(a \triangleleft \mathtt{new}@c)$ and associated outgoing replies $\mathtt{out}(c \triangleleft \mathtt{reply}(t))$. t is the name of the newly created ticker actor and is exported in the reply. This causes t to be added to the set of receptionists, initially just $\{a\}$. Thus the interaction paths contain as sub-paths interaction paths for each exported ticker-actor.

3.1.4. *Function Composer*

In this example we define a distributed method for computing $g \circ f$ for composable functions f and g: the functions are computed by two different actors that are coordinated by a third actor. Graphical diagrams for this specification are in Figure 4.

For any actor name, a, and function $f \in V \to W$, the following diagram $\mathrm{F}(a, f)$ specifies a component that accepts requests to a of the form $\mathtt{compute}(v)@c$ and sends to c a $\mathtt{reply}(f(v))$.

$$\mathrm{F}(a, f) =$$

$$\{\ x, xc :\ [\ \mathtt{receive}(a \triangleleft \mathtt{compute}(x)\ @\ xc);\ \mathtt{send}(xc \triangleleft \mathtt{reply}(f(x)))\ ]^{0\ldots\infty}\ \}$$

For any actor names a, af, ag, $\mathrm{FC}(a, af, ag)$ specifies a component that accepts requests to a of the form $\mathtt{compute}(v)@c$, asks af to $\mathtt{compute}$ on v, and then asks ag to $\mathtt{compute}$ on the result from af, sending that result to c.

$$\mathrm{FC}(a, af, ag) =$$

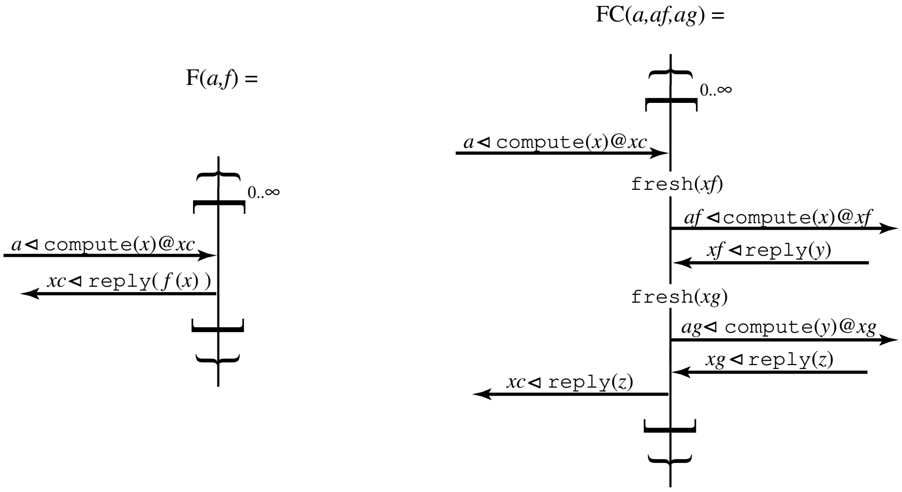

Figure 4. Function Composer Specifications

$$
\begin{array}{l}
\{\ x, y, z, xc, xf, xg : \\
\ [\ \mathtt{receive}(a \triangleleft \mathtt{compute}(x) \ @\ xc); \mathtt{fresh}(xf);\ \mathtt{send}(af \triangleleft \mathtt{compute}(x) \ @\ xf); \\
\quad \mathtt{receive}(xf \triangleleft \mathtt{reply}(y));\ \mathtt{fresh}(xg);\ \mathtt{send}(ag \triangleleft \mathtt{compute}(y) \ @\ xg); \\
\quad \mathtt{receive}(xg \triangleleft \mathtt{reply}(z));\ \mathtt{send}(xc \triangleleft \mathtt{reply}(z))\ ]^{0 \ldots \infty}\ \}
\end{array}
$$

The fresh names xf and xg are private names FC sends as the customer to the target af or ag only. The target in turn replies to this actor name, guaranteeing the reply came from the target or the target's accomplice, and ruling out the possibility of spoofing. This is because outsiders may only send to actors in the receptionist set, and these names will not be there. Thus in a context where af is the name of an f computer and ag is the name of a g computer, FC coordinates af and ag to become a $g \circ f$ computer. $\mathrm{C}(a, f, g, af, ag)$ puts FC in such a context:

$$\mathrm{C}(a, f, g, af, ag) = (\mathrm{FC}(a, af, ag) \mid \mathrm{F}(af, f) \mid \mathrm{F}(ag, g))$$

The full specification of $\mathrm{C}(a, f, g, af, ag)$ appears diagrammatically in Figure 5.

3.2. Relating Specification Diagrams

SD's can be used in a wide range of roles: as an implementation level description expressing low-level code-like details, for a high-level but precise behavioral description; and for a description of abstract properties of the possible interactions of a system. In this section we introduce notions of interaction refinement and equivalence used to relate **SD**'s. These relations

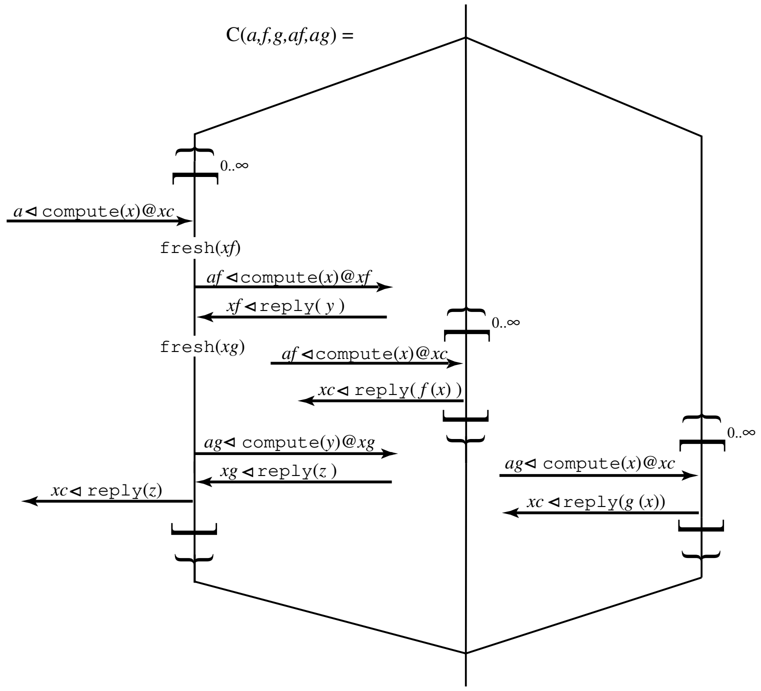

Figure 5. Composition of the Function Composer

are defined in terms of the semantic function $[\![\_]\!]$ (see Definition 4.11 in section 4), which gives the semantics of a top-level diagram as a set of interaction paths.

3.2.1. *Interaction Refinement*

One specification is an *interaction refinement* of another if every interaction path permitted by the first is also permitted by the second.

Definition 3.1 (Interaction Refinement):

$$\langle D_I \rangle^{\rho}_{\chi} \overset{i}{\subseteq} \langle D_S \rangle^{\rho}_{\chi} \quad \text{iff} \quad [\![\langle D_I \rangle^{\rho}_{\chi}]\!] \subseteq [\![\langle D_S \rangle^{\rho}_{\chi}]\!]$$

The notion of refinement appears in many specification formalisms including: algebraic specifications [43], Back's action systems [7, 6], the Abadi and Lamport work on refinement mappings [1] (**SD** refinement corresponds to their *implements* relation), and in the work on CSP [26, 35]. CSP is the most closely related to our approach and we discuss this relationship in § 6.

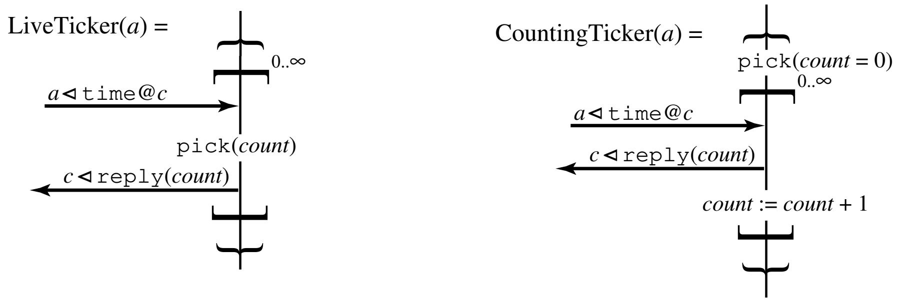

Figure 6. Live Ticker and Counting Ticker Specifications

3.2.2. *Refinement and the Ticker*

As an example of using the refinement relation consider the following **SD** (shown in Figure 6):

$$\mathrm{LiveTicker}(a) =$$

$$\{\ [\mathtt{receive}(a \triangleleft \mathtt{time} @ c);\ \mathtt{pick}(count);\ \mathtt{send}(c \triangleleft \mathtt{reply}(count))\,]^{0\ldots\infty}\ \}$$

This specification of the ticker only requires that all time requests receive a reply. The refinement relation

$$\langle \mathrm{Ticker}(a) \rangle^{a}_{\emptyset} \ \overset{i}{\subseteq}\ \langle \mathrm{LiveTicker}(a) \rangle^{a}_{\emptyset}$$

then is in effect asserting the liveness of the Ticker. (Note, such a liveness property was termed *responsiveness* by Hewitt [5].)

Another form of refinement is when one **SD** (thought of as an implementation) is more deterministic than the other (thought of as a specification). For the ticker, an implementation could be

$$\begin{array}{l} \mathrm{CountingTicker}(a) = \\ \quad \{\ \mathtt{pick}(count = 0); \\ \quad\ \ [\ \mathtt{receive}(a \triangleleft \mathtt{time} @ c); \\ \quad\ \ \ \ \mathtt{send}(c \triangleleft \mathtt{reply}(count)); \\ \quad\ \ \ \ count := count + 1\ ]^{0\ldots\infty}\ \} \end{array}$$

This implementation replies with a number one bigger, refining the Ticker specification.

$$\langle \mathrm{CountingTicker}(a) \rangle^{a}_{\emptyset} \ \overset{i}{\subseteq}\ \langle \mathrm{Ticker}(a) \rangle^{a}_{\emptyset}$$

Note that by transitivity,

$$\langle \mathrm{CountingTicker}(a) \rangle^{a}_{\emptyset} \ \overset{i}{\subseteq}\ \langle \mathrm{LiveTicker}(a) \rangle^{a}_{\emptyset}$$

3.2.3. *Maximal and Minimal Diagrams*

To better understand the dynamic nature of receptionist and external actor sets, consider the following specification.

$$
\begin{array}{l}
\mathrm{MaximalSpec}(\rho_0, \chi_0) = \\
\quad \texttt{pick}(\mathit{rho} = \rho_0, \mathit{chi} = \chi_0); \\
\quad [\texttt{pick}(r \in \mathit{rho});\ \texttt{receive}(r \triangleleft m);\ \mathit{chi} := \mathit{chi} \cup \mathit{acq}(m) - \mathit{rho} \\
\quad \oplus \\
\quad \texttt{pick}(z \in \mathit{chi});\ [\texttt{fresh}(q)]^{0\ldots\omega};\ \texttt{pick}(x); \\
\quad \texttt{send}(z \triangleleft x);\ \mathit{rho} := \mathit{rho} \cup \mathit{acq}(x) - \mathit{chi} \\
\quad]^{0\ldots\infty}
\end{array}
$$

$\mathrm{MaximalSpec}$ specifies the largest possible set of interaction paths for a system with interface (ρ, χ). In particular it has the property of being *maximal* with respect to the $\overset{i}{\subseteq}$ relation:

$$\langle D\rangle^{\rho}_{\chi} \overset{i}{\subseteq} \langle \mathrm{MaximalSpec}(\rho, \chi)\rangle^{\rho}_{\chi}$$

for any specification diagram $\langle D\rangle^{\rho}_{\chi}$. The specification receives arbitrary data, and sends arbitrary messages to its acquaintances. To be truly general in this regard, it needs to create arbitrarily (finitely) many new names for possible insertion into each message it sends out; that is the purpose of the $\texttt{fresh}(q)$ loop above. The value of $\texttt{pick}(x)$ will have acquaintances that may include these fresh names. The diagram variables rho and chi serve to keep explicit track of the evolving ρ and χ sets. If a new name is received in a message, it is added to ext and then may later be the target of a message. $\mathrm{MaximalSpec}$ is the analog of the CSP process Chaos_A which can participate in any action (over its alphabet A) and can also refuse any action. Chaos_A is refined, in the CSP sense, by every process with alphabet A.

At the other extreme of the refinement relation, is the following diagram.

$$\mathrm{MinimalSpec} = \{\texttt{constrain}(\mathit{false})\}$$

For any receptionists, ρ, and externals χ, the top-level diagram

$$\langle \mathrm{MinimalSpec}\rangle^{\rho}_{\chi}$$

has no admissible interaction paths as all paths will be aborted by the failed constraint. The empty set of interaction paths can also be expressed without appealing to `constrain`, by instead requiring that a message be received that can not possibly be delivered.

$$\mathrm{MinimalSpec} = \{\texttt{fresh}(x); \texttt{receive}(x \triangleleft \texttt{nil})\}$$

In particular, $\mathrm{MinimalSpec}$, just above must receive a message for a newly created actor whose name is not known to any possible sender. Thus top-level specification $\langle \mathrm{MinimalSpec}\rangle^{\rho}_{\chi}$ is minimal with respect to the refinement relation

$$\langle \mathrm{MinimalSpec} \rangle^{\rho}_{\chi} \overset{i}{\subseteq} \langle D \rangle^{\rho}_{\chi}$$

There are no "empty" processes in CSP. $\mathrm{MinimalSpec}$ can be thought of as an inconsistent specification, emphasizing the logical or specification aspect of **SD**'s. The minimal CSP process, $\mathrm{STOP_A}$, has a single trace, namely the empty trace, and can refuse any subset of A. This process corresponds to the top-level specification diagram $\langle \mathtt{eod} \rangle^{\rho}_{\chi}$ which also has a single admissible interaction path, the empty path with interface (ρ, χ), and we have

$$\langle \mathrm{MinimalSpec} \rangle^{\rho}_{\chi} \overset{i}{\subseteq} \langle \mathtt{eod} \rangle^{\rho}_{\chi}$$

but it is not the case that

$$\langle \mathtt{eod} \rangle^{\rho}_{\chi} \overset{i}{\subseteq} \langle D \rangle^{\rho}_{\chi}$$

for any top-level diagram $\langle D \rangle^{\rho}_{\chi}$. For example, if $D = \{\mathtt{receive}(a \triangleleft \mathtt{nil})\}$ with $a \in \rho$, then the empty interaction path is not an admissible path for $\langle D \rangle^{\rho}_{\chi}$ since D specifies that exactly one message be received. The difference here is that interaction semantics is based on complete fair computations rather than just the finite computations.

3.2.4. *Input Restriction*

We often want to consider only the interaction paths of a component in which inputs from the environment are restricted to a subset of the messages under consideration. For example, we are only interested in inputs of `compute` requests to the function composer. The other messages are intended for internal communication only.

Definition 3.2 (Input restriction): Let $\mathbf{MP}_a$ be a subset of $\mathbf{MP}$ then $\langle D \rangle^{\rho}_{\chi} \upharpoonright \mathbf{MP}_a$ is the restriction of the interactions of top-level diagram $\langle D \rangle^{\rho}_{\chi}$ to inputs in $\mathbf{MP}_a$. In particular

$$[\![\langle D \rangle^{\rho}_{\chi} \upharpoonright \mathbf{MP}_a]\!] = [\![\langle D \rangle^{\rho}_{\chi}]\!] \upharpoonright \mathbf{MP}_a$$

where for any set P of interaction paths, $P \upharpoonright \mathbf{MP}_a$ is the subset of interaction paths in P such that every input interaction, $\mathtt{in}(mp)$, has $mp \in \mathbf{MP}_a$.

3.2.5. *Input Restriction for the Function Composer*

For example, let $\mathbf{MP}_{\mathtt{compute}}$ be the subset of $\mathbf{MP}$ with contents of the form $\mathtt{compute}@c$ and target a. Then the interaction paths of $\langle \mathrm{C}(a, f, g, af, ag) \rangle^{a}_{\emptyset} \upharpoonright \mathbf{MP}_{\mathtt{compute}}$ are those in which only `compute` requests are accepted from the environment.

3.2.6. *Interaction equivalence*

When two specifications each refine the other they are said to be *interaction equivalent*. This means that their observable behaviors are identical from the

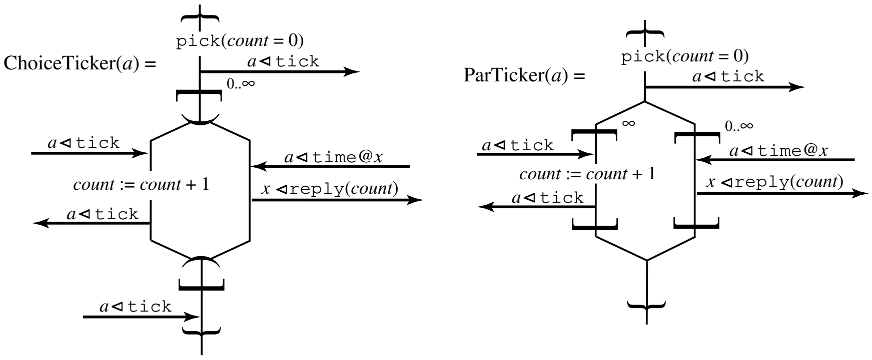

Figure 7. Alternate Codings: Choice Ticker and Parallel Ticker Specifications

point of view of the environment. For example if an implementation fully and faithfully satisfies a specification, the two must have the same sets of interaction paths, and thus they are interaction equivalent.

Definition 3.3 (SD interaction equivalence):

$$\langle D_I \rangle^{\rho}_{\chi} \stackrel{i}{\simeq} \langle D_S \rangle^{\rho}_{\chi} \quad \text{iff} \quad [\![\langle D_I \rangle^{\rho}_{\chi}]\!] = [\![\langle D_S \rangle^{\rho}_{\chi}]\!].$$

3.2.7. *Interaction Equivalence for the Function Composer*

A good example of interaction equivalence is found in the context of the function composer example above. A high-level specification for computing $g \circ f$ is just $\mathrm{F}(a, g \circ f)$ which directly computes $g \circ f$. We may then assert the following.

Theorem 3.4:

$$\langle \mathrm{C}(a, f, g, af, ag) \rangle^{a}_{\emptyset} \stackrel{i}{\simeq} \langle \mathrm{F}(a, g \circ f) \rangle^{a}_{\emptyset}$$

This theorem will be proved in Section 5.3. The proof illustrates the use of semantics-based techniques for reasoning about **SD**'s.

3.2.8. *Interaction equivalence and the Ticker*

A more low-level specification of a ticker might use `tick` messages to update its counter: every time it receives a `tick`, it increments its counter and sends itself another `tick`. (also see Figure 7):

$$\mathrm{ChoiceTicker}(a) =$$

$$\{\, \texttt{pick}(count = 0);\ \texttt{send}(a \triangleleft \texttt{tick});$$
$$[\ \texttt{receive}(a \triangleleft \texttt{tick});\ count := count + 1;\ \texttt{send}(a \triangleleft \texttt{tick})$$
$$\oplus$$
$$\texttt{receive}(a \triangleleft \texttt{time}\ @\ x);\ \texttt{send}(x \triangleleft \texttt{reply}(count))\,]^{0\ldots\infty};$$
$$\texttt{receive}(a \triangleleft \texttt{tick})$$
$$\texttt{(* eat final tick if finite iteration *)}\ \}$$

Another reasonable alternative to express this same idea is $\mathrm{ParTicker}$, which uses parallelism between the `tick` and `time` instead of nondeterministic choice (also shown in Figure 7):

$$\mathrm{ParTicker}(a) =$$
$$\texttt{pick}(count = 0);\ \texttt{send}(a \triangleleft \texttt{tick});$$
$$[\texttt{receive}(a \triangleleft \texttt{tick});\ count := count + 1;\ \texttt{send}(a \triangleleft \texttt{tick})]^{\infty}$$
$$|$$
$$[\texttt{receive}(a \triangleleft \texttt{time}\ @\ x);\ \texttt{send}(x \triangleleft \texttt{reply}(count))]^{0\ldots\infty}$$

No final $\texttt{receive}(a \triangleleft \texttt{tick})$ is needed in the case of $\mathrm{ParTicker}(a)$ since the `tick` receiving thread does not terminate, even if the `time` receiving thread does. The Ticker, $\mathrm{ChoiceTicker}$, and $\mathrm{ParTicker}$ are equivalent in a context where actor a receives only `time` messages from the environment. For this purpose we again use the restriction operator to restrict interactions to relevant paths. Let $\mathbf{MP}_{\texttt{time}} = \{a \triangleleft \texttt{time}\ @\ c \mid c \in \mathbf{A} \text{ and } c \neq a\}$. Thus the interactions of $\langle \mathrm{ChoiceTicker}(a)\rangle^a_\emptyset \restriction \mathbf{MP}_{\texttt{time}}$ are those in which only `time` messages are received from the environment.

Theorem 3.5:

$$\langle \mathrm{ChoiceTicker}(a)\rangle^a_\emptyset \restriction \mathbf{MP}_{\texttt{time}} \overset{i}{\simeq}$$
$$\langle \mathrm{ParTicker}(a)\rangle^a_\emptyset \restriction \mathbf{MP}_{\texttt{time}} \overset{i}{\simeq}$$
$$\langle \mathrm{Ticker}(a)\rangle^a_\emptyset .$$

Note that $\langle \mathrm{Ticker}(a)\rangle^a_\emptyset \restriction \mathbf{MP}_{\texttt{time}} \overset{i}{\simeq} \langle \mathrm{Ticker}(a)\rangle^a_\emptyset$, since $\mathrm{Ticker}(a)$ does not accept `tick` messages. These interaction equivalence relations are established in Section 5.2 below.

3.3. Asserting Properties of Specifications Diagrammatically

Safety and liveness properties can be asserted directly in the **SD** language. We present three different techniques for asserting safety and liveness. The first method is based on interaction refinement. The second is based on diagrammatically defining an environment which requires the specification to have the proper behavior. The third method is by directly decorating the specification with logical assertions.

The underlying idea of the first method is that a top-level **SD** can be thought of as defining a predicate on interaction paths (such as a liveness condition). The assertion that all behaviors of one top-level diagram $\langle D' \rangle^{\rho}_{\chi}$ have the property defined by another $\langle D \rangle^{\rho}_{\chi}$ is then just stating that $\langle D' \rangle^{\rho}_{\chi} \overset{i}{\subseteq} \langle D \rangle^{\rho}_{\chi}$.

An example of this method was in fact given earlier: the property that all `time` messages sent to the Ticker will receive a reply was expressed by the $\mathrm{LiveTicker}$ specification, and the statement $\langle \mathrm{Ticker}(a) \rangle^{a}_{\emptyset} \overset{i}{\subseteq} \langle \mathrm{LiveTicker}(a) \rangle^{a}_{\emptyset}$ asserts the Ticker has such a property. $\mathrm{LiveTicker}$ actually states a stronger property than liveness – it asserts a bijection between receives and sends. It is fairly easy to show directly that the Ticker has the liveness property; but, this technique will also work for more complex behaviors.

A second and perhaps more convincing way to assert liveness is by specifying an environment which requires liveness. For the Ticker, the environment should assert that all `time` requests are handled. The `assert` predicate is used for this purpose (more precisely the `fail` macro, defined as $\texttt{assert}(\mathit{false})$) in the following $\mathrm{LiveTickerEnvt}$:

$$\mathrm{LiveTickerEnvt}(a) =$$

$$\{\ [\texttt{fresh}(c); \texttt{send}(a \triangleleft \texttt{time} \mathbin{@} c); (\texttt{receive}(c \triangleleft x) \oplus (\texttt{fail};\ \texttt{eod}))]^{0\ldots\infty}\ \}$$

Failure arises only when there is a `time` request that does not get answered – the `receive` choice is never possible and so failure is the only possibility. If failure were chosen when there was in fact a reply to be received, that path would be unfair because a message was never received. The `eod` is needed in the failure branch of the choice to ensure that if failure happens it is the end of the computation. Otherwise the failure branch could be taken randomly and pending messages could be received later. Asserting

$$\mathrm{OK}(\langle \mathrm{Ticker}(a) \mid \mathrm{LiveTickerEnvt}(a) \rangle^{\emptyset}_{\emptyset})$$

says that there are no paths of this combined system which contain a `fail` event; see Definition 4.12 in the next section for the formal definition of $\mathrm{OK}(\cdot)$. Although this is an assertion about the possible computation paths, it implies the desired responsiveness assertion. $\mathrm{LiveTickerEnvt}$ is an example of a general form of diagram that can be used to specify responsiveness to requests and to reduce the work of establishing such a property to reasoning directly about possible computations.

The third manner in which properties may be diagrammatically asserted is that safety properties may be directly asserted in the specification itself via `assert` decorations. An example of a specification decorated with a safety assertion is $\mathrm{SafeTicker}(a)$ (shown in Figure 8), a ticker which asserts successive outputs are non-decreasing.

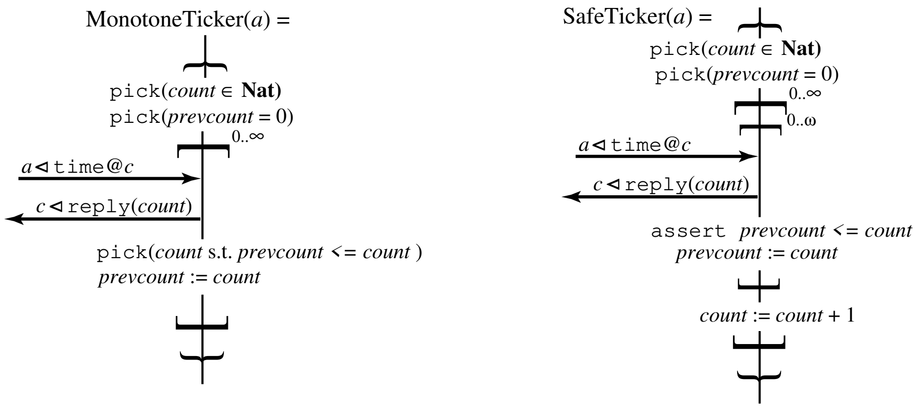

Figure 8. Examples of expressing assertions via diagrams: the Monotone and Safe Ticker specifications

$$\begin{array}{l} \mathrm{SafeTicker}(a) = \\ \quad \{\ \mathtt{pick}(\mathit{count} \in \mathbf{Nat});\ \mathtt{pick}(\mathit{prevcount} = 0); \\ \quad [\ [\ \mathtt{receive}(a \triangleleft \mathtt{time} \mathbin{@} c);\ \mathtt{send}(c \triangleleft \mathtt{reply}(\mathit{count})); \\ \quad\ \ \mathtt{assert}(\mathit{prevcount} \leq \mathit{count});\ \mathit{prevcount} := \mathit{count}\ ]^{0\ldots\omega}; \\ \quad \mathit{count} := \mathit{count} + 1\ ]^{0\ldots\infty}\ \} \end{array}$$

The safety assertion for $\mathrm{SafeTicker}$

$$\mathrm{OK}(\langle \mathrm{SafeTicker}(a) \rangle^{a}_{\emptyset}).$$

says that the sequence of values of *count* sent in replies is non-decreasing. We can combine this with method 1 to assert a corresponding property for Ticker

$$\langle \mathrm{Ticker}(a) \rangle^{a}_{\emptyset} \ \stackrel{i}{\subseteq}\ \langle \mathrm{SafeTicker}(a) \rangle^{a}_{\emptyset}$$

In fact

$$\langle \mathrm{Ticker}(a) \rangle^{a}_{\emptyset} \ \stackrel{i}{\simeq}\ \langle \mathrm{SafeTicker}(a) \rangle^{a}_{\emptyset}$$

This follows from a general result that an **SD** can be decorated with new diagram variables, assignments to these variables, and assertions without changing the interaction path semantics.

Safety assertions can also be expressed directly combining the use of additional diagram variables and the diagram as predicate interpretation (method 1). As an example consider the $\mathrm{MonotoneTicker}$ specification, also shown in Figure 8:

$$\mathrm{MonotoneTicker}(a) =$$

$$\{\ \mathtt{pick}(count \in \mathbf{Nat}); \mathtt{pick}(prevcount = count);$$
$$[\ \mathtt{receive}(a \triangleleft \mathtt{time} \mathbin{@} c);\ \mathtt{send}(c \triangleleft count);$$
$$\mathtt{pick}(count\ \mathbf{s.t.}\ prevcount \leq count);\ prevcount := count\ ]^{0 \ldots \infty}\ \}$$

and the assertion

$$\langle \mathrm{Ticker}(a) \rangle^{a}_{\emptyset} \ \overset{i}{\subseteq}\ \langle \mathrm{MonotoneTicker}(a) \rangle^{a}_{\emptyset}$$

which analogously to SafeTicker is asserting that `time` replies are monotone. This monotonicity condition could also have been asserted by an environment in the manner of LiveTickerEnvt above. Some properties may not be easy to assert diagrammatically, but many common forms of assertion can be naturally expressed in this manner. Although the language is not trivial to learn, we believe for practitioners its operational basis and graphical syntax will make it easier to learn than e.g. temporal logic, and thus encourage increased use of formal methods. Decision procedures and formal proof systems exist for many temporal logics. Tools do need to be be developed for checking **SD** properties in restricted but common cases, or for automatically inserting runtime checks when an assertion can not be established by the tools at hand. Such tools are an important topic for future work.

4. Semantics of Diagrams

In this section we study the semantics of specification diagrams. We first define an operational semantics, then use this to determine the set of interaction paths $[\![\langle D \rangle^{\rho}_{\chi}]\!]$ defining the behavior of top-level diagrams $\langle D \rangle^{\rho}_{\chi}$.

The operational semantics of a top-level **SD** is the set of allowed computation paths described by the diagram. A computation path is an infinite sequence of internal steps interleaved with interactions with the environment. Internal steps traverse the diagram, modifying the diagram state, sending and receiving messages using a local message pool, forking new threads, creating new names, evaluating constraints, *etc*. Interactions with the environment allow messages to receptionists to come into the system and messages to external actors to be emitted from the system. Allowed computations must obey certain constraints on message delivery and give fair treatment to independent threads of activity. A more abstract (denotational) semantics, *the interaction path semantics* is derived from the allowed computation paths semantics by observing only the interactions with the environment.

The computation paths of a component are given by a labelled transition relation on configurations derived from local reaction rules. A configuration represents a snapshot of an executing component. The transition labels corresponding to local reaction rules give us an operational view of the internal workings of a component. They are useful for defining notions of fairness and for reasoning about syntactic transformations. Interaction semantics hides these internal labels, allowing only the interactions with the environment to be observed. In fact, what we will define is an actor theory [40, 42] semantics for **SD**'s. Thus we shall call the transition system the *specification diagram actor theory*. We will fill in enough detail that this paper can be read independently of the above references.

4.1. PRELIMINARIES

We begin with some mathematical preliminaries. To simplify computation paths we consider diagram expressions $D; \texttt{skip}$, $\texttt{skip}; D$ and D to denote the same diagram. This is consistent with the graphical form, since sequential composition of a diagram with a straight line does not change the graph. Also, we will equate $\texttt{skip} \mid \texttt{skip}$ and $\texttt{skip}$. To define the local rules, we use the approach of [19] and factor a (runtime) diagram D into a redex D_{rdx} and reduction context R such that $D = R[D_{\mathrm{rdx}}]$ identifying D_{rdx} as the next redex to be reduced. Notation is also needed for looking up, modifying, and extending variable environments. The concepts of reduction context and environment are in fact intertwined: environments are local to particular subdiagrams (so e.g. parallel threads may have differing environments) and so are spread around the reduction context. As discussed in section 2, these local environments are finite functions $\gamma \in \mathbf{X}_{\mathrm{d}} \xrightarrow{\omega} \mathbf{U}$ which hold the current state of diagram variables $\mathbf{X}_{\mathrm{d}}$. Extension/updating, $\gamma_0; \gamma_1$, is defined as usual. $\gamma_0; \gamma_1$ is the finite function γ such that $\mathrm{Dom}(\gamma) = \mathrm{Dom}(\gamma_0) \cup \mathrm{Dom}(\gamma_1)$ and for $x \in \mathrm{Dom}(\gamma)$,

$$\gamma(x) = \textbf{if } x \in \mathrm{Dom}(\gamma_1) \textbf{ then } \gamma_1(x) \textbf{ else } \gamma_0(x).$$

$\gamma\{x \mapsto u\}$ abbreviates $\gamma; \{x \mapsto u\}$.

We extend the language syntax to include syntax for binding variables inside an executing diagram: $\{\!|\gamma : D|\!\}$ indicates a lexical scoping construct $\{D\}$ under which execution is actively occurring, with current local environment γ.

The grammar of reduction contexts is

$$\begin{aligned} R = \ & \bullet + \\ & R \mid D + \\ & D \mid R + \\ & R; D + \\ & \{\!|\gamma : R|\!\} \end{aligned}$$

$\bullet$ is called a hole, and is a place holder for a redex. Each reduction context has a unique occurrence of $\bullet$ and $R[D]$ denotes the act of replacing that occurrence in R with diagram D. Note that, unlike the case for traditional sequential languages, the decomposition of a diagram is not necessarily unique, since a parallel construct typically has two decompositions, allowing the redex to be chosen from either part.

Notation is next defined for manipulation of the environment embedded in a reduction context. The basic operations needed include $R\,@\,x$ to look up the value of x in the environments of R, and $R\{x \mapsto u\}$ to modify the value of an already-declared variable x. $\epsilon(R)$ extracts the environment implicit in R.

Definition 4.1 ($\epsilon(R)$, $R\,@\,x$, $R\{x \mapsto u\}$):

$$
\begin{aligned}
&\epsilon(\bullet) = \bar{\emptyset} \ \text{ the empty finite function} \\
&\epsilon(R \mid D) = \epsilon(D \mid R) = \epsilon(R; D) = \epsilon(R) \\
&\epsilon(\{\!|\gamma : R|\!\}) = \gamma; \epsilon(R) \\
&R\,@\,x = \epsilon(R)(x) \text{ if } x \in \mathrm{Dom}(\epsilon(R)) \\
&\bullet\{x \mapsto u\} = \bullet \\
&(R \mid D)\{x \mapsto u\} = (D \mid R)\{x \mapsto u\} = (R; D)\{x \mapsto u\} = R\{x \mapsto u\} \\
&(\{\!|\gamma : R|\!\})\{x \mapsto u\} \\
&\quad = \ \texttt{if}\ x \in \mathrm{Dom}(\epsilon(R))\ \texttt{then}\ R\{x \mapsto u\}\ \texttt{else}\ \{\!|\gamma\{x \mapsto u\} : R|\!\}
\end{aligned}
$$

We use the usual convention for multiple assignment: if $\overline{x}$ is a list of distinct diagram variables and $\overline{u}$ is a list of values of the same length, then $\{\overline{x} \mapsto \overline{u}\}$ is the environment mapping each variable to its corresponding value (order of assignment does not matter here).

The value of an expression u_{d} occurring in a redex situated in a reduction context R is obtained by applying the expression to the environment associated with R: $u_{\mathrm{d}}(\epsilon(R))$.

We want to ensure that the actor acquaintance laws are obeyed: an actor can only know the name of another actor if it was told that name at creation time (hence the creator must have known the name) or if it received the name in a message, or if it created the actor with that name. To determine the actor names occurring in messages or other values, we assume given an acquaintance function $acq : \mathbf{U} \cup \mathbf{U}_{\mathrm{d}} \rightarrow \mathbf{P}_{\omega}(\mathbf{A})$ and lift this function homomorphically to environments and diagrams, using the fact that state and recursion variables have no acquaintances. We require that evaluation not generate any new acquaintances: $acq(u_{\mathrm{d}}(\gamma)) \subseteq acq(u_{\mathrm{d}}) \cup acq(\gamma)$.

4.2. Operational and Interaction Semantics

Now we define the specification diagram actor theory $SDTh$, that is, the transition system and the admissibility requirements that determine the allowed computation paths of a top level diagram. The states of the transition system are called *configurations*. A configuration has an interface and an internal part. The internal part contains the execution state of the system and the pool (multiset) of pending message packets. The transition rules are derived from local reaction rules by lifting them to configurations and adding rules for interaction with the environment. We first define the execution states, and local reaction rules. Then we define configurations, the transition relation on configurations, and the admissible computation paths.

Definition 4.2 ($SDTh$ states and reaction rules): The execution states of $SDTh$ are diagrams in the extended syntax. The labelled reaction rules for $SDTh$ are given in Figure 9. The left-hand side of each rule has the form $R[D_{\mathrm{rdx}}]$ where R is a reduction context and D_{rdx} is the *redex*. In the `send` and `receive` rules we use the convention that messages received appear above the arrow and messages send appear below the arrow.

Definition 4.3 (Configurations): A configuration has the form $K = I\,{}^{\rho}_{\chi}$ where (ρ, χ) is the interface and I is the interior. An interior has the form $I = D \cdot \mu$ where D is a diagram execution state as above, and μ is a multiset of message packets, and we define $I \cdot \mu'$ to be $D \cdot \mu \cdot \mu'$ if $I = D \cdot \mu$. Thus, expanding definitions, $K = \langle\, D \cdot \mu \,\rangle\,{}^{\rho}_{\chi}$.

The computation paths of a configuration are given by a labelled transition relation with elements of the form $K \xrightarrow{tl} K'$. The transition label tl is either an internal label, an input label, an output label, or the idle label, `idle`. The `idle` label indicates absence of action, either because no rules apply, or simply to allow "time" to pass. An internal label has the form $l(\mu_r)$ where l is a rule label, and μ_r is the sent message packet, if any. An input label has the form $\mathtt{in}(a \triangleleft M)$, indicating a message is arriving from the environment. An output label has the form $\mathtt{out}(a \triangleleft M)$ indicating a message is being transmitted to the environment. [1]

Definition 4.4 (Transition rules and Computation Paths): The transition rules for configurations are the following.

(r) $\quad \langle\, D_0 \cdot \mu_r \cdot \mu \,\rangle\,{}^{\rho}_{\chi} \xrightarrow{l(\mu_r)} \langle\, D_1 \cdot \mu_s \cdot \mu \,\rangle\,{}^{\rho}_{\chi}$

if $l : D_0 \xrightarrow[\mu_s]{\mu_r} D_1$ is a diagram rule, and

[1] To simplify the presentation we have suppressed bookkeeping information in execution states and transition labels which are used to avoid misuses of actor names and to make precise the conditions for admisability.

$\mathtt{choose}(l)$: $\quad R[D_l \oplus D_r] \;\longrightarrow\; R[D_l]$ $\qquad$ similarly for $\mathtt{choose}(r)$

$\mathtt{fork}$: $\quad R[\mathtt{fork}(D)] \;\longrightarrow\; (R[\mathtt{skip}] \mid \{\!|\epsilon(R) : D|\!\})$

$\mathtt{receive}$: $\quad R[\mathtt{receive}(a_\mathrm{d} \triangleleft M_\mathrm{d})] \;\xrightarrow{mp}\; R'[\mathtt{skip}]$

if $\overline{x}$ are the diagram variables occurring in M_d,

$\overline{u}$ are values such that $mp = a_\mathrm{d}(\epsilon(R)) \triangleleft M_\mathrm{d}(\{\overline{x} \mapsto \overline{u}\})$,

and $R' = R(\{\overline{x} \mapsto \overline{u}\})$

$\mathtt{send}$: $\quad R[\mathtt{send}(a_\mathrm{d} \triangleleft M_\mathrm{d})] \;\underset{mp}{\longrightarrow}\; R[\mathtt{skip}]$

where $mp = a_\mathrm{d}(\epsilon(R)) \triangleleft M_\mathrm{d}(\epsilon(R))$

$\mathtt{pick}$: $\quad R[\mathtt{pick}(x)] \;\longrightarrow\; (R\{x \mapsto u\})[\mathtt{skip}]$ $\qquad$ if $acq(u) \subseteq acq(R)$

$\mathtt{fresh}(a)$: $\quad R[\mathtt{fresh}(x)] \;\longrightarrow\; (R\{x \mapsto a\})[\mathtt{skip}]$ $\qquad$ if $a \notin acq(R)$

$\mathtt{assert}(b)$: $\quad R[\mathtt{assert}(\phi)] \;\longrightarrow\; R[\mathtt{skip}]$ $\qquad$ if $b = \phi(\epsilon(R)) \in \mathbf{Bool}$

$\mathtt{constrain}$: $\quad R[\mathtt{constrain}(\phi)] \;\longrightarrow\; R[\mathtt{skip}]$

if $\phi(\epsilon(R)) = true$

$R[\mathtt{constrain}(\phi)] \;\longrightarrow\; R[\mathtt{receive}(a \triangleleft \mathtt{nil})]$

if $\phi(\epsilon(R)) \neq true$ and $a \notin acq(R)$

$\mathtt{assign}$: $\quad R[x := \psi] \;\longrightarrow\; R'[\mathtt{skip}]$

if $R' = R\{x \mapsto \psi(\epsilon(R))\}$

$\mathtt{rec}$: $\quad R[\mathtt{rec}\,X.D] \;\longrightarrow\; R[D[(\mathtt{rec}\,X.D)/X]]$

$\mathtt{scope\text{-}in}$: $\quad R[\{x_1, \ldots, x_n : D\}] \;\longrightarrow\; R[\{\!|\gamma : D|\!\}]$

$\mathrm{Dom}(\gamma) = \{x_1, \ldots, x_n\}$ and

$acq(\gamma(x_i)) \subseteq acq(R) \cup acq(D)$ for $1 \leq i \leq n$

$\mathtt{scope\text{-}out}$: $\quad R[\{\!|\gamma : \mathtt{skip}|\!\}] \;\longrightarrow\; R[\mathtt{skip}]$

Figure 9. Reaction Rules for Diagrams

any actor names generated by the diagram rule are fresh

(in) $\quad I^{\rho}_{\chi} \xrightarrow{\mathtt{in}(a \triangleleft M)} \langle\, (I \cdot a \triangleleft M)\,\rangle^{\rho}_{\chi \cup (acq(M) - \rho)}$

if $a \in \rho$ and $acq(M) \cap (acq(I) - \chi) \subseteq \rho$

(out) $\quad \langle\, (I \cdot a \triangleleft M)\,\rangle^{\rho}_{\chi} \xrightarrow{\mathtt{out}(a \triangleleft M)} I^{\rho \cup (acq(M) - \chi)}_{\chi}$ if $a \in \chi$

(idle) $\quad I^{\rho}_{\chi} \xrightarrow{\mathtt{idle}} I^{\rho}_{\chi}$

Computation paths $\pi \in \mathcal{P}$ are infinite sequences of the form

$$\pi = [\langle D_i \cdot \mu_i \rangle^{\rho_i}_{\chi_i} \xrightarrow{tl_i} \langle D_{i+1} \cdot \mu_{i+1} \rangle^{\rho_{i+i}}_{\chi_{i+1}} \mid i \in \mathbf{Nat}]$$

where each element is a transition rule instance. Note that considering only infinite sequences is not a restriction, since a computation that terminates after finitely many steps can be made infinite by use of the `idle` transition.

Example 4.5 (An Example Computation): We now illustrate the rules via an example computation. Recall the function composer diagram FC of Section 3. We will show one complete pass through the loop. For this purpose we define additional diagrams that correspond to positions in the unfolding of the initial state diagram using the environment γ to specify currently bound variables. Let γ contain the bindings $\{x \mapsto v, y \mapsto w, z \mapsto u, xc \mapsto c, xf \mapsto cf, xg \mapsto cg\}$ and we define

$$\begin{array}{l}
\mathrm{FC0}(a, af, ag, \gamma) = \{\!|\ \gamma : \mathtt{rec}(X) \\
\quad (\ [\ \mathtt{receive}(a \triangleleft \mathtt{compute}(x)); \mathtt{fresh}(xf); \mathtt{send}(af \triangleleft \mathtt{compute}(x)\ @\ xf); \\
\qquad \mathtt{receive}(xf \triangleleft \mathtt{reply}(y)); \mathtt{fresh}(xg); \mathtt{send}(ag \triangleleft \mathtt{compute}(y)\ @\ xg); \\
\qquad \mathtt{receive}(xg \triangleleft \mathtt{reply}(z)); \mathtt{send}(xc \triangleleft \mathtt{reply}(z)); X\ ] \\
\quad \oplus \\
\qquad \mathtt{skip}\ |\!\})
\end{array}$$

(recall $\mathtt{[}D\mathtt{]}^{0\ldots\infty}$ abbreviates $\mathtt{rec}\,X.((D;X) \oplus \mathtt{skip})$)

$$\begin{array}{l}
\mathrm{FC1}(a, af, ag, \gamma) = \\
\quad \{\!|\ \gamma :\ \mathtt{receive}(xf \triangleleft \mathtt{reply}(y)); \mathtt{fresh}(xg); \mathtt{send}(ag \triangleleft \mathtt{compute}(y) @ xg); \\
\qquad\quad \mathtt{receive}(xg \triangleleft \mathtt{reply}(z)); \mathtt{send}(xc \triangleleft \mathtt{reply}(z)); \mathrm{FC0}^-(a, af, ag)\ |\!\}
\end{array}$$

$$\begin{array}{l}
\mathrm{FC2}(a, af, ag, \gamma) = \\
\quad \{\!|\ \gamma :\ \mathtt{receive}(xg \triangleleft \mathtt{reply}(z)); \mathtt{send}(xc \triangleleft \mathtt{reply}(z)); \mathrm{FC0}^-(a, af, ag)\ |\!\}
\end{array}$$

where $\mathrm{FC0}^-$ is FC0 with the outermost scoping construct, $\{\!|\gamma : \ldots |\!\}$, removed

Using the reaction rules we obtain the following example computation path in which there is a single `compute` request from the environment and γ_0 contains some (arbitrary) initial bindings for the diagram variables.

$$\begin{array}{l}
\langle \mathrm{FC}(a, af, ag) \rangle^{a}_{af,ag} \\
\qquad \xrightarrow{\mathtt{scope\text{-}in}} \\
\quad \langle\ \mathrm{FC0}(a, af, ag, \gamma_0)\ \rangle^{a}_{af,ag} \\
\qquad \xrightarrow{\mathtt{in}(a \triangleleft \mathtt{compute}(v) @ c)}
\end{array}$$

$$\langle\, \mathrm{FC0}(a, af, ag, \gamma_0) \cdot a \triangleleft \mathtt{compute}(v) \;@\; c \,\rangle\, ^{a}_{af,ag,c}$$

$$\xrightarrow{\mathtt{rec};\mathtt{choose}(l);\mathtt{receive}(a\triangleleft\mathtt{compute}(v)@c);\mathtt{fresh}(cf);\mathtt{send}(af\triangleleft\mathtt{compute}(v)@cf)}$$

$$\langle\, \mathrm{FC1}(a, af, ag, \gamma_1) \cdot af \triangleleft \mathtt{compute}(v)@cf \,\rangle\, ^{a}_{af,ag,c}$$

where $\gamma_1 = \gamma_0\{x \mapsto v, xc \mapsto c, xf \mapsto cf\}$

$$\xrightarrow{\mathtt{out}(af\triangleleft\mathtt{compute}(v)@cf)} \quad \xrightarrow{\mathtt{in}(cf\triangleleft\mathtt{reply}(w))}$$

$$\langle\, \mathrm{FC1}(a, af, ag, \gamma_1) \cdot cf \triangleleft \mathtt{reply}(w) \,\rangle\, ^{a,cf}_{af,ag,c}$$

$$\xrightarrow{\mathtt{receive}(cf\triangleleft\mathtt{reply}(w));\mathtt{fresh}(cf);\mathtt{send}(ag\triangleleft\mathtt{compute}(w)@cg)}$$

$$\langle\, \mathrm{FC2}(a, af, ag, \gamma_2) \cdot ag \triangleleft \mathtt{compute}(w)@cg \,\rangle\, ^{a,cf}_{af,ag,c}$$

where $\gamma_2 = \gamma_1\{y \mapsto w, xg \mapsto cg\}$

$$\xrightarrow{\mathtt{out}(ag\triangleleft\mathtt{compute}(w)@cg)} \quad \xrightarrow{\mathtt{in}(cg\triangleleft\mathtt{reply}(u))}$$

$$\langle\, \mathrm{FC2}(a, af, ag, \gamma_2) \cdot cg \triangleleft \mathtt{reply}(u) \,\rangle\, ^{a,cg,cf}_{af,ag,c}$$

$$\xrightarrow{\mathtt{receive}(cg\triangleleft\mathtt{reply}(u));\mathtt{send}(c\triangleleft\mathtt{reply}(u))}$$

$$\langle\, \mathrm{FC0}(a, af, ag, \gamma_2) \cdot c \triangleleft \mathtt{reply}(u) \,\rangle\, ^{a,cf,cg}_{af,ag,c}$$

$$\xrightarrow{\mathtt{out}(c\triangleleft\mathtt{reply}(u))}$$

$$\langle\, \mathrm{FC0}(a, af, ag, \gamma_2) \,\rangle\, ^{a,cf,cg}_{af,ag,c}$$

Given the definitions of the states, reaction rules, and certain additional admissibility information, the actor theory framework tells us which computation paths are admissible. Rather than introduce the general definition, we state a lemma that gives an equivalent characterization of admissible computation paths for the **SD** actor theory. To ensure that the acquaintance laws are not violated, freshness constraints on actor names must be interpreted in terms of the computation path up to the point of rule application. This is because some names that have been used may not occur in the current configuration. In addition, two fairness properties are needed, one for redexes and one for messages. The basic idea is that a redex occurring in a reduction context hole is considered enabled, and if the computation path is admissible an enabled redex occurrence must eventually be reduced. Furthermore every message that appears in the internal message set (generated by an internal send or an input) must be received.

Definition 4.6 (Fairness properties):
Redex-fairness: a path, π, is *redex-fair* if for all configurations of the form $\langle D_i \cdot \mu_i \rangle\, ^{\rho_i}_{\chi_i}$ arising in π, if $D_i = R[D_{\mathrm{rdx}}]$ for some R, D_{rdx}, then there is

a later transition in which the redex reduced in this transition is the same subterm occurrence D_{rdx}.

Message-fairness: a path, π, is *message-fair* if for all configurations of the form $\langle D_i \cdot \mu_i \rangle^{\rho_i}_{\chi_i}$ arising in π, each packet $mp \in \mu_i$ must be delivered in some later transition, either by being output (if the target is external) or by application of the `receive` (if the target is internal).

Lemma 4.7 (SD admissibility): The admissible paths of $SDTh$ are the computation paths which are both *redex-fair* and *message-fair*. We let $\mathcal{A}(K)$ denote the admissible paths with initial configuration K.

Definition 4.8 (Interaction Path): An interaction path consists of an interface and a sequence (finite or infinite) of interactions $io \in (\mathtt{in}(\mathbf{MP}) \cup \mathtt{out}(\mathbf{MP}))$ satisfying the interaction path laws. There are two main laws. The first says that inputs must be initial receptionists or actors that have become receptionists by having their name exported in previous outputs. The second says that outputs must be to actors that are initially in the external set or whose names have been imported in previous inputs. For technical convenience we allow gaps in the sequence (i.e. the sequence is a partial function from $\mathbf{Nat}$ to interactions allowing for times that no interaction occurs) and sometimes represent the partial function as a set of pairs (i, io) for i in the function domain.

The interaction path semantics of a configuration is the set of interaction paths obtained by hiding the internal details of admissible computation paths. Formally we define a function $cp2ip$ mapping computation paths to interaction paths: $cp2ip(\pi)$ has the same interface as the interface of the initial configuration of π, its interaction sequence is given by the set of pairs of the form (i, io) such that the ith transition has an interaction label, io.

Definition 4.9 ($cp2ip(\pi)$): For a computation π with initial configuration $I_0{}^{\rho_0}_{\chi_0}$, the associated interaction path is defined by

$$cp2ip(\pi) = \varepsilon^{\rho_0}_{\chi_0} \quad \text{where}$$

$$\varepsilon = \{(i, io) \mid i \in \mathbf{Nat} \wedge \pi(i) = K_i \xrightarrow{io} K_{i+1} \wedge io \in (\mathtt{in}(\mathbf{MP}) \cup \mathtt{out}(\mathbf{MP}))\}$$

The interaction semantics of a configuration $[\![K]\!]$ is the set of interaction paths associated with admissible computation paths of K.

Definition 4.10 ($[\![K]\!]$):

$$[\![K]\!] = \{ip \mid (\exists \pi \in \mathcal{A}(K))\, ip = cp2ip(\pi)\}$$

Interaction sequences should be thought of as convenient representations of totally ordered multisets of I/O interactions. Two representations of the same totally ordered multiset can be considered equivalent. Because of the presence of `idle` transitions, the set of interaction sequences for a configuration is closed under this equivalence and representation details can be ignored for most purposes.

Definition 4.11 ($[\![\langle D\rangle^{\rho}_{\chi}]\!]$): To define the interaction path semantics of a top-level diagram, $[\![\langle D\rangle^{\rho}_{\chi}]\!]$, all we need to do is define the corresponding initial configuration:

$$[\![\langle D\rangle^{\rho}_{\chi}]\!] = [\![\langle \{\!\!\{D\}\!\!\} \cdot \emptyset\rangle^{\ \rho}_{\ \chi}]\!]$$

Notice that by our implicit scoping and declaration convention (see section 2), all variables appearing in a top-level diagram are bound during execution. Extending the scope of the above correspondence, we adopt the convention that whenever a top-level diagram, $\langle D\rangle^{\rho}_{\chi}$, occurs in a context where a configuration is expected it denotes the corresponding configuration, $\langle \{\!\!\{D\}\!\!\} \cdot \emptyset\rangle^{\ \rho}_{\ \chi}$.

Now we can give the definition of the test for failed assertions.

Definition 4.12: A diagram is assertion-valid, $\mathrm{OK}(\langle D\rangle^{\rho}_{\chi})$, iff there is no $\pi \in \mathcal{A}(\langle D\rangle^{\rho}_{\chi})$ that contains an $\mathtt{assert}(\mathit{false})$-labelled transition.

4.3. Commentary on the Specification Diagram Semantics

The use of local reaction rules to define a labelled transition system is fairly common. The **SD** rules correspond to a language with syntax having some features of process algebras such as CSP or the π-calculus, but with asynchronous message passing and name handling modeled on the actor approach. Also, the syntax is not intended to define an algebra of processes, but to be a notation for specifying interaction patterns. In the following we discuss some of the more subtle and perhaps nonstandard aspects of the rules.

The `receive` rule contains implicit pattern-matching. Diagram variables in M_{d} are considered pattern variables, and are matched against the packet mp. It is important to note that the receiver a_{d} is not considered part of the pattern, rather it is evaluated in the before receipt environment. Thus only those actors considered to be internal to the system are able to receive messages and it is not possible to receive a message destined for an arbitrary actor. $acq(\bar{u}) \subseteq acq(mp)$ holds by the pattern match thus insuring that the actor locality laws are obeyed by the `receive` transition. The `pick` rule assigns to the variable x an arbitrary value based on the names of actors currently known. `fresh` corresponds to actor creation, assigning an actor name to x which is not currently known. `assign` updates the value assigned to x, the new value being the value ψ in the current environment, if this value is defined. (By redex-fairness a computation path containing an undefined (stuck) assignment redex is not admissible.) All of these rules operate on the lexically closest binding of the effected variable. Note that by our variable declaration convention, these variables will be bound in the reduction context environment, although the binding may be some arbitrarily chosen initial value.

The `scope-in` rule allocates a new local environment to store the values of the newly declared variables and initially assigns them arbitrarily chosen values.

The intuitive semantics of $\mathtt{constrain}(\phi)$ is that the computation continues if the constraint holds and otherwise the resulting computation path is not allowed. The second clause of `constrain` causes the computation path to be inadmissible by requiring a receive by a freshly created actor name that can never be sent a message. This is a technical trick to express inadmissibility locally in terms of message passing requirements, rather than add another parameter to the notion of admissibility. An alternative semantics would be to simply not reduce the redex if the constraint expression does not evaluate to $true$. In this case it might later become true and reduce, or it might remain stuck, and be inadmissible by the redex fairness requirement. The `assert` rule, on the other hand, reduces if the predicate expression is defined. In this case it has no effect other than the label issued, which if ever false means the diagram viewed as a proposition is false (see Definition 4.12). If the predicate remains undefined then the redex is stuck and, as for the `constrain` case, the computation path is not admissible by redex fairness.

4.3.1. *Admissibility*

One unusual aspect of the semantics lies in the details of the admissibility requirement. The **SD** admissibility requirement rules out any computation path which contains a redex that is stuck, *i.e.*, does not reduce.

We use the fact that admissibility rules out paths with false constraints in the definition of the `if-then-else` macro. The macro expansion of

$$\mathtt{if}\ \phi\ \mathtt{then}\ D_1\ \mathtt{else}\ D_2$$

illustrates this: in the expansion

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

the choice could pick the wrong branch, for instance taking the left branch when $\neg\phi$ held, but the constraint would fail, and the path would not progress and thus be ruled out.

The redex admissibility requirement is significantly stronger than standard fairness requirements, and makes the computation system unrealizable. Even without requiring admissibility the computation system is unrealizable because the value of expressions need not be computable. However, restricting to the case where expressions are computable, some diagrams still may not be realized by any actor computation. One example is:

$\mathtt{pick}(nomorezeros = false);$
$[\ (\mathtt{receive}(a \triangleleft 0);\ \mathtt{constrain}(\neg nomorezeros);$
$\ nomorezeros := true;\ \mathtt{send}(c \triangleleft 1))$
$\oplus$
$(\mathtt{receive}(a \triangleleft x);\ \mathtt{constrain}(\neg(x = 0\ \wedge\ nomorezeros));$
$\ \mathtt{send}(c \triangleleft 0))\ ]^{0..\infty}$

This diagram replies 0 to all inputs, except the *last* 0 input *may* get a 1 reply. No realizable system can foresee the future to know when the last input of a particular form has arrived. The source of the uncomputability here is `constrain`, which for obvious reasons is called a “miraculous” command in the Dijkstra language.

With respect to messages, admissibility is designed both to rule out paths with buffered messages in μ not received, and to rule out paths in which there is a `receive` redex for which a matching message is never input. A simple consequence of the latter is that **SD**s can express requirements that certain messages must be input. A simple consequence of the former is that message packet restriction can be expressed implicitly in **SD**'s, simply by having no receives for some packets.

Lemma 4.13: Let V be a subset of $\mathbf{Msg}$ and let D be a diagram in which no occurrence of $\mathtt{receive}(a_d \triangleleft M_d)$ could possibly match a message of the form $a \triangleleft v$ for $v \in V$ (for any $a \in \mathbf{A}$). Then

$$[\![\langle D\rangle^{\rho}_{\chi}]\!] = [\![\langle D\rangle^{\rho}_{\chi} \restriction \mathbf{A} \triangleleft V]\!]$$

Proof. We only need to show the $\subseteq$ direction because the other direction holds by definition. For this let $ip \in [\![\langle D\rangle^{\rho}_{\chi}]\!]$ and let $\pi \in \mathcal{A}(\langle D\rangle^{\rho}_{\chi})$ such that $cp2ip(\pi) = ip$. Suppose $\pi(i)$ has label $\mathtt{in}(a \triangleleft M)$ for some i and some $a \triangleleft M$. Then message fairness, the message must be received, and hence there must be an occurrence $\mathtt{receive}(a_d \triangleleft M_d)$ in D that matches $a \triangleleft M$ and hence $M \notin V$. □

4.3.2. *Relation to the Actor Model*

Specification diagrams were designed as a notation for describing sets of interaction paths. In particular, we want our specification notation to be able to express both high-level coordination patterns, and assumptions about the environment, as well as properties of a components behavior given such assumptions. Thus the **SD** semantics differs from the semantics of traditional actor programming languages in a number of ways. One difference has to do with synchronization. In an **SD** computation more that one actor can participate in a given thread of activity and a single actor can participate in multiple threads of activity. The former allows **SD**'s to express some forms of synchronization constraints. As shown in [20] such constraints can often be expressed by more complex patterns of message passing. The ability of a single actor to participate in multiple threads is in fact a property of the original actor model [25] in which actors were *unserialized* and could work on multiple tasks simultaneously.

A second difference is a consequence of the redex-admissibility requirement applied to `receive` redexes. A basic tenet of the actor model and of open systems in general is that a system component cannot control the environment in which it lives. In particular it cannot *require* that a particular message be sent. Thus an actor waiting to receive a message, will patiently wait, possibly forever. Furthermore, an actor system can not refuse to take a message addressed to a receptionist. In general there is no obligation to reply, and the sender is not forced to block waiting for a reply, although it may choose to do so. These distinctions are about language and expressiveness and not about the underlying computation model.

4.3.3. *Choice*

The choice operator $\oplus$ is just a coin flip, analogous to the internal form of choice from process algebra. However, in the context of the manner in which admissibility is defined, this internal choice operator can model the external forms of choice found in process algebra. For instance, in

$$\mathtt{receive}(a \triangleleft \mathtt{wow}); D_0 \qquad \oplus \qquad \mathtt{receive}(a \triangleleft \mathtt{wee}); D_1$$

even though the choice is a coin flip, if the `wow` choice was taken and the only input was `wee`, the $\mathtt{receive}(a \triangleleft \mathtt{wow})$ will starve and the computation ruled unfair. Thus it is as if that choice never happened, and the effect is the same as if this were an external choice. The specification diagram analog of external choice is the constraints on the environment due to the admissibility conditions that allow a specification diagram to determine messages that are acceptable and messages that are prevented. Thus external choice is more of an admissibility issue. As noted above, in the underlying actor model, external choice is not meaningful, since an actor system itself cannot so constrain the environment.

5. Proving Specifications Correspond

In this section we give two examples to illustrate reasoning about diagrams. In the first example we establish interaction equivalence of three ticker specification diagrams. In the second example two function composer specifications are related.

5.1. Techniques for Reasoning About Interaction Equivalence

We begin by introducing two general techniques useful for establishing properties based on interaction semantics.

Recall that two top-level diagrams are interaction equivalent just if they have the same interaction semantics.

$$\langle D_0\rangle^{\rho}_{\chi} \overset{i}{\simeq} \langle D_1\rangle^{\rho}_{\chi} \quad \text{iff} \quad [\![\langle D_0\rangle^{\rho}_{\chi}]\!] = [\![\langle D_1\rangle^{\rho}_{\chi}]\!]$$

Thus to establish interaction equivalence, we must show that for each admissible computation path π_0 of $\langle D_0\rangle^{\rho}_{\chi}$ there is an admissible computation path π_1 of $\langle D_1\rangle^{\rho}_{\chi}$ with the same associated interaction path, and conversely.

The first proof technique is to simplify the description of the set of interaction paths for a given top-level diagram by restricting attention to computation paths given by a *big-step semantics*, thus reducing the amount of interleaving to be considered. The second technique is a method for giving local descriptions of the correspondence between computation paths.

5.1.1. *Big-step Semantics*

The big-step technique can be applied to arbitrary diagrams. However, to simplify the discussion, we restrict attention to diagrams with *static* structure, that is *tail-recursive* diagrams defined without the use of the `fork` construct. A diagram is tail-recursive if every recursion variable occurrence is such that in any execution, it will be the last redex in the scope of the binding recursion operator. Thus there is no duplication of diagram text caused by recursion – for example two threads expanding to four by parallel recursive calls. Also when the recursion variable redex is reached there is no remaining context inside the calling scope. For example, the diagram macros defining the various loop forms are tail-recursive.

A computation is in big-step form relative to a selected subset of diagram states if it consists of input/output transitions interleaved with sequences of internal transitions connecting the selected states that can be considered as single steps.

The FC computation given in example 4.5 is in big-step form relative to the four families of diagram states FC, FC0, FC1, FC2. For example, the transition sequence with the label

$$\begin{array}{l}\mathtt{rec}; \mathtt{choose}(l); \mathtt{receive}(a \lhd \mathtt{compute}(v)@c); \mathtt{fresh};\\ \qquad \mathtt{send}(af \lhd \mathtt{compute}(v)@cf)\end{array}$$

is a big-step connecting an FC0 state to an FC1 state.

We now make precise the notions of big-step and big-step semantics. Intuitively a big-step is a single-threaded sequence of internal transitions that are independent of any steps that could be executed in parallel. Thus any interleaving of the parallel execution steps can always be permuted so that the steps of the big-step sequence are adjacent, without changing the observed interactions or the fairness properties. Execution of statements without any observable effect can always be included in a big step. For example, `choose`, `rec`, and reads or writes of state variables local to the thread may appear in big steps. Furthermore, some effects may be performed in a big step. A single variable read/write or message receive/send is fine if all other steps are purely local, and a single receive followed by one or more sends is also unaffected by any interleaving. A write followed by a read does not meet the big-step criteria, because an interleaving with a different write could cause the value of the read to change. Formally, we consider three classes of effect: *read effect*, *unbuffered write effect* (observable immediately), and *buffered write effect*(observable sometime in the future).

Definition 5.1 (Redex effects): The effects of an atomic redex are classified in the context of step sequences with states of the form $R[R_i[D_i]]$. An occurrence of a variable in D_i is said to be *local* if it is bound in $R_i[D_i]$. The effects of the atomic redexes are as follows (note a single statement could have both read and write effects):

read effect $\mathtt{receive}(a_{\mathrm{d}} \triangleleft M_{\mathrm{d}}), x := \ldots y \ldots, \mathtt{constrain}(\ldots y \ldots),$ $\mathtt{send}(\ldots y \ldots)$ for y non-local

unbuffered write effect $x := \psi, \mathtt{pick}(x), \mathtt{fresh}(x), \mathtt{receive}(a_{\mathrm{d}} \triangleleft \ldots x \ldots)$ for x non-local

buffered write effect $\mathtt{send}(a_{\mathrm{d}} \triangleleft M_{\mathrm{d}})$

A step is *effect-free* if it has no effect (read, unbuffered write or buffered write).

In the following, we assume diagrams have been placed in a form where each $\mathtt{rec}\,X.D$ uses a unique recursion variable X, in addition to our assumption of fork-freeness and tail-recursion.

Definition 5.2 (Big step): A *big step* is a rule sequence the form

$$[l_i : R[R_i[D_i]] \xrightarrow[(\mu_s)_i]{(\mu_r)_i} R[[R_{i+1}[D_{i+1}]] \bigm| 0 \leq i < n]$$

such that

(1) for each i, D_i is a redex and R_i is defined without use of the $|$ clauses;

(2) each $\mathtt{rec}(X)$ label appears at most once in $l_1, \ldots, l_n$, meaning each recursion operator is unrolled at most once per big step;

(3) there is no receive by a freshly created actor; that is, there is no label subsequence of the form $\mathtt{fresh}(x)\ldots\mathtt{receive}(x \triangleleft M_{\mathrm{d}})$

and furthermore

(4) if there is a step with an unbuffered write effect, there may be no read effects present in other steps;

(5) there is at most one step with a read effect;

(6) buffered write effects must not occur before any read or unbuffered write effects.

In the above definition, (1) ensures that the rule sequence is single threaded, (2) guaranteeing finiteness of big steps connecting diagram states, (3) is needed because an actor can only receive a message after its name has been exported, and this entails an ordering that prevents permuting steps. Conditions (4-6) ensure permutablility with parallel computations.

Definition 5.3 (Big-step semantics): A *big-step semantics* for a top-level diagram is given by selecting a subset of the reachable diagram states such that

(1) any sequence of rule applications starting in one of the selected states reaches another selected state in finitely many steps, and

(2) the resulting sequence is a big-step.

The resulting semantics is the set of admissible computations generated by using only input/output transistions and the big-step sequences viewed as atomic transitions.

Lemma 5.4 (Big-step semantics is sound): Given a top-level diagram and a subset of the reachable diagram states that defines a big-step semantics, for every interaction path for the diagram there is a computation of the big-step semantics that is mapped to this path by $cp2ip$.

Proof. This lemma is proved via a *Permutation Lemma* which establishes that certain pairs of actions in different threads can be permuted. Big steps are then formed by iteratively permuting actions thus transforming any admissible computation to a big-step computation with the same associated interaction path. Similar proofs are given in [3, 31]. □

Note that the full admissible computation semantics is also a big-step semantics, obtained by selecting all possible diagram states.

For diagrams with static structure, computations can easily be visualized by marking the active redex points (one for each active thread) and moving the marks around the diagram as the computation proceeds. A selection of intermediate reachable states can be represented as a subset of the possible active redex markings. The states will be parameterized by global diagram parameters plus an environment mapping visible variables to current values. In the function composer computation above the intermediate states are given by marks at the diagram entry (FC), the point just before the receive by the initial receptionist a (FC0), the point just before the receive by the actor bound to cf (FC1), and the point just before the receive by actor bound to cg (FC2). One more mark is needed at the diagram exit to allow for the possibility of termination after a finite number of iterations.

To select a subset of states that give rise to big-step computations for a given diagram D we mark stopping points (redex occurrences) on the diagram that correspond to intermediate states of interest in the computation, for example points for which we want to express invariants or other properties of the computation. This marking is further refined to a marking $\widetilde{D}$ such that the corresponding selected states define a big-step semantics.

5.1.2. *Interaction Simulation*

Given two specification diagram configurations K_0 and K_1, they may be shown interaction equivalent for inputs $\mathbf{Amp}$ by finding an interaction preserving correspondence between their admissible computations $\mathcal{A}(K_j \upharpoonright \mathbf{Amp})$. By Lemma 5.4 we may assume that K_0 and K_1 are configurations arising in some big-step semantics and restrict attention to the admissible big-step computations. An *interaction simulation* is a relation on (big-step) configurations and enabled (big-step) transitions that provides a local means of defining such a correspondence. The idea is to define a relation on (big-step) transitions, lift this relation to admissible configurations, preserving input/output transitions, and then show that admissibility is preserved.[2]

Definition 5.5 (Interaction simulation): Let $\mathbf{K}_j$ be the reachable configurations in some big-step semantics for $j < 2$. A *pre-interaction simulation* for inputs in $\mathbf{Amp}$ consists of a binary relation $\sim_C$ on $\mathbf{K}_0 \times \mathbf{K}_1$ together with functions Φ_j for $j < 2$ mapping enabled internal and idle transitions of K_j to (possibly empty) sequences of big-steps starting from K_{1-j} such that for each related pair $K_0 \sim_C K_1$ the following conditions hold:

(1) K_0 and K_1 have the same interface and the same multiset of undelivered packets to external actors.

[2] It would be desirable to define the local relation so that preserving admissibility is guaranteed, but to achieve this means both complicating the definition of local simulation and greatly resticting the espressiveness of the simulation relation. Thus we have chosen to separate out the prove of preservation of admissibility.

(2) if $K_0 \xrightarrow{tl} K_0'$, with inputs in $\mathbf{Amp}$, then there is K_1' such that $K_0' \sim_C K_1'$ and $K_1 \xrightarrow{\Phi_0(tl)} K_1'$, and dually with 0-1 interchanged (extending Φ_0 as the identity on i/o labels).

A pre-interaction simulation $(\sim_C, \Phi_0, \Phi_1)$ induces functions (also called Φ_j) from computations for K_j to computations for K_{1-j} (in the chosen big-step semantics) for $K_0 \sim_C K_1$. This simulation is an *interaction simulation* if these functions preserve admissibility.

Lemma 5.6 (Isim): Let $\mathbf{K}_j$ be the reachable configurations in some big-step semantics for $j < 2$ and let $(\sim_C, \Phi_0, \Phi_1)$ be an interaction simulation on big-step computations of $(\mathbf{K}_0, \mathbf{K}_1)$ with inputs restricted to $\mathbf{Amp}$. If $K_j \in \mathbf{K}_j$ for $j < 2$, then

$$K_0 \sim_C K_1 \Rightarrow (K_0) \upharpoonright \mathbf{Amp} \overset{i}{\simeq} (K_1) \upharpoonright \mathbf{Amp}$$

Proof. We need only show the interactions are preserved. This follows from the fact that the transition functions extend as the identity on interaction transitions. □

5.2. TICKER EXAMPLE

In this section we prove the Ticker equivalence theorem 3.5:

$$\langle \mathrm{ChoiceTicker}(a) \rangle^a_\emptyset \upharpoonright \mathbf{MP}_{\mathtt{time}} \overset{i}{\simeq}$$
$$\langle \mathrm{ParTicker}(a) \rangle^a_\emptyset \upharpoonright \mathbf{MP}_{\mathtt{time}} \overset{i}{\simeq}$$
$$\langle \mathrm{Ticker}(a) \rangle^a_\emptyset$$

where the parameter a is the name of the ticker and

$$\mathbf{MP}_{\mathtt{time}} = \mathbf{A} \triangleleft \mathtt{time} \mathbin{@} \mathbf{A}.$$

We do this by first defining a big-step semantics for each of the top-level diagrams. Proving equivalence of ChoiceTicker and ParTicker is then a matter of establishing an interaction simulation between computation paths of the corresponding big-step semantics. Equivalence of Ticker and ChoiceTicker requires establishing a more complex correspondence. Local transformations are not adequate for relating Ticker and ChoiceTicker because the Ticker relies on the ability to discard paths when a wrong choice has been made, for example if a choice is made to accept two `time` messages before the next counter increment and only one more `time` message arrives.

The diagram $\mathrm{Ticker}(a)$ is given in Section 3.1.2 (Figure 3) and the diagrams for $\mathrm{ChoiceTicker}(a)$ and $\mathrm{ParTicker}(a)$ are given in Section 3.2.8

(Figure 7). For the readers convenience, we recall the diagrams, in textual form. $\mathrm{Ticker}(a)$ describes an actor that increments its counter after replying to some finite number of `time` requests.

$$\begin{array}{l}
\mathrm{Ticker}(a) = \\
\quad \{\mathtt{pick}(\mathit{count} \in \mathbf{Nat}); \\
\quad\; [\ [\ \mathtt{receive}(a \triangleleft \mathtt{time} @ x);\ \mathtt{send}(x \triangleleft \mathtt{reply}(\mathit{count}))\ ]^{0\ldots\omega}; \\
\quad\quad \mathit{count} := \mathit{count} + 1\ ]^{0\ldots\infty}\ \}
\end{array}$$

The other two describe actors that increment the counter upon receipt of (internally generated) `tick` messages. $\mathrm{ChoiceTicker}(a)$ non-deterministically chooses between accepting `time` and `tick` messages.

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

while $\mathrm{ParTicker}(a)$ accepts `time` and `tick` messages in parallel.

$$\begin{array}{l}
\mathrm{ParTicker}(a) = \\
\quad \{\ \mathit{count} : \mathit{count} := 0;\ \mathtt{send}(a \triangleleft \mathtt{tick}); \\
\quad\; [\ \mathtt{receive}(a \triangleleft \mathtt{tick});\ \mathit{count} := \mathit{count} + 1;\ \mathtt{send}(a \triangleleft \mathtt{tick})\ ]^{\infty} \\
\quad\; | \\
\quad\; [\ \mathtt{receive}(a \triangleleft \mathtt{time} @ x);\ \mathtt{send}(a \triangleleft \mathtt{reply}(\mathit{count}))\ ]^{0\ldots\infty}\ \}
\end{array}$$

5.2.1. *The Choice Ticker Big-step Semantics*

We define a big-step semantics for the $\mathrm{ChoiceTicker}(a)$ diagram by defining a marking, $\mathrm{ChoiceTickerM}(a)$, of the diagram, and showing that this marking determines a big-step semantics. $\mathrm{ChoiceTickerM}(a)$ has three marks: the diagram entry point, the entry to the loop, and the exit point. Corresponding to the three marks there are three families of states parameterized by the internal actor and the value of the diagram variable, where relevant:

- $\mathrm{cT}(a)$ — the diagram entry point with internal actor a ($\mathrm{cT}(a) = \mathrm{ChoiceTicker}(a)$),
- $\mathrm{cTl}(a, n)$ — the loop entry point with $\mathit{count} = n$,
- $\mathrm{End}(a)$ — the exit point.

The set of rule sequences connecting these diagram states are:

$$(L_0) \quad \mathrm{cT(a)} \xrightarrow[a\triangleleft\mathtt{tick}]{} \mathrm{cTl(a,0)}$$

$$(L_1) \quad \mathrm{cTl(a,n)} \xrightarrow[a\triangleleft\mathtt{tick}]{a\triangleleft\mathtt{tick}} \mathrm{cTl(a,n+1)}$$

$$(L_2) \quad \mathrm{cTl(a,n)} \xrightarrow[c\triangleleft\mathtt{reply}(n)]{a\triangleleft\mathtt{time}@c} \mathrm{cTl(a,n)}$$

$$(L_3) \quad \mathrm{cTl(a,n)} \xrightarrow{a\triangleleft\mathtt{tick}} \mathrm{End(a)}$$

where

$$L_0 = \mathtt{assign}; \mathtt{send}$$
$$L_1 = \mathtt{rec}(X); \mathtt{choose}(l); \mathtt{choose}(l); \mathtt{receive}; \mathtt{assign}; \mathtt{send}$$
$$L_2 = \mathtt{rec}(X); \mathtt{choose}(l); \mathtt{choose}(r); \mathtt{receive}; \mathtt{send}$$
$$L_3 = \mathtt{rec}(X); \mathtt{choose}(r); \mathtt{receive}$$

Clearly each of these is big-step and thus we have a big-step semantics.

5.2.2. *A Big-step Semantics for the Parallel Ticker*
Defining a big-step semantics for the parallel ticker requires a little more care, since this diagram has two threads of activity that share state. Let diagram $\mathrm{ParTickerM}(a)$ be $\mathrm{ParTicker}(a)$ with the following marks: the diagram entry point, the entry point to the loop on each parallel thread, the exit point of the `time` thread, the point between the receive and send on the `time` thread, and the point between the send and the assign on the `tick` thread. This marking leads to seven families of states the initial state, and six joint states of the two threads.

– $\mathrm{pT}(a) = \mathrm{ParTicker}(a)$ — the diagram entry point with internal actor a.

– $\mathrm{pT}_{i,j}(a, n, c) = \{count = n : [\mathrm{PTl}_i(a) \mid \{x = c : \mathrm{PTr}_j(a)\}]\}$ — two threaded states with internal actor a, count n and customer c.

where the sub-diagrams $\mathrm{PTl}_i(a)$ and $\mathrm{PTr}_j(a)$ for $i \in \{0,1\}$ and $j \in \{0,1,e\}$ are intermediate diagrams for the two parallel branches, given by

$$\mathrm{PTl}_0(a) = \mathtt{rec}(X)(\mathtt{receive}(a\triangleleft\mathtt{tick});\ count := count{+}1;\ \mathtt{send}(a\triangleleft\mathtt{tick});\ X)$$
$$\mathrm{PTl}_1(a) = count := count + 1;\ \mathtt{send}(a \triangleleft \mathtt{tick});\ \mathrm{PTl}_0(a)$$
$$\mathrm{PTr}_0(a) = \mathtt{rec}(Y)(\mathtt{skip}\oplus [\ \mathtt{receive}(a\triangleleft\mathtt{time}@x);\ \mathtt{send}(x\triangleleft\mathtt{reply}(count));\ Y\ ])$$
$$\mathrm{PTr}_1(a) = \mathtt{send}(x \triangleleft \mathtt{reply}(count));\ \mathrm{PTr}_0(a)$$

$\mathrm{PTr}_e(a) = \mathtt{skip}$

The possible rule sequences connecting these states are (omitting mention of rules with no effect):

$$(L_0) \quad \mathrm{pT}(a) \xrightarrow[a\triangleleft\mathtt{tick}]{} \mathrm{pT}_{0,0}(a, 0, \mathtt{nil})$$

$$(L_{l0}) \quad \mathrm{pT}_{0,j}(a, n, c) \xrightarrow{a\triangleleft\mathtt{tick}} \mathrm{pT}_{1,\mathrm{j}}(a, n, c)$$

$$(L_{l1}) \quad \mathrm{pT}_{1,j}(a, n, c) \xrightarrow[a\triangleleft\mathtt{tick}]{} \mathrm{pT}_{0,\mathrm{j}}(a, n+1, c)$$

$$(L_{r0}) \quad \mathrm{pT}_{i,0}(a, n, c') \xrightarrow{a\triangleleft\mathtt{time}@c} \mathrm{pT}_{i,1}(a, n, c)$$

$$(L_{r1}) \quad \mathrm{pT}_{i,1}(a, n, c) \xrightarrow[c\triangleleft\mathtt{reply}(n)]{} \mathrm{pT}_{i,0}(a, n, c)$$

$$(L_{re}) \quad \mathrm{pT}_{i,0}(a, n, c) \longrightarrow \mathrm{pT}_{i,e}(a, n, c)$$

It is easy to check (by filling in the hidden rule applications) that these states and rule sequences determine a big-step semantics. The need for marking intermediate points on the two threads is due the the fact that one thread reads $count$ and the other writes it. Specifically, according to the big-step definition, L_{r0} and L_{r1} cannot be condensed into one big-step because there can be at most one read effect in a canonical step, and the receive and the read of the value n are read effects in the two respective rules.

5.2.3. *Choice Ticker—Parallel Ticker Equivalence*

To prove the equivalence of the choice and parallel ticker diagrams, we define an interaction simulation for the big-step semantics that relates the choice and parallel ticker top-level diagrams restricted to time request inputs. The relation $\sim_C$ on configurations is given by

(init) $\langle\, \mathrm{cT}(a) \cdot \mu \,\rangle^{a}_{\chi} \sim_C \langle\, \mathrm{pT}(a) \cdot \mu \,\rangle^{a}_{\chi}$

(0,0) $\langle\, \mathrm{cTl}(a, n) \cdot \mu \,\rangle^{a}_{\chi} \sim_C \langle\, \mathrm{pT}_{0,0}(a, n) \cdot \mu \,\rangle^{a}_{\chi}$

(1,0) $\langle\, \mathrm{cTl}(a, n) \cdot a \triangleleft \mathtt{tick} \cdot \mu \,\rangle^{a}_{\chi} \sim_C \langle\, \mathrm{pT}_{1,0}(a, n) \cdot \mu \,\rangle^{a}_{\chi}$

(0,1) $\langle\, \mathrm{cTl}(a, n) \cdot a \triangleleft \mathtt{time} @ c \cdot \mu \,\rangle^{a}_{\chi} \sim_C \langle\, \mathrm{pT}_{0,1}(a, n, c) \cdot \mu \,\rangle^{a}_{\chi}$

(1,1) $\langle\, \mathrm{cTl}(a, n) \cdot a \triangleleft \mathtt{tick} \cdot a \triangleleft \mathtt{time} @ c \cdot \mu \,\rangle^{a}_{\chi} \sim_C \langle\, \mathrm{pT}_{1,1}(a, n, c) \cdot \mu \,\rangle^{a}_{\chi}$

(1,e) $\langle\, \mathrm{End}(a) \cdot \mu \,\rangle^{a}_{\chi} \sim_C \langle\, \mathrm{pT}_{1,e}(a, n) \cdot \mu \,\rangle^{a}_{\chi}$

(0,e) $\langle\, \mathrm{End}(a) \cdot \mu \,\rangle^{a}_{\chi} \sim_C \langle\, \mathrm{pT}_{0,e}(a, n) \cdot a \triangleleft \mathtt{tick} \cdot \mu \,\rangle^{a}_{\chi}$

The transition function Φ_{c2p} in the choice to parallel ticker direction maps single cT steps to macro steps of pT as follows:

$\Phi_{\mathrm{c2p}}(L_0) = L_0$

$\Phi_{\mathrm{c2p}}(L_1) = L_{l0}; L_{l1}$

$\Phi_{\mathrm{c2p}}(L_2) = L_{r0}; L_{r1}$

$\Phi_{\mathrm{c2p}}(L_3) = L_{re}$

$\Phi_{\mathrm{c2p}}(\mathtt{idle}) = L_{l0}; L_{l1}$

The loop entry transitions correspond in an obvious way. The receive transition maps apply to the correspondence case (0,0) and `idle` transition map applies to the correspondence case (0,e). These are the only cases that arise in the cT to pT direction. The non-trivial idle transition map is needed because the parallel ticker keeps on ticking forever while the choice ticker stops ticking if requests stop arriving.

The transition function Φ_{p2c} in the parallel to choice ticker direction maps the receive pT steps to idle steps of cT and the sends to receive/send cT steps as follows:

(0) $\Phi_{\mathrm{p2c}}(L_0) = L_0$ for the correspondence case (init)

(l0) $\Phi_{\mathrm{p2c}}(L_{l0}) = \mathtt{idle}$ for the correspondence cases (0,0), (0,1) and (0,e).

(l1) $\Phi_{\mathrm{p2c}}(L_{l1}) = L_1$ for the correspondence cases (1,0) and (1,1)

(r0) $\Phi_{\mathrm{p2c}}(L_{r0}) = \mathtt{idle}$ for the correspondence cases (0,0) and (1,0).

(r1) $\Phi_{\mathrm{p2c}}(L_{r1}) = L_2$ for the correspondence cases (0,1) and (1,1).

(re) $\Phi_{\mathrm{p2c}}(L_{re}) = L_3$ for the correspondence cases (0,0) and (1,0).

(l13) $\Phi_{\mathrm{p2c}}(L_{l1}) = \mathtt{idle}$ for the correspondence case (1,e)

To see that this satisfies the conditions for being an **SD** interaction simulation we need to show that admissible paths are mapped to admissible paths by the transition functions. Suppose π is an admissible computation for the choice ticker and $\pi' = \Phi_{\mathrm{c2p}}(\pi)$. If π' is not admissible then either there is a message that is not delivered or an exposed redex that is not reduced. In the message case, external messages and their outputs correspond so it must be a `tick` or `time` message that is not delivered. By definition of the transition map and configuration correspondence there must also be such a message undelivered in π, contradicting admissibility of π. Similarly if there is a redex exposed and not reduced it must be one of the parallel thread entry points. Again by the correspondence this means that the loop entry point for the choice ticker remains unreduced from some point which is not possible. Going the other direction, suppose π' is an admissible path for the parallel ticker, $\pi = \Phi_{\mathrm{p2c}}(\pi')$, and path' is not admissible. Suppose a `tick` or `time` message is undelivered in π, then either there is a corresponding undelivered message in π', the parallel ticker is in a state that has received that message and is ready to do a send, whose corresponding transition in π would receive the message. Thus it can not remain undelivered. Similarly, if the choice ticker is not in its end state the corresponding parallel ticker must eventually to a transition that causes the choice redex to reduce. Thus by Lemma 5.6 we have proved

$$\mathrm{ChoiceTicker}(a)\,{}^{a}_{\emptyset} \upharpoonright \mathbf{MP}_{\mathtt{time}} \stackrel{i}{\simeq} \mathrm{ParTicker}(a)\,{}^{a}_{\emptyset} \upharpoonright \mathbf{MP}_{\mathtt{time}}$$

5.2.4. *Choice Ticker—Ticker Equivalence*

Next we consider the $\mathrm{Ticker}(a)$. We first define a marking, $\mathrm{TickerM}(a)$, for this diagram and verify that it determines a big-step semantics. Then we show that there is an interaction preserving correspondence between the big-step semantics of $\mathrm{Ticker}(a)$ and that of $\mathrm{ChoiceTicker}(a)$ (restricting inputs to $\mathbf{MP}_{\mathtt{time}}$). To describe the marking, we first expand the loop abbreviations in the $\mathrm{Ticker}(a)$ diagram obtaining

$$
\begin{array}{l}
\mathrm{Ticker}(a) = \\
\quad \{\mathtt{pick}(\mathit{count} \in \mathbf{Nat}); \\
\quad\ \mathtt{rec}\,(X)(\ (\mathtt{pick}(\mathit{icnt} \in \mathbf{Nat}); \\
\qquad\qquad\quad \mathtt{rec}\,(Y)(\,[\ \mathtt{constrain}(\mathit{icnt} = 0);\ \mathtt{skip}\ ] \\
\qquad\qquad\qquad\qquad \oplus \\
\qquad\qquad\qquad\qquad [\ \mathtt{constrain}(\mathit{icnt} > 0);\ \mathtt{receive}(a \triangleleft \mathtt{time} @ x); \\
\qquad\qquad\qquad\qquad\ \ \mathtt{send}(x \triangleleft \mathtt{reply}(\mathit{count}));\ \mathit{icnt} := \mathit{icnt} - 1;\ Y\ ]\,); \\
\qquad\qquad\quad \mathit{count} := \mathit{count} + 1;\ X) \\
\qquad\qquad\quad \oplus \\
\qquad\qquad\quad \mathtt{skip}\ )\ \}
\end{array}
$$

$\mathrm{TickerM}$ has marks at the diagram entry point, at entry to $\mathtt{rec}\,(X)$, at the entry to $\mathtt{rec}\,(Y)$, and at the diagram exit point. Since Ticker has a single thread of activity, each mark corresponds to a family of states parameterized by the internal actor and the current count. Thus we have

- $\mathrm{T}(a)$ — the initial state;
- $\mathrm{TX}(a, n)$ — the state corresponding to entry point to $\mathtt{rec}\,(X)$, with $\mathit{count} = n$;
- $\mathrm{TY}(a, n, m)$, the state corresponding to the entry point in the to $\mathtt{rec}\,(Y)$ with $\mathit{count} = n$ and $\mathit{icnt} = m$; and
- $\mathrm{End}(a)$ — the state corresponding to the exit point.

The possible sequences of rules connecting these states (suppressing labels for rules with no effect) are the following

$(L_0)\quad \mathrm{T}(a) \longrightarrow \mathrm{TX}(a,n)$

$(L_{xe})\quad \mathrm{TX}(a,n) \longrightarrow \mathrm{End}(a)$

$(L_{xy})\quad \mathrm{TX}(a,n) \longrightarrow \mathrm{TY}(a,n,m)$

$(L_{yx})\quad \mathrm{TY}(a,n,0) \longrightarrow \mathrm{TX}(a,n+1)$

$(L_{yy})\quad \mathrm{TY}(a,n,m+1) \xrightarrow[c\triangleleft\mathtt{reply}(n)]{a\triangleleft\mathtt{time}@c} \mathrm{TY}(a,n,m)$

Again it is easy to see that the above states and rule sequences determine a big-step semantics. Now we establish the claimed correspondence between the big-step semantics of $\mathrm{Ticker}(a)$, call it $\mathcal{B}_{\mathrm{T}}$, and that of $\mathrm{ChoiceTicker}(a)$, $\mathcal{B}_{\mathrm{cT}}$,.

Lemma 5.7 (cT-T): There are functions mapping $\mathcal{B}_{\mathrm{cT}} \restriction \mathbf{MP}_{\mathtt{time}})$ to $\mathcal{B}_{\mathrm{T}}$ and conversely that preserve the associated interaction path.

Proof. The T to cT direction is easy. Define a map from T-configurations to cT-configurations as follows.

$\langle\, \mathrm{T}(a)\cdot\mu \,\rangle^{a}_{\chi} \mapsto \langle\, \mathrm{cT}(a)\cdot\mu \,\rangle^{a}_{\chi}$ where μ contains only `time` packets

$\langle\, \mathrm{End}(a)\cdot\mu \,\rangle^{a}_{\chi} \mapsto \langle\, \mathrm{End}(a)\cdot\mu \,\rangle^{a}_{\chi}$ where μ contains only `reply` packets

$\langle\, \mathrm{TX}(a,n)\cdot\mu \,\rangle^{a}_{\chi} \mapsto \langle\, \mathrm{cT}(a,n)\cdot\mu\cdot a\triangleleft\mathtt{tick} \,\rangle^{a}_{\chi}$
where μ contains only `time` or `reply` packets

$\langle\, \mathrm{TY}(a,n,m)\cdot\mu \,\rangle^{a}_{\chi} \mapsto \langle\, \mathrm{cT}(a,n)\cdot\mu\cdot a\triangleleft\mathtt{tick} \,\rangle^{a}_{\chi}$
where μ contains only `time` or `reply` packets

Assume $\pi \in \mathcal{B}_{\mathrm{T}}$. Construct π' so that at each stage i, the source and target states are the image of those in π. If the label of $\pi(i)$ is an input/output or idle transition then the label of $\pi'(i)$ is the same. If the label of $\pi(i)$ consumes a `time` packet then the label of $\pi'(i)$ is the internal transition of cT consuming the same packet. If the label of $\pi(i)$ is the initial transition choosing an n, the $\pi'(i)$ does n `tick`s (treated as a macro step so the bookkeeping is easier). If the label of $\pi(i)$ is the transition to the $\mathrm{End}(a)$ state then the label of $\pi'(i)$ is the transition the consumes a tick and moves to the corresponding state. If the label of $\pi(i)$ is the move from TY to TX then the label of $\pi'(i)$ is consumes and sends a `tick`. Finally, if the label of $\pi(i)$ is the move from TX to TY choosing an m, the label of $\pi'(i)$ is `idle`. It is easy to check that π' so constructed is a path of $\mathcal{B}_{\mathrm{cT}}$.

The cT to T direction is little tricker, since the mapping of $\mathcal{B}_{\mathrm{cT}}$ configurations to $\mathcal{B}_{\mathrm{T}}$ configurations depends on the stage in the path. For $\pi \in \mathcal{B}_{\mathrm{cT}}$, construct π' by mapping each transition $\pi(i)$ to a segment of one or two transitions which we refer to as $\pi'(i)$ to simplify bookkeeping. By induction and intial conditions we assume we know the source of $\pi'(i)$.

- if $\pi(i)$ is an input, output or idle transition then $\pi'(i)$ is the transition with the same label starting with the known source.
- if $\pi(i)$ is the initial internal transition with with source $\langle\ \mathrm{cT}(a) \cdot \mu\ \rangle^{a}_{\chi}$ (where μ has only `time` packets) and target $\langle\ \mathrm{cTl}(a, 0) \cdot \mu \cdot a \triangleleft \mathtt{tick}\ \rangle^{a}_{\chi}$, then $\pi'(i)$ is a segment of two steps with source $\langle\ \mathrm{T}(a) \cdot \mu\ \rangle^{a}_{\chi}$ and target $\langle\ \mathrm{TY}(a, 0, m) \cdot \mu\ \rangle^{a}_{\chi}$ where m is the number of `time` transitions in π after i and before the next `tick`.
- if $\pi(i)$ is a `time` transition with source $\langle\ \mathrm{cTl}(a, n) \cdot \mu\ \rangle^{a}_{\chi}$ then $\pi'(i)$ is a a `time` transition with source $\langle\ \mathrm{TY}(a, n, m + 1) \cdot \mu\ \rangle^{a}_{\chi}$
- if $\pi(i)$ is a `tick` receive/send transition with source $\langle\ \mathrm{cTl}(a, n) \cdot \mu\ \rangle^{a}_{\chi}$ then $\pi'(i)$ is a segment of two transitions in which the state moves from $\mathrm{TY}(a, n, 0)$ to $\mathrm{TY}(a, n + 1, m)$ where again m is the number of `time` transitions before the next `tick`
- if $\pi(i)$ is a `tick` receive only transition to the end state, then $\pi'(i)$ is a segment of two transitions in which the state moves from $\mathrm{TY}(a, n, 0)$ to $\mathrm{TX}(a, n + 1)$ and then to the end state.

Again it is fairly easy to see that $\pi'(i)$ so constructed is a path of $\mathcal{B}_{\mathrm{T}}$. □

5.3. Function composer example

We now establish Theorem 3.4 of Section 3.2, which states that the purely local computation of $g \circ f$ is equivalent to its distributed implementation. This illustrates how to simplify a complex composite diagram taking advantage of the additional known context and emerging internal invariants. Recall that for actor names a, af, ag and functions $f, g : V \to V$ on $V \subseteq U$, the diagrams $\mathrm{F}(a, f)$, $\mathrm{FC}(a, af, ag)$, and $\mathrm{C}(a, f, g, af, ag)$ are defined by

$$\mathrm{F}(a, f) = [\mathtt{receive}(a \triangleleft \mathtt{compute}(x) \mathbin{@} xc);\ \mathtt{send}(xc \triangleleft \mathtt{reply}(f(x)))]^{0\ldots\infty}$$

$$\mathrm{FC}(a, af, ag) =$$

$$\begin{aligned}&[\ \mathtt{receive}(a \triangleleft \mathtt{compute}(x) \mathbin{@} xc);\ \mathtt{fresh}(xf);\ \mathtt{send}(af \triangleleft \mathtt{compute}(x) \mathbin{@} xf);\\ &\ \mathtt{receive}(xf \triangleleft \mathtt{reply}(y));\ \mathtt{fresh}(xg);\ \mathtt{send}(ag \triangleleft \mathtt{compute}(y) \mathbin{@} xg);\\ &\ \mathtt{receive}(xg \triangleleft \mathtt{reply}(z));\ \mathtt{send}(xc \triangleleft \mathtt{reply}(z))\ ]^{0\ldots\infty}\end{aligned}$$

$\mathrm{C}(a, f, g, af, ag) = (\mathrm{FC}(a, af, ag) \mid \mathrm{F}(af, f) \mid \mathrm{F}(ag, g))$

What we must prove is:

$$\mathrm{C}(a, f, g, af, ag)\,{}^{a}_{\emptyset} \overset{i}{\simeq} \mathrm{F}(a, g \circ f)\,{}^{a}_{\emptyset}$$

That is, we must show that

$$[\![\mathrm{C}(a, f, g, af, ag)\,{}^{a}_{\emptyset}]\!] = [\![\mathrm{F}(a, g \circ f)\,{}^{a}_{\emptyset}]\!]$$

As for the ticker examples, the first step is to define a suitable big-step semantics.

5.3.1. *Big-steps for the Function Computer*
The marked function computer diagram, $\mathrm{FM}(a, f)$, has two marks—one at the entry to the loop, which is also the entry point of the diagram, and one at the diagram exit point. Corresponding to this marking we have two families of states: $\mathrm{F}(f)(a) = \mathrm{F}(a, f)$ corresponding to the entry point; and $\mathrm{FEnd}(a)$ corresponding to the exit point. The rule sequences connecting these states correspond to the following big-steps.

(L_{end}) $\quad \mathrm{F}(a, f) \longrightarrow \mathrm{FEnd}(a)$

(L_F) $\quad \mathrm{F}(a, f) \xrightarrow[c \triangleleft \mathtt{reply}(f(v))]{a \triangleleft \mathtt{compute}(v) @ c} \mathrm{F}(a, f)$

5.3.2. *Big steps for the Function Composer*
For the big-step simplification of the function composer we form the marked diagram $\mathrm{CM}(a, f, g, af, ag)$ by putting marks on the $\mathrm{FC}(a, af, ag)$ part at the loop entry point, before the second and third `receives`, and at the loop exit point, and marking the entry and exit points of the $\mathrm{F}(af, f)$ and $\mathrm{F}(ag, g)$ parts. We represent the states corresponding to these marks as a family of states indexed by triples i, j, k indicating the diagram mark for each part, and parameterized by an environment, γ, binding relevant state variables:

$$\mathrm{C}_{i,j,k}(a, af, ag, \gamma).$$

The index i gives the FC component state: 0 is the loop entry, 1, 2, indicate the second and third `receives`, and e is the exit point. Similarly j, k give the f and g function computer states, 0 being the loop entry and e being the loop exit. γ binds the state variables $\{x, y, xc, xf, xg\}$, although not all of the bindings are meaningful at any given marking triple. The possible rule sequences connecting states are the union of those for each of the three parallel threads. The rules for the FC part are

(L_{fc0}) $\quad \mathrm{C}_{0,j,k}(a, af, ag, \gamma) \xrightarrow[af \triangleleft \mathtt{compute}(v) @ cf]{a \triangleleft \mathtt{compute}(v) @ c}$

$$\mathrm{C}_{1,j,k}(a, af, ag, \gamma\{x \mapsto v, xc \mapsto c, xf \mapsto cf\}$$
where cf is fresh

$$(L_{fc1}) \quad \mathrm{C}_{1,j,k}(a, af, ag, \gamma) \xrightarrow[ag\triangleleft\mathtt{compute}(w)@cg]{cf\triangleleft\mathtt{reply}(w)}$$
$$\mathrm{C}_{2,j,k}(a, af, ag, \gamma\{y \mapsto w, xg \mapsto cg\}$$
where cg is fresh and $\gamma(xf) = cf$

$$(L_{fc2}) \quad \mathrm{C}_{2,j,k}(a, af, ag, \gamma) \xrightarrow[c\triangleleft\mathtt{reply}(u)]{cg\triangleleft\mathtt{reply}(u)}$$
$$\mathrm{C}_{0,j,k}(a, af, ag, \gamma\{y \mapsto w, xg \mapsto cg\}$$
if $\gamma(xg) = cg$ and $\gamma(xc) = c$

$$(L_{fce}) \quad \mathrm{C}_{0,j,k}(a, af, ag, A, \gamma) \longrightarrow \mathrm{C}_{e,j,k}(a, af, ag, A, \gamma)$$

The rules for the f computer part are

$$(L_{f0}) \quad \mathrm{C}_{i,0,k}(a, af, ag, \gamma) \xrightarrow[cf\triangleleft\mathtt{reply}(f(v))]{af\triangleleft\mathtt{compute}(v)@cf} \mathrm{C}_{i,0,k}(a, af, ag, \gamma)$$

$$(L_{fe}) \quad \mathrm{C}_{i,0,k}(a, af, ag, \gamma) \longrightarrow \mathrm{C}_{i,e,k}(a, af, ag, \gamma)$$

The rules for the g computer part are similar. Again, it is easy to see the this determines a big-step semantics since each sequence has at most a receive followed by a send in addition to some effect free rules.

5.3.3. *Function Computer–Function Composer Equivalence*
To complete the proof of Theorem 3.4, we form an interaction simulation relating initial states

$$\langle\, \mathrm{F}(a, g \circ f)\, \rangle^{a}_{\emptyset} \quad \sim_C \quad \langle\, \mathrm{C}_{0,0,0}(a, af, ag, \gamma)\, \rangle^{a}_{\emptyset}$$

We observe the following property of function composer configurations.

- If the C index is 0, then there are no pending messages to af or ag.
- If the C index is 1, then, either there is a single pending `compute` message to af or a single pending `reply` message to $\gamma(xf)$.
- If the C index is 2, then, either there is a single pending `compute` message to ag or a single pending `reply` message to $\gamma(xg)$.
- If the C index is e, then there are no pending messages to a, af or ag, and no further inputs.

The interaction simulation is as follows (we elide paramaters of $\mathrm{F}/\mathrm{C}/\mathrm{FEnd}$ and the interfaces to avoid clutter):

$$\langle\, \mathrm{F} \cdot \mu \,\rangle \quad \sim_C \quad \langle\, \mathrm{C}_{0,0,0} \cdot \mu \,\rangle$$

$$\langle\, \mathrm{F} \cdot \mu \cdot a \triangleleft \mathtt{compute}(v) \,\rangle \quad \sim_C \quad \langle\, \mathrm{C}_{1,0,0} \cdot \mu \cdot af \triangleleft \mathtt{compute}(v) \,\rangle$$

$$\langle\, \mathrm{F} \cdot \mu \cdot a \triangleleft \mathtt{compute}(v) \,\rangle \quad \sim_C \quad \langle\, \mathrm{C}_{1,0,0} \cdot \mu \cdot cf \triangleleft \mathtt{reply}(f(v)) \,\rangle$$

$$\langle\, \mathrm{F} \cdot \mu \cdot a \triangleleft \mathtt{compute}(v) \,\rangle \quad \sim_C \quad \langle\, \mathrm{C}_{2,0,0} \cdot \mu \cdot ag \triangleleft \mathtt{compute}(f(v)) \,\rangle$$

$$\langle\, \mathrm{F} \cdot \mu \cdot a \triangleleft \mathtt{compute}(v) \,\rangle \quad \sim_C \quad \langle\, \mathrm{C}_{2,0,0} \cdot \mu \cdot cg \triangleleft \mathtt{reply}(g(f(v))) \,\rangle$$

$$\langle\, \mathrm{FEnd} \cdot \mu \cdot a \triangleleft \mathtt{compute}(v) \,\rangle \quad \sim_C \quad \langle\, \mathrm{C}_{1,e,0} \cdot \mu \cdot cf \triangleleft \mathtt{reply}(f(v)) \,\rangle$$

$$\langle\, \mathrm{FEnd} \cdot \mu \cdot a \triangleleft \mathtt{compute}(v) \,\rangle \quad \sim_C \quad \langle\, \mathrm{C}_{2,e,0} \cdot \mu \cdot ag \triangleleft \mathtt{compute}(f(v)) \,\rangle$$

$$\langle\, \mathrm{FEnd} \cdot \mu \cdot a \triangleleft \mathtt{compute}(v) \,\rangle \quad \sim_C \quad \langle\, \mathrm{C}_{2,i,j} \cdot \mu \cdot cg \triangleleft \mathtt{reply}(g(f(v))) \,\rangle$$

for $(i, j) = (0, e)$, $(e, 0)$, or e, e

$$\langle\, \mathrm{FEnd} \cdot \mu \,\rangle \quad \sim_C \quad \langle\, \mathrm{C}_{i,j,k} \cdot \mu \,\rangle$$

for $(i, j, k) = (0, 0, e)/(0, e, 0)/(e, 0, 0)/(0, e, e)/(e, e, 0)/(e, 0, e)/(e, e, e)$

The transition maps are defined in a similar manner to the choice/parallel ticker. For example, in the mapping Φ_{C2F} from C to F, $L_{fc0}, L_{fc1}, L_{f0}, L_{g0}$ transitions all map to idle transitions, and L_{fc2} maps to L_F. By an argument similar to the choice/parallel ticker, admissibility is preserved. Thus by Lemma 5.6, the proof is complete.

6. Related Work

A wide variety of notations for concurrent/distributed system specification have been proposed. Specification diagrams share features with previous work but are still quite separate from existing schools. Different forms of specification have different strengths and weaknesses, and for large systems the combination of multiple techniques will probably be needed. We briefly review some of the related approaches here.

The general idea of taking a diagrammatic approach to specification has been advocated by multiple research projects. A book covering several approaches is [4]. One reason why the time is ripe for graphical approaches is the emergence of graphical editors, which allow specifications to be entered without resort to text-based input (see *e.g.*[9]).

Specification diagrams are most closely related to other forms of message-passing diagrams, diagrams with vertical lines for processes/threads, and horizontal lines for messages. Message passing diagrams have a long history in software specification and are now most widely known as either UML Sequence Diagrams [36], or Message Sequence Charts (MSC's) [27]. However, these sequence diagrams are primarily designed to show possible scenarios of execution, and not to give all possible scenarios. More recent versions of UML sequence diagrams [13] have considerable extra syntax added so more features can be expressed. Several extensions to high-level MSC's have also been proposed to make them more expressive, see *e.g.* [22] and references therein. These systems are still considerably less expressive than specification diagrams, which allow arbitrary logical expressions to be embedded, recursion, shared state, and asynchronous fair messaging all in one formalism An elementary semantics for sequence diagrams appears in [11]. Wirsing and Knapp [44] have developed tools for generating formal executable specifications from Jacobson's interaction diagrams (an ancestor of sequence diagrams) extended with formal annotations. In the actor model, event diagrams [21, 24, 12] graphically model scenarios of actor computation by message-passing edges between actors.

SDL (Specification and Description Language) is an ITU standardised language that is mainly used for telecommunication applications. Like specification diagrams, SDL defines both a graphical and a textual representation form, with message passing an important aspect of the specification. Following the evolution of SDL a number of formal semantics have been developed (see [17, 18]). SDL supports specification of timing properties, but with respect to control and logical properties it is less expressive than specification diagrams. There are a number of tools for simulation of SDL specifications and for code generation, something currently missing for specification diagrams.

Specification diagrams also share commonalities with other approaches to precise specification, in particular with process algebras such as CSP [26, 35], CCS [32], and the π-calculus [33]. A number of full specification languages based on process algebra have been developed; examples include LOTOS [10], which is based on CSP; it is now an an ISO standard. Process algebras including CSP do not generally address the issue of fairness. One consequence of this is $\mathtt{rec}\,X.X$ is equivalent to $\mathtt{eod}$ in specification diagrams but not in CSP since the former can starve other processes.

CSP processes have several different semantics (cf. [35]). The traces model $\mathcal{T}(P)$ of a process P is as the name suggests the set of sequences of action labels of possible computations of P. The traces model is simple and useful for a number of analyses. However, the traces model does not distinguish between the CSP internal and external choice operators, which produce different processes. This is because traces do not express what a process cannot do. The failures model $\mathcal{F}(P)$, which augments each trace with the sets of actions that a process can refuse to do, makes the desired distinction. The internal-external choice distinction is not a natural distinction in the actor model (see § 4.3.3).

There is one more problem that must be solved before a suitable process semantics for CSP is achieved. Namely, a divergent subprocess (one with an infinite sequence of silent transitions) can make a whole CSP process diverge, preventing other subprocesses from communicating. The failures divergences model $\mathcal{N}(P)$ extends traces leading to a state that could diverge with all possible behaviors from that point, expressing the view that if divergence is possible, then there is nothing useful to say about the process from that point on. In the actor model, divergence is not a semantic problem. A divergent actor or group of actors might slow things down, but because of the fairness assumption they will not affect the interaction semantics of the component that they belong to.

Hoare [26, 35] defines a satisfaction relation for CSP processes as follows: Let R be a predicate on traces; then, $P \models R$ just if for every trace tr of P ($tr \in \mathcal{T}(P)$), $R(tr)$ holds. This has a strong analogy to how an **SD** satisfies a mathematical specification. Let M be a predicate on interaction paths with interface (ρ, χ), then a **SD** with interface (ρ, χ) satisfies M just if M holds for each of its interaction paths.

In [35] CSP refinement is characterized algebraically by the requirement that P is refined by Q (written $P \sqsubseteq Q$) just if $P \sqcap Q = P$, where $\sqcap$ is the CSP non-deterministic (internal) choice operator. This gives different notions of refinement depending on the chosen notion of equality, that is, on the choice of process semantics [35]. CSP's notion of refinement is analogous to our interaction refinement relation: supposing P and Q were **SD**s, our analogous relation is $Q \overset{i}{\subseteq} P$ (note the subset direction is opposite in the two formalisms: refinement produces fewer paths). However, interaction refinement has no similar algebraic characterization.

CSP Refinement is transitive and compositional in the sense that if P is refined by Q and $E[X]$ is a process term with a place holder X for processes, then $E[P]$ is refined by $E[Q]$. An actor component algebra [41] has parallel composition, hiding, and renaming as primitive operations, and using these operations we can define component contexts analogous to CSP process contexts such as E above. The compositionality properties of CSP refinement above hold for interaction refinement in any actor component algebra, and in particular hold for **SD**s since **SD**s form a component algebra. So there is also a CSP-**SD** analogy here.

An important property of failures refinement is that it reduces internal choices but preserves external choice. The specification diagram analog of external choice (see § 4.3.3) comes from the ability to express constraints on the environment and interaction refinement for specification diagrams may produce further constraints on the environment. However, such diagrams do not represent actor system behaviors, since as discussed in § 4.3.2 an actor system can not refuse messages coming from the environment. The interaction refinement relation on actor system components will have the property that the environment is not further constrained as far as messages to visible actors are concerned, while the variety of replies may be reduced. Thus on actor system behaviors, interaction refinement enjoys an analogous property.

Temporal logic formulae have been extensively used as a means for logical specification of concurrent and distributed systems [30, 29]. Recently temporal logics for distributed object based systems have been developed [16, 14]. While such logics 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. Logics are additionally very useful for expressing properties once the system is specified. Specification diagrams themselves can serve the purpose of a logic by directly expressing safety and liveness properties, as was illustrated by the examples of Section 3. The use of embedded non-computational assertions is very similar to forms found in Dijkstra-style weakest precondition logics for non-concurrent programs. The manner in which computations may be "cancelled" in the middle of computing via `constrain` is similar to the behavior of the **assume** ("miraculous" or "partial") command introduced by Nelson and others [34]. Predicates that impose requirements, $\mathtt{assert}(\phi)$ in specification diagrams, correspond to the Dijkstra calculus **assert** command.

Finite automata are useful for formally specifying systems which have a strong state-based behavior. They lack expressivity, but partly make up for this lack by their amenability to automatic verification by state-space search techniques. The Statecharts automata formalism [23] has become particularly popular in industry. The primary weakness of finite automata is that a complex software system may not have a meaningful global state, and properties of such systems are often more naturally expressed in terms of events and relations on events.

Concluding Remarks

Formal specification of protocols is an important research topic due to the critical nature of network protocols as well as the significant chance of design errors arising in these protocols. Formal languages to analyze protocols which can be placed in the hands of practitioners will help increase understanding and reliability of designs. Our language has a strong formal basis, so specifications will not be ambiguous. It is also designed to be usable, the main reason for the graphical notation. Mathematicians may prefer the textual version, but practicing engineers will have a strong preference for the graphical presentation.

For small specifications as presented here, complete formal proofs of properties can be given. For larger specifications, protocols can be completely specified, and more limited aspects of the protocols proven correct by formal reasoning. So, there is a spectrum of ways in which the language may be used, for both small and large specifications, and formal and semi-formal reasoning.

We have developed several larger examples to more fully test the theory; these will be published elsewhere. Perhaps the most important future work is development of decision procedures for checking restricted properties of specifications. It would also be interesting to consider developing a real-time version of **SD**.

Acknowledgements

Partial funding for the first author was provided by NSF grants CCR-9619843 and CCR-9988491. Partial funding for the second author was provided by ONR grant N00014-94-1-0857, NSF grant CRR-9633419, ONR N00012-99-C-0198, NSF CCR-9900326, DARPA/Rome Labs grant AF F30602-96-1-0300, DARPA/SRI subcontract 17-000042, DARPA/NASA contract NAS2-98073.

Thanks to Gul Agha, Carlos H.C. Duarte, Ian Mason, Prasanna Thati, and the anonymous referees for many helpful comments on various versions of this paper.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.
2. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.
3. 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.
4. G. Allwein and J. Barwise, editors. *Logical Reasoning With Diagrams*. Oxford University Press, Oxford, 1996.
5. G. Attardi and C. Hewitt. Specifying and proving properites of guardians for distributed systems, 1978.
6. R. J. R. Back. Refinement calculus II: Parallel and reactive systems. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*. Springer–Verlag, 1990.
7. R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13:133–180, 1989.
8. H. G. Baker and C. Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, Aug. 1977.
9. R. Bardohl. Genged - a generic graphical editor for visual languages. In *1998 IEEE Symposium on Visual Languages*, September 1998.

10. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
11. R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. In *ECOOP '97*, volume 1241 of *Lecture Notes in Computer Science*, pages 344–365. Springer-Verlag, 1997.
12. W. D. Clinger. *Foundations of Actor Semantics*. PhD thesis, MIT, 1981. MIT Artificial Intelligence Laboratory AI-TR-633.
13. R. S. Corporation. *UML Notation Guide, version 1.1*. Sept. 1997. Obtained From `http://www.rational.com`.
14. G. Denker. DTL$^+$: A Distributed Temporal Logic Supporting Several Communication Principles. Technical Report , SRI International, Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA 94025, 1998. *To appear*.
15. E. Dijkstra and C. Scholten. *Predicate Calculus and Program Semantics*, volume 14 of *Texts and Monographs in Computer Science*. Springer-Verlag, 1990.
16. C. H. C. Duarte. A proof-theoretic approach to the design of object-based mobility. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems, Volume 2*, pages 37–53. Chapman & Hall, 1997.
17. J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL—Formal Object-oriented Language for Communication Systems*. Prentice Hall, 1997.
18. R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz. On the formal semantics of sdl-2000: a compilation approach based on an abstract sdl machine. In *International Workshop on Abstract State Machines (ASM'2000)*, volume 1912 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
19. M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.
20. S. Frølund. *Coordinated Distributed Objects: An Actor Based Approach to Synchronization*. MIT Press, 1996.
21. I. Greif. Semantics of communicating parallel processes. Technical Report 154, MIT, Project MAC, 1975.
22. E. Gunter, A. Muscholl, and D. Peled. Compositional message sequence charts. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 496–511. Springer, Apr. 2001.
23. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
24. C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.
25. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of 1973 International Joint Conference on Artificial Intelligence*, pages 235–245, Aug. 1973.
26. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
27. Message Sequence Chart (MSC). ITU-T Recommendation Z.120, International Telecommunications Union, Nov. 1996.
28. ITU-T. Revised Recommendation Z. 100 Specification and Description Language (SDL), May 1994. Addendum 1996.
29. L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.
30. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1992.
31. I. A. Mason and C. L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 220:409 – 467, 1999.

32. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.
33. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
34. G. Nelson. A generalization of dijkstra's calculus. *TOPLAS*, 11:517–561, 1987.
35. A. Roscoe. *The Theory and Practice of Concurrency*. Prentice–Hall, 1998.
36. J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
37. V. A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA, 1993.
38. S. Smith. On specification diagrams for actor systems. In C. T. A. Gordon, A. Pitts, editor, *Proceedings of the Second Workshop on Higher-Order Techniques in Semantics*, Electronic Notes in Theoretical Computer Science. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume10.html`.
39. S. Smith and C. Talcott. Specification diagrams for actor systems. In *Formal Methods in Object-Oriented Distributed Systems (FMOODS)*. Kluwer Academic Publishers, 1999.
40. C. L. Talcott. Interaction semantics for components of distributed systems. In E. Najm and J.-B. Stefani, editors, *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96*, 1996. Proceedings published in 1997 by Chapman & Hall.
41. C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.
42. C. L. Talcott. Actor theories in rewriting logic, 1999. submitted for publication.
43. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 675–788. North-Holland, Amsterdam, 1990.
44. M. Wirsing and A. Knapp. A formal approach to object oriented software engineering. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and Its Applications*, number 4 in Electronic Notes in Theoretical Computer Science. Elsevier, 1996. URL: `http://www.elsevier.nl/locate/entcs/volume4.html`.