Actor Languages

Their Syntax, Semantics, Translation, and Equivalence

Ian A. Mason
University of New England
iam@turing.une.edu.au

Carolyn L. Talcott
Stanford University
clt@sail.stanford.edu

Abstract

In this paper we present two actor languages and a semantics preserving translation between them. The source of the translation is a high-level language that provides object-based programming abstractions. The target is a simple functional language extended with basic primitives for actor computation. The semantics preserved is the interaction semantics of actor systems—sets of possible interactions of a system with its environment. The proof itself is of interest since it demonstrates a methodology based on the actor theory framework for reasoning about correctness of transformations and translations of actor programs and languages and more generally of concurrent object languages.

Contents

1 Introduction

In this paper we continue our investigation of the actor model of computation [Hew77, Agh86, Agh90, AMST97, Tal96b, Tal96a]. Actors are independent computational agents that interact solely via *asynchronous* message passing. An actor can create other actors; send messages; and modify its own local state. An actor can only effect the local state of other actors by sending them messages, and it can only send messages to its acquaintances – addresses of actors it was given upon creation, it received in a message, or that it created. Actor semantics requires computations to be fair.

We take two views of actors: as individuals and as elements of components. Individual actors provide units of encapsulation of state and control. Components are collections of actors (and messages) provided with an interface specifying the receptionists (internal actors accessible from outside the component) and external actors (accessible from but not existing inside the component). Collecting actors into components provides for composability and coordination. Individual actors are described in terms of local transitions. Components are described in terms of interactions with their environment.

The actor model provides a natural framework for inter-operation of multiple languages since the details of the code describing an individual actors behavior are not visible outside that actor. All that needs to be common is the messages communicated among the different actors. In [Tal96b], this intuition is formalized using the notion of an *abstract actor structure*. Here we generalize the notion of an abstract actor structure to an *actor theory*. Actor theories provide a general semantic framework for specifying and reasoning about actor systems as well as for reasoning about relations between different actor languages. An actor theory describes the behavior of individual actors. The models of an actor theory account directly for the interaction (exchange of messages) of an actor component with its environment. Each model of an actor theory gives rise to a corresponding semantics of actor components. Two important models are: computation paths—analogous to labelled transition system semantics; and interaction paths—obtained from computation paths by omitting details of internal computation. These give rise to computation path and interaction semantics, respectively. Both semantics are composable and as we will see, interaction semantics provides a basis for specifying and reasoning about actor systems that is independent of any particular choice of actor language.

In this paper we illustrate the ideas and techniques based on actor theories by showing how they can be used to establish the correctness of a translation from a high-level actor language to low-level actor language such as might be found in a compiler preprocessor. The low-level kernel language, ${}^{\mathrm{k}}\mathcal{L}$, is an extension of a simple functional language based on the call-by-value λ-calculus with primitives for actor computation. The high-level user language, ${}^{\mathrm{u}}\mathcal{L}$, provides object-based syntax. To illustrate the techniques involved we have chosen a small number of high-level abstractions: methods; synchronization constraints; and remote procedure call (rpc) method invocation. Actors *qua* objects have methods and synchronization constraints. The language provides the basic object primitives for object creation and synchronous method invocation (requesting that an object carry out an operation using the specified method and subsequently waiting for a reply). Concurrency is achieved through asynchronous method invocation that allows multiple objects to execute concurrently and independently. Synchronization constraints allow an object to control when a method can be invoked.

Each of the languages is given a semantics by defining a corresponding actor theory. We choose to give a separate semantics for the user language in order to be able to reason directly about user programs. The correctness theorem shows that we can also reason about user programs by translating to the kernel language and reasoning in terms of the kernel semantics.

The translation, $u2k$, from the user language to the kernel language eliminates the object-based programming abstractions in favor of the simple actor primitives. Method dispatch translates straightforwardly into conditionals. Synchronization constraints are translated using an auxiliary actor to manage the internal mail queue. Synchronous method invocation is translated using an auxiliary freshly created actor to accept the reply. This corresponds to the use of a private channel for communication. In the translation of both constraints and remote procedure calls it is necessary to refer to the remainder of the local computation. Rather than use a continuation passing transformation we add a control operation `clc` to the kernel language. This could eliminated by further application of the same ideas (c.f. [Ama94]).

The main result presented here is that the translation, $u2k$ preserves the interaction semantics.

Theorem ($u2k$):

$$Isem({}^{\mathrm{u}}P) = Isem(u2k({}^{\mathrm{u}}P))\lceil {}^{\mathrm{u}}\mathbf{M}$$

where ${}^{\mathrm{u}}P$ is a user language program, $Isem$ maps programs to their interaction semantics, and $\lceil {}^{\mathrm{u}}\mathbf{M}$ restricts the kernel interactions to user language messages.

The proof that the translation preserves interaction semantics itself is of interest since it demonstrates a methodology for proving correctness of transformations and translations of actor languages and more generally of concurrent object languages. For the proof we lift the translation to semantic configurations that correspond to the possible actor system states and show that the following diagram commutes

$$\begin{array}{ccc} {}^{\mathrm{u}}P & \xrightarrow{u2k} & {}^{\mathrm{k}}P \\ [\![\_]\!]\downarrow & & \downarrow\, [\![\_]\!] \\ {}^{\mathrm{u}}K & \xrightarrow{u2k} & {}^{\mathrm{k}}K \end{array}$$

where P is a top-level program, K, is a configuration, and $[\![\_]\!]$ gives the semantics of a program in terms of the initial configuration that it describes. [1] The proof is completed by showing that interaction semantics is preserved by translation at the semantic level

$$Isem({}^{\mathrm{u}}K) = Isem(u2k({}^{\mathrm{u}}K))\lceil {}^{\mathrm{u}}\mathbf{M}.$$

This proof involves establishing a correspondence between the (possibly infinite) computations of two systems. The actor theories defined for each of the languages correspond to standard transition system semantics with transitions that are small and easy to understand, but expose much irrelevant detail. We make use of a general interaction semantics preserving actor theory transformation that can be thought of as moving from a small step a big step operational semantics. Changing the level of abstraction of the operational semantics of a fixed language is a general technique useful for reasoning about systems at the desired level of detail. Reasoning about the level changing transformation on actor theories and the language changing translation is simplified by using ideas from the rewriting logic model of concurrent computation [Mes92, Mes96, Tal96a] to define notions of computation path equivalence.

Plan

The remainder of this paper is organized as follows. In §2 we briefly review the actor theory framework for actor system semantics and define the small-step to big-step actor theory transformation. The kernel-language syntax and semantics is presented in §3 and the user-language syntax and semantics is presented in §4. The user to kernel translation and the proof of its correctness are presented in §5. §6 contains some concluding remarks.

Notation

We use the usual notation for set membership and function application. $\mathbf{Z}$ is the set of integers, $\mathbf{N}$ is the set of natural numbers. Let Y, Y_0, Y_1 be sets. We specify meta-variable conventions in the form: let y range over Y, which should be read as: the meta-variable y and decorated variants such as y', y_0, ..., range over the set Y. Y^n is the set of sequences of elements of Y of length n. Y^* is the set of finite sequences of elements of Y, and Y^+ is the set of non-empty sequences. $\bar{y} = [y_1, \ldots, y_n]$ is the sequence of length $\mathrm{Len}(\bar{y}) = n$ with ith element y_i. (Thus $[\,]$ is the empty sequence.) $u * v$ denotes the concatenation of the sequences u and v. $\mathcal{P}_\omega[Y]$ is the set of finite subsets of Y. $\mathcal{M}_\omega[Y]$ is the set of (finite) multi-sets with elements in Y. $\emptyset$ is the empty multiset and if X_1 and X_2 are multisets, then $X_0\ ,\ X_1$ is the multiset union of the two. $\mathrm{Fmap}[Y_0, Y_1]$ is the set of finite maps from Y_0 to Y_1. $[Y_0 \to Y_1]$ is the set of total functions, f, with domain Y_0 and range contained in Y_1. $[Y_0 \stackrel{\circ}{\to} Y_1]$ is the set of partial functions, f, with domain contained in Y_0 and range contained in Y_1. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. For any function f: $f\{y := y'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \mathrm{Dom}(f)$; and $f\lceil Y$ is the restriction of f to the set Y.

2 A Semantic Framework for Actors

In this section we introduce *actor theories* as a general semantic framework for specifying and reasoning about actor computation. The notion of actor theory provides an axiomatic characterization of actor languages: the basic features; capabilities; and constraints. Actor theories can be considered as an operational alternative to the domain theoretic

[1] We use the following convention: if X is a meta variable for some sort common to the user and kernel language, then we use the super-prescript ${}^{\mathrm{u}}X$ to indicate that X belongs to the user language and ${}^{\mathrm{k}}X$ to indicate that X belongs to the kernel language. So for example ${}^{\mathrm{u}}K$ is an user language configuration. However to prevent a notational quagmire we use this convention sparingly.

behaviors used by Clinger [Cli81]. They are a simplification and generalization of the notion of abstract actor structures presented in [Tal96b, Tal96a].

2.1 Actor Theories and Their Semantics– an Introduction

An actor theory describes individual actor behaviors and their local interactions in a representation independent manner. An actor theory specifies sets of *actor names*, *actor states*, *message contents*, and *labelled reaction rules*. Actor names are the means of uniquely identifying individual actors. Actor states are intended to carry information traditionally contained in the script (methods) and acquaintances (values of instance variables), as well as the local message queue and the current processing state. Message contents represent the information that can be communicated between actors, both locally and as interactions with the environment. Reaction rules determine what an actor in a given state can do next and how it will respond to messages with given contents. More generally reaction rules describe synchronous interactions of groups of actors and messages. Reaction rules are labelled. These labels are used in deriving a labelled transition system semantics. In this way the labels provide information concerning the basic observations that can be made as an actor system evolves.

An actor theory must obey the fundamental acquaintance (locality) laws of actors [BH77, Cli81] in addition to renaming laws that express the fact that computation is uniformly parameterized in the choice of actor names—renaming commutes with everything. To state these laws an actor theory also provides a primitive operation to determine the acquaintances of (actor names occurring in) the various entities and a primitive operation to rename them.

The operational semantics of an actor theory is given by the transition relation on configurations derived from the reaction rules. A configuration can be thought of as representing a global snapshot of an actor system with respect to some idealized observer [Agh86]. It contains a set of receptionist names, a set of external actor names, and a collection of actors and messages. The sets of receptionist names and external actor names are the interface of an actor configuration to its environment. They specify what internal actors are visible from the environment, and what actor connections must be provided for the configuration to function. Both the set of receptionist names and the set of external actor names may grow as the configuration evolves. The collection of actors and messages is the *interior* of the configuration. It specifies the internal actors and their current states, and the state of the internal message system. Configurations evolve either by internal computation or by interaction with the environment. The transition relation expresses the ways a configuration might evolve and interact with its environment. The computation path semantics of a configuration is the set of fair computations possible starting with that configuration.

Interaction semantics gives a more abstract view of an actor system, specifying only the possible interactions (patterns of message passing) a system can have with its environment. Interaction semantics is the result of hiding all information concerning the internal computations and what actors may be present beyond the receptionists.

In their full generality actor theories allow for modeling of what might be considered unactor like behavior. For example particular actor theories allow for direct expression of synchronization between two or more actors. The point is that some actor theories are intended to model basic actor computation while others model higher-level programming abstractions and still others are intended simply to be descriptions of the interaction semantics of actor systems without a commitment to the details of how the behavior is realized. The use of a generalized actor theory is typically justified by a mapping to a basic actor theory, or showing that the behavior described can be realized by an actor system.

The term *reaction rule* is used here in the same spirit as in the Chemical Abstract Machine [BB92] to indicate local interactions of reactive entities. Actors and messages can be thought of as special kinds of molecules and interiors are like solutions. Actor theories are in fact a special case of rewrite theories and the mechanisms we use to derive the computations of an actor system are based on those of rewriting logic [Mes92].

2.2 Actor Theories Formally

An actor theory, AT, is a structure of the following form:

$$AT = \langle\ \langle \mathbf{A}, \mathbf{S}, \mathbf{M}, \mathbf{L} \rangle,\ \langle acq, \widehat{\ \ } \rangle,\ RR\ \rangle$$

$\mathbf{A}, \mathbf{S}, \mathbf{M}, \mathbf{L}$ are the primitive sorts of AT. $\mathbf{A}$ is a countable set of actor names, $\mathbf{S}$ is a set of actor states, $\mathbf{M}$ is a set of message contents, and $\mathbf{L}$ is a set of labels. From the primitive sorts we form actor entities, $\mathbf{AE}$, messages, $\mathbf{Msg}$, and configuration interiors, $\mathbf{I}$.

$\mathbf{AE} = [\mathbf{S}]_{\mathbf{A}}$—the set of actor entities (briefly actors) (actor entities are simply an actor name paired with an actor state, thus $\mathbf{AE}$ is set theoretically isomorphic to $\mathbf{A} \times \mathbf{S}$).

$\mathbf{Msg} = \mathbf{A} \triangleleft \mathbf{M}$—the set of messages (a message merely consists of an actor address paired with a message contents, thus $\mathbf{Msg}$ is set theoretically isomorphic to $\mathbf{A} \times \mathbf{M}$).

$\mathbf{I} \subset \{\sigma \cup \mu \mid \sigma \in \mathcal{M}_\omega[\mathbf{AE}], \mu \in \mathcal{M}_\omega[\mathbf{Msg}]\}$—the set of configuration interiors (briefly interiors).

We let a range over $\mathbf{A}$, M range over $\mathbf{M}$, s range over $\mathbf{S}$, l range over $\mathbf{L}$, and I range over $\mathbf{I}$. $[s]_a$ is an *actor* with *name*, a, in *state*, s and $a \triangleleft M$ is a *message* with *addressee*, a, and *contents*, M. A configuration interior, I, is the union of a multiset of messages and a multiset of actors in which no two actor entities have the same name:

$$(\forall a \in \mathbf{A})(|\{s \in \mathbf{S} \mid [s]_a \in I\}| \leq 1).$$

This constraint means that the multiset union of two internal configurations I and I' is only defined when the set of names of actor entities occurring in I is disjoint from those of I'.

RR is a set of reaction rules that specify the behavior of individual actors and their synchronization with other internal actors and messages. Elements of RR are triples of the form

$$l : I \Rightarrow I'$$

where l is the rule label, I is rule source, and I' is the rule target.

The primitive operations of AT are: acq and $\widehat{\ }$. The acquaintance function, $acq : \mathbf{S} \cup \mathbf{M} \cup \mathbf{L} \to \mathcal{P}_\omega[\mathbf{A}]$, gives the finite set of actor names occurring in a state, message contents, or label. acq extends homomorphically to structures built from the primitive sorts. Thus

$$acq([s]_a) = \{a\} \cup acq(s) \qquad acq(a \triangleleft M) = \{a\} \cup acq(M) \qquad acq(I_0\ ,\ I_1) = acq(I_0) \cup acq(I_1)$$

Actor addresses cannot be explicitly created by actors, and the semantics cannot depend on the particular choice of addresses of a group of actors. A renaming mechanism is used to formulate this requirement. We let $\mathbf{Bij}(\mathbf{A})$ be the set of bijections on $\mathbf{A}$ and let α range over $\mathbf{Bij}(\mathbf{A})$. We call elements of $\mathbf{Bij}(\mathbf{A})$ *renamings*. For any such α, $\widehat{\alpha}$ is the associated renaming function on states, message contents, and labels:

$$\widehat{\alpha} : \mathbf{S} \to \mathbf{S} \qquad \widehat{\alpha} : \mathbf{M} \to \mathbf{M} \qquad \widehat{\alpha} : \mathbf{L} \to \mathbf{L}$$

This is analogous to α-renaming in the λ-calculus. Renaming is extended naturally to structures built from addresses, states, and values. Thus

$$\widehat{\alpha}([s]_a) = [\widehat{\alpha}(s)]_{\alpha(a)} \qquad \widehat{\alpha}(a \triangleleft M) = \alpha(a) \triangleleft \widehat{\alpha}(M) \qquad \widehat{\alpha}(I_0\ ,\ I_1) = \widehat{\alpha}(I_0)\ ,\ \widehat{\alpha}(I_1)$$

We define two functions on interiors: $InAct, ExtAct : \mathbf{I} \to \mathcal{P}_\omega[\mathbf{A}]$. $InAct(I)$ is the set of names of the internal actors of I, and $ExtAct(I)$ is the set of names of external actors referred to in I.

$$InAct(I) = \{a \in \mathbf{A} \mid (\exists s \in \mathbf{S})([s]_a \in I)\} \qquad ExtAct(I) = acq(I) - InAct(I)$$

Note that by definition the external actors $ExtAct(I)$ are disjoint from the internal actors $InAct(I)$. Using $InAct$ we can express the requirement for $I\ ,\ I'$, the multiset union of the actors and messages of I and I', to be defined as $InAct(I) \cap InAct(I') = \emptyset$. In this case we have

$$InAct(I\ ,\ I') = InAct(I) \cup InAct(I')$$

$$ExtAct(I\ ,\ I') = (ExtAct(I) \cup ExtAct(I')) - InAct(I\ ,\ I')$$

The locality and renaming laws that an actor theory must obey are expressed by the axioms (**AR**) and (**RR**).

Acquaintance and Renaming Axioms (AR)

(i) $acq(\widehat{\alpha}(v)) = \alpha(acq(v))$ for $v \in \mathbf{A} \cup \mathbf{M} \cup \mathbf{L}$

(ii) $(\forall a \in acq(v))(\alpha(a) = a) \Rightarrow \widehat{\alpha}(v) = v$ for $v \in \mathbf{A} \cup \mathbf{M} \cup \mathbf{L}$

(iii) $\alpha_0 \widehat{\circ} \alpha_1 = \widehat{\alpha_0} \circ \widehat{\alpha_1}$

where we extend renamings pointwise to sets of actor names. (i) states that renaming commutes with the acquaintance function. (ii) states that if a renaming fixes the acquaintances of an object then applying it does not change the object. (iii) states that the renaming mechanism commutes with function composition. (ii-iii) imply that two renamings that agree on the acquaintances of an object have the same result when applied to the object, and that $\widehat{\alpha}$ is a bijection on $\mathbf{A} \cup \mathbf{M} \cup \mathbf{L}$.

Axioms for Reaction rules (RR) If $l : I \Rightarrow I' \in RR$, then
(i) $InAct(I) \neq \emptyset$
(ii) $l : I_0 \Rightarrow I_0' \in RR$ implies $InAct(I) = InAct(I_0)$ and $InAct(I') = InAct(I_0')$
(iii) $InAct(I) \subseteq InAct(I') \subseteq acq(l)$
(iv) $ExtAct(I') \subseteq ExtAct(I)$
(v) $\widehat{\alpha}(l) : \widehat{\alpha}(I) \Rightarrow \widehat{\alpha}(I') \in RR$ for any renaming α in $\mathbf{Bij}(\mathbf{A})$
(i) states that reactions must involve at least one existing actor; (ii) states that a label uniquely determines the actors involved in a reaction; (iii) states that actors cannot disappear and that the actors involved in a reaction must be made explicit as acquaintances of the reaction label; (iv) states that no references to external actors are acquired in an internal transition, although some may be forgotten; and (v) states that the set of rules is closed under renaming.
If $l : I \Rightarrow I' \in RR$, we call $InAct(I)$ the *old* actors of l and $InAct(I') - InAct(I)$ the *new* actors of l.

2.3 Computation Path Semantics

An actor configuration is a configuration interior, I, together with two sets of actor names: the receptionists, ρ, which are a subset of the internal actors of the interior; and the externals, χ, which include all actors mentioned in the interior that are not internal actors.

Definition (Configurations, K):

$$\mathbf{K} = \{ \left\langle\!\!\left\langle\, I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} \;\mid\; \rho \subseteq InAct(I) \;\wedge\; ExtAct(I) \subseteq \chi \;\wedge\; \chi \cap InAct(I) = \emptyset \}$$

We let K range over $\mathbf{K}$.

The computations of a configuration are given by the labelled transition relation with elements of the form $K \xrightarrow{l} K'$. K is the *source* of the transition and K' is the *target* and l is the *label*. Transition labels are either rule labels, input labels, output labels, or a special idle label, `idle`. 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. We now let l range over $\mathbf{L} \cup \mathtt{in}(\mathbf{A} \triangleleft \mathbf{M}) \cup \mathtt{out}(\mathbf{A} \triangleleft \mathbf{M})$.

Definition (Transition rules):

(internal) $\left\langle\!\!\left\langle\, I_0 \;,\; I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} \xrightarrow{l} \left\langle\!\!\left\langle\, I_1 \;,\; I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$ if $l : I_0 \Rightarrow I_1 \in RR$

(in) $\left\langle\!\!\left\langle\, I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} \xrightarrow{\mathtt{in}(a,M)} \left\langle\!\!\left\langle\, I \;,\; a \triangleleft M \,\right\rangle\!\!\right\rangle_{\chi \cup (acq(M) - \rho)}^{\rho}$ if $a \in \rho \;\wedge\; acq(M) \cap InAct(I) \subseteq \rho$

(out) $\left\langle\!\!\left\langle\, I \;,\; a \triangleleft M \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} \xrightarrow{\mathtt{out}(a,M)} \left\langle\!\!\left\langle\, I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho \cup (acq(M) - \chi)}$ if $a \notin InAct(I)$

(idle) $\left\langle\!\!\left\langle\, I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} \xrightarrow{\mathtt{idle}} \left\langle\!\!\left\langle\, I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$

In (**internal**) we assume that the configurations are well-formed (1–4), and that the actors created by the rule l are new (5):

1. $InAct(I_j) \cap InAct(I) = \emptyset$,
2. $\rho \subseteq InAct(I_j \;,\; I)$,
3. $ExtAct(I_j \;,\; I) \subseteq \chi$,
4. $InAct(I_j \;,\; I) \cap \chi = \emptyset$, and

5. $(InAct(I_1) - InAct(I_0)) \cap acq(I) = \emptyset$

A computation path is an infinite sequence of transitions such that target of each transition is the source of the next transition. The computation paths of a configuration, $\mathcal{P}(K)$, are the computation paths whose initial configuration is K.

Definition (Computation Paths, $\mathcal{P}$, $\mathcal{P}(K)$):

$\mathcal{P}$ is the set of sequences, π, of the form $\pi = [K_i \xrightarrow{l_i} K_{i+1} \mid i \in \mathbf{N}]$

$\mathcal{P}(K) = \{\pi \in \mathcal{P} \mid K \text{ is the source of } \pi(0)\}$

A finite computation is a path in which all but a finite number of the transition labels are `idle`. Recall that actor computations are required to be fair. Thus we do not want to consider arbitrary paths, only the fair ones. A computation is fair if whenever a transition is enabled, either it eventually fires or it becomes permanently disabled. We only consider enabledness for transitions whose label is a reaction rule label or an output label. We do not wish to force the environment to do an input, and the idle transitions are simply ignored for the purpose of fairness. We say that a label l is enabled in K if it is in $\mathbf{L} \cup \mathtt{out}(\mathbf{Msg})$ and if K has a transition with label l' where l' is the same as l up to choice of names for new actors. A label, l, is enabled at step i in π, $Enabled(\pi, i, l)$, if l is enabled in the source K_i of $\pi(i)$. l fires in π at j, $Fires(\pi, j, l)$, if $\pi(j)$ has the form $K_j \xrightarrow{l_j} K_{j+1}$ where l_j differs from l only in the names of new actors. $\mathcal{F}(K)$ is the set of fair paths with initial configuration K.

Definition (Fair Paths):

$$Fair(\pi) \Leftrightarrow (\forall i, l)(Enabled(\pi, i, l) \Rightarrow ((\exists j)(Fires(\pi, i+j, l)) \vee (\exists k)(\forall j)(\neg Enabled(\pi, i+k+j, l))))$$

$$\mathcal{F}(K) = \{\pi \in \mathcal{P}(K) \mid Fair(\pi)\}$$

2.4 Interaction semantics

The set of computation paths, $\mathcal{P}(K)$, and the set of fair computation paths, $\mathcal{F}(K)$ both give a compositional semantics for actor systems [Tal96a]. However these semantics contain too much detail about the inner workings of an actor system to yield a useful notion of equivalence (an equivalence is useful if equivalent entities cannot be distinguished by interacting with other actor systems, or – for that matter – any system). In analogy to the idea of a sequential procedure as a black box characterized by its input/output relation, we would like to consider an actor system as a black box characterized by the set of possible interactions with its environment. Thus we define the interaction semantics of an actor system in such a way as to hide the details of internal transitions. It is a composable semantics with many pleasant properties [Tal96b, Tal97, Tal98].

The interaction semantics of a configuration is its set of possible interaction paths. An interaction path of a configuration is an infinite sequence of interaction labels together with an *initial interface* consisting of a pair of finite sets of actor names (the receptionists and externals). An interaction label is either an input/output label or the special sign, τ^*, standing for possible internal activity.

The function $isem$ maps transition labels to interaction labels and computation paths to interaction paths. The receptionists and externals of $isem(\pi)$ are those of the initial configuration of π. The interaction sequence of $isem(\pi)$ is the sequence of labels obtained by replacing the internal and idle transition labels by τ^*.

Definition ($isem(\pi)$ $Isem(K)$):

$$isem(l) = \begin{cases} \tau^* & \text{if } l \in \mathbf{L} \cup \{\mathtt{idle}\} \\ l & \text{if } l \in \mathtt{in}(\mathbf{Msg}) \cup \mathtt{out}(\mathbf{Msg}) \end{cases}$$

$$isem(\pi) = \vartheta^{\rho_0}_{\chi_0} \quad \text{where} \quad \pi(i) = \left\langle\!\!\left\langle I_i \right\rangle\!\!\right\rangle^{\rho_i}_{\chi_i} \xrightarrow{l_i} \left\langle\!\!\left\langle I_{i+1} \right\rangle\!\!\right\rangle^{\rho_{i+1}}_{\chi_{i+1}} \quad \text{and} \quad \vartheta(i) = isem(l_i) \quad \text{for} \quad i \in \mathbf{N}$$

$$Isem(K) = \{isem(\pi) \mid \pi \in \mathcal{F}(K)\}$$

We say that two configurations are *interaction equivalent* if they have the same interaction semantics.

So far we have been working in the context of a particular fixed actor theory. In the case that we consider interaction semantics in more than one actor theory, we qualify the configuration K by the name of the actor theory, writing

$Isem(K : AT)$ for example. Thus we can express the interaction equivalence of configurations in different actor theories (with the same messages) by writing $Isem(K_0 : AT_0) = Isem(K_1 : AT_1)$. Idle transitions are included as a technical convenience. In what follows we consider two interaction paths to be the same if the differ only by insertion or deletion of τ^* interactions.

2.5 Constrained Actor Theories and the Big-Step Transform

For reasoning about semantics of actor programs or configurations we generalize the notion of actor theory further by replacing the fixed set of fair computations by a set of admissible computations, $\mathcal{A}$, specified as a part of the theory. We call these *constrained actor theories*. In an constrained actor theory the set of admissible computations replaces the set of fair computations in the definition of interaction semantics of a configuration:

$$Isem(K : AT) = \{isem(\pi) \mid \pi \in \mathcal{A}\}$$

A standard actory theory can be thought of as an constrained actor theory by defining $\mathcal{A} = \mathcal{F}$.

An example of the use of constrained actor theories is to be able to compare configurations in two theories with differing messages by restricting attention to the set of messages that they have in common. This can be done by a message restriction theory transformation. Let AT be an actor theory with messages $\mathbf{M}$ and admissible paths $\mathcal{A}$. For $V \subset \mathbf{M}$, we define $AT\lceil V$ to be the actor theory with the same underlying structure as AT ($\langle\ \langle \mathbf{A}, \mathbf{S}, \mathbf{M}, \mathbf{L}\rangle,\ \langle acq, \hat{\ }\rangle,\ RR\ \rangle$) and with admissible paths, $\mathcal{A}\lceil V$, consisting of those paths with input restricted to messages in V.

$$\mathcal{A}\lceil V = \{\pi = [K_i \xrightarrow{l_i} K_{i+1} \mid i \in \mathbf{N}] \mid \pi \in \mathcal{A} \wedge (\forall i \in \mathbf{N})(l_i \in \{\mathtt{idle}\} \cup \mathbf{L} \cup \mathtt{in}(\mathbf{A} \triangleleft V) \cup \mathtt{out}(\mathbf{A} \triangleleft \mathbf{M}))\}$$

Working in the context of a fixed actor theory AT, we define $Isem(K)\lceil V$ to be $Isem(K : AT\lceil V)$.

These constrained actor theories should be thought of not as different models of actor computation, but as more convenient descriptions of the interaction semantics. If AT is a standard actor theory and $AT^\dagger$ is an constrained variant of AT (same basic ingredients) with admissible paths $\mathcal{A}$ then using $AT^\dagger$ in place of AT is typically justified by an equivalence theorem of the form

$$(\forall \pi \in \mathcal{F})(\exists \pi' \in \mathcal{A})(isem(\pi) = isem(\pi') \wedge \text{source}(\pi(0)) = \text{source}(\pi'(0)))$$

and

$$(\forall \pi \in \mathcal{A})(\exists \pi' \in \mathcal{F})(isem(\pi) = isem(\pi')) \wedge \text{source}(\pi(0)) = \text{source}(\pi'(0))$$

Thus for any AT configuration (equivalently any $AT^\dagger$ configuration)

$$Isem(K : AT) = Isem(K : AT^\dagger).$$

More generally we can use this kind of equivalence to justify moving from one constrained actor theory to another.

As a simple example of a semantics preserving transformation, we define the restriction of an actor theory to a subset of its configurations. Let AT be an actor theory with configurations $\mathbf{K}$ and let $\mathbf{K}_0 \subset \mathbf{K}$. Then $AT\lceil \mathbf{K}_0$ is the actor theory with the same underlying structure as AT and with admissible paths $\mathcal{A}\lceil \mathbf{K}_0$ the admissible paths of AT with initial configuration in $\mathbf{K}_0$.

$$\mathcal{A}\lceil \mathbf{K}_0 = \{\pi \in \mathcal{A} : AT \mid \text{source}(\pi(0)) \in \mathbf{K}_0\}$$

Clearly this restriction preserves interaction semantics on configurations in $\mathbf{K}_0$.

Lemma (cfig.restr): For $K \in \mathbf{K}_0$,

$$Isem(K : AT) = Isem(K : (AT\lceil \mathbf{K}_0))$$

A more substantial example of moving from one theory to another while preserving semantics is the restriction of attention to computations having some particular canonical form. To illustrate the idea we define the notion of *big-step form* and the *big-step transformation* that restricts attention to computations in *big-step form*. The big-step transformation is interesting in the context of using actor theories to define the semantics of actor programming languages,

since the semantics is usually defined by giving a reaction rule for each construct of the language. Such a semantics is simple to define and easy to understand *in the small*. However it gives rise to computations with many small and mostly uninteresting steps, making reasoning *in the large* unnecessarily complicated. The big-step transformation allows one to think of computations in terms of the interesting steps that involve interactions with the environment, creation of actors and messages, or that involve some synchronization of actors and messages, suppressing details of internal computation of an actor.

The transition rules on configurations can be divided into three groups:

1. The *interaction rules*, consisting of the (**in**), (**out**) and (**idle**) rules.
2. The *silent rules*, consisting of the uninteresting internal rules. These rules are those that correspond to a single actor computing, or more accurately changing state.
3. The *non-silent internal rules*, consisting of the internal rules in which more than one actor or message is involved. Typical non-silent steps are the receipt of a message by an actor, the sending of messages, the creation of actors, or some other more elaborate synchronization of actors and messages.

From the interaction point of view, the only steps we are interested in are the *interaction* steps. The silent steps are the least interesting, they merely transform the state of an actor till it is ready to participate in the next non silent step it participates in. The silent steps are the ones that are collapsed in the transformation from small step to big step theories. We also have a fair degree of freedom when it comes to the manipulation of silent moves.

Definition (Silent rule and step): A silent rule is a rule of the form $l : [\, s \,]_a \Rightarrow [\, s' \,]_a$. A silent step at a is a transition whose label is a silent rule involving only a:

$$\left\langle\!\!\left\langle\, I\,,\ [\, s \,]_a \,\right\rangle\!\!\right\rangle^{\rho}_{\chi} \xrightarrow{\ l\ } \left\langle\!\!\left\langle\, I\,,\ [\, s' \,]_a \,\right\rangle\!\!\right\rangle^{\rho}_{\chi}$$

Consider two adjacent steps:

$$K_0 \xrightarrow{\ l_0\ } K_1 \xrightarrow{\ l_1\ } K_2$$

then a simple examination of the definition of the transition rules reveals that these may be commuted whenever: (**1**) the old actors of l_1 are disjoint from the old and new actors of l_0; (**2**) the messages produced in the l_0 rule do not participate in the l_1 rule; and (**3**) l_0 and l_1 are not both interaction labels. If this is the case then there will be a K_1' such that:

$$K_0 \xrightarrow{\ l_1\ } K_1' \xrightarrow{\ l_0\ } K_2$$

Computation paths that differ only by such permutations are said to be equivalent and equivalent paths give rise to the same interaction path. This notion of equivalence is spelled out in more detail in [Tal96a] using the concepts of rewriting logic.

This simple observation has several useful consequences. Suppose that $I_0 \xrightarrow{\ l\ } I_1$ is a rewrite rule. Firstly, since silent moves at an actor a neither produce messages nor consume any messages they may be commuted with any other rule, such as l, that they do not participate in, i.e. $a \notin InAct(I_0)$. Secondly, an idle transition may be commuted with any other transition. Thirdly, an $\mathtt{in}(a, M)$ may be commuted with any internal transition, such as l, that the incoming message does not participate in, i.e. $a \triangleleft M \notin I_0$. Fourthly, an $\mathtt{out}(a, M)$ may be commuted with any internal transition, such as l, except the transitions that the outgoing message participates in (i.e. $a \triangleleft M \in I_1 - I_0$). In defining the big-step form for computations of an actor theory we will mainly be concerned with grouping silent steps into groups of silent step that share a certain purpose.

We begin with some definitions and a lemma about putting computations into a standard form.

Definition (Silent move): A silent move is a sequence of silent steps.

- An actor a in state s silently moves to state s', written

  $$[\, s \,]_a \xrightarrow{\ *\ } [\, s' \,]_a$$

  if there are s_j, l_j, such that $[\, s_j \,]_a \xrightarrow{\ l_j\ } [\, s_{j+1} \,]_a$ with l_j a silent rule $0 \leq j < n$ and $s = s_0$, $s' = s_n$.

- For any set of actors, A, a configuration interior, I, moves silently at A to I' (written $I \xrightarrow{*} I'$) if there are I_j, l_j, $a_j \in A$ such that

$$\left\langle\!\left\langle\, I_j \,\right\rangle\!\right\rangle^{\rho}_{\chi} \xrightarrow{l_j} \left\langle\!\left\langle\, I_{j+1} \,\right\rangle\!\right\rangle^{\rho}_{\chi}$$

  with l_j a silent step at a_j for $0 \leq j < n$, $I = I_0$, and $I' = I_n$. When we write $I \xrightarrow{*} I'$ with A implicit, we take it to be the union of the old actors of the transitions $\{l_j \mid 0 \leq j < n\}$.

Definition (Permanently silent): An actor $[\,s\,]_a$ in a configuration K is *permanently silent* if every sequence of steps (possibly infinite) involving a (starting with a in state s) consists solely of silent steps.
Thus if a has become permanently silent, then any maximal computation sequence involving this actor either reaches a state which has no moves or is an infinite sequence of silent steps.

Definition (Macro label and step): A macro label L is simply a sequence of transition labels $l_0; \ldots; l_k$. Correspondingly a macro step is a sequence of steps that are a segmemt of a computation. We write $K_0 \xrightarrow{L} K_{k+1}$ if $L = l_0; \ldots; l_k$ and $[K_i \xrightarrow{l_i} K_{i+1} \mid 0 \leq i \leq k]$.

Macro labels and steps are simply a convenient notation for grouping steps of a computation path. We are particularly interested in macro steps that we call big steps.

Definition (Big step): A big step is a macro step with label of the form $l_0; \ldots; l_k$ where l_k is a non-silent internal step with old actors A and if $k > 0$, then $\{l_0, \ldots l_{k-1}\}$ are silent steps at A.

Definition (Big step form): We say a path π is in big step form if it can be written using macro steps as

$$\pi = [K_i \xrightarrow{L_i} K_{i+1} \mid i \in \mathbf{N}]$$

where each L_i is either a big step, or a singleton interaction step (i.e. $\mathtt{idle}$, $\mathtt{in}(a, M)$ or $\mathtt{out}(a, M)$). Furthermore we require the path to be *observationally fair*. That is, ignoring actors who have become permanently silent, the path is fair. Thus a computation path π is in big-step form if it is observationally fair and

1. Between any adjacent non-silent internal steps there are only silent steps at old actors of the label of the latter step.
2. The interaction steps between the adjacent non-silent internal steps occur before any of the silent steps.

Lemma (big step form): For every computation path π, there is a path, π' such that

- π' is in big step form,
- $isem(\pi) = isem(\pi')$, and
- if π is fair then π' is observationally fair.

Furthermore, if π' is a path in big step form, then there is a fair path π such that $isem(\pi) = isem(\pi')$.

Proof :

The idea behind the proof is simply to move silent steps (of eventually active actors) forward until they bump into a later silent step at the same actor, or a non-silent step in which they participate. This is possible by the above observation concerning when steps can be commuted. Let $\pi = [K_i \xrightarrow{l_i} K_{i+1} \mid i \in \mathbf{N}]$ be a computation path. Let $N : \mathbf{N} \to \mathbf{N}$ enumerate in order the non-silent internal and interaction steps of π (called the enumerated steps in the following). Thus for $i \in \mathbf{N}$ there is m_i such that

$$K_{N(i-1)+1} \xrightarrow{l_{N(i-1)+1}} \cdots \xrightarrow{l_{N(i-1)+m_i}} K_{N(i)} \xrightarrow{l_{N(i)}} K_{N(i)+1}$$

are the m_i steps leading from the target of $i-1$st enumerated step (or the initial configuration if $N(i) = 0$) to the i^{th} enumerated step (it is quite possible that $m_i = 0$). Finally suppose that the segment from $K_{N(i-1)+1}$ to $K_{N(i)+1}$ is the first segment not in big step form (i.e. is a sequence of steps in which the last is an enumerated step, the first starts with the initial configuration or is the step following the preceding enumerated step, and those in between are silent). We show how to transform it into a path with the same interactions, in which this step is in big step form.

If $K_{N(i)} \xrightarrow{l_{N(i)}} K_{N(i)+1}$ is an interaction step then, since silent steps can be commuted with interaction steps we may move the interaction steps to the front of the sequence.

If $K_{N(i)} \xrightarrow{l_{N(i)}} K_{N(i)+1}$ is a non-silent internal step, then we move all silent steps of actors not participating in this step past this step, preserving their internal order. This is again possible by the above observation on commuting rules. Silent steps commute with silent steps at distinct actors, and with non-silent internal steps that they do not participate in.

The result of theses two steps is a path π' whose first i enumerated steps are in big step form. This process may be repeated indefinitely. Now we need to show that in the limit the resulting path is observationally fair if the original path was fair. This is easy to see by noting that the enabledness conditions on transitions are preserved by the transformation, and that π' has exactly the transitions π omitting the silent steps at actors which have become permanently silent, since these are indefinitely postponed.

Finally suppose that π' is in big step form. If π' is not fair this can only be due to silent steps by actors which have become permanently silent. A fair path with the same interaction path can be obtained by interleaving silent steps of the ignored actors with other steps. □

Definition (Big step transform, $AT^\dagger$): Let AT be a standard actor theory (admissible paths are the fair paths). Then the big step transform, $AT^\dagger$, of AT is obtained from AT by defining the admissible paths $\mathcal{A}^\dagger$ to be those computations of AT that are in big step form.

The crucial property of the big step transformation is that it preserves interaction semantics.

Theorem (big step transform): Let AT be a standard actor theory and let K be a configuration of AT. Then

$$Isem(K : AT) = Isem(K : AT^\dagger)$$

Proof : For $\subseteq$ assume $\zeta \in Isem(K : AT)$. Then $\zeta = isem(\pi)$ for some $\pi \in \mathcal{F}$ and by (**big step form**) there is $\pi' \in \mathcal{A}^\dagger$ such that $isem(\pi') = isem(\pi)$ and hence $\zeta \in Isem(K : AT^\dagger)$. For $\supseteq$ assume $\zeta \in Isem(K : AT^\dagger)$. Then $\zeta = isem(\pi')$ for some $\pi' \in \mathcal{A}^\dagger$ and again by (**big step form**) there is $\pi \in \mathcal{F}$ such that $isem(\pi') = isem(\pi)$. □

3 The Kernel Language

We now introduce our kernel language, presenting first the syntax, then a set of simple but illustrative examples, and finally the semantics given by defining a suitable actor theory and mapping programs to configurations of this actor theory.

3.1 Kernel Syntax

We assume given an infinite set of variables, $\mathbf{X}$. We also assume as given a collection of basic or atomic data, $\mathbf{At}$, that includes the booleans $\mathtt{t}, \mathtt{f} \in \mathbf{Bool}$, Scheme style symbols, $\mathbf{Sym}$, ($\mathbf{Sym}$ includes `nil`, the empty or null list), (constants denoting the elements of) the integers, $\mathbf{Z}$, and actor names, $\mathbf{A}$. Expressions are built from atoms and variables by the following operations: λ-abstraction, application of primitive operations to sequences of expressions, conditional branching, and an actor creation construct. The primitive operations include operations on basic data and pairs, and kernel primitives manipulating actors, procedures, and local continuations. The data operations $\mathbf{dOp}$ contains the recognizers: `boolean`? for booleans, `symbol`? for symbols, `integer`? for integers, `cons`? for pairs, and `actor`? for actors (all of arity 1); pairing `cons`, `car`, `cdr` (arities 2, 1, 1); the equality predicate, `equal`?, on atomic data (arity 2); and the usual arithmetic operations, $\mathbf{aOp}$. We consider actor addresses to be atomic data and consequently can tell one address from another. The functional specific primitives are `procedure`?, the recognizer for procedures (arity 1), `app`, lambda application (arity 2), and `clc`, control abstraction (arity 1). We include `app` in the list of primitive operations as a technical convenience, to make the syntax more concise. The actor primitives consist of an actor creation construct plus the operations: `self` (of arity 0), the name of the executing actor; `send`, asynchronous send (arity 2); `ready`, establishing behavior for receiving (arity 1). Actor creation expressions are of the form `letactor`$\{x_0 := e_0, \ldots x_k := e_k\}$ e where the x_i are pairwise distinct variables. Executing a `letactor`

expression creates a new actor entity a_i for each x_i executing expressions e_i with x_i bound to a_i. The original executing actor then proceeds by executing e (with x_i bound to a_i).

The top level syntactic construct is a *kernel program* which describes a configuration. For convenience, kernel programs may include a library of mutually recursive definitions. For this purpose we reserve a subset $\mathbf{FunId}$ of $\mathbf{X}$ to be used as function names.

Definition (Kernel Programs):

$$\begin{aligned}{}^{\mathrm{k}}\mathbf{Program} = \texttt{program}(&\texttt{receptionists} : \mathcal{P}_\omega[\mathbf{A}], \quad \texttt{externals} : \mathcal{P}_\omega[\mathbf{A}] \\ &\texttt{library} : \mathcal{P}_\omega[\mathbf{FunId} := \lambda\mathbf{X}.\mathbf{E}] \\ &\texttt{actors} : \mathcal{P}_\omega[\mathbf{A} := \mathbf{E}] \\ &\texttt{messages} : \mathcal{M}_\omega[\mathbf{A} \triangleleft {}^{\mathrm{k}}\mathbf{M}])\end{aligned}$$

$${}^{\mathrm{k}}\mathbf{M} = \mathbf{At} \cup \texttt{cons}({}^{\mathrm{k}}\mathbf{M}, {}^{\mathrm{k}}\mathbf{M})$$

where the function identifiers in the `library` part and actor names in the `actors` part must be distinct, and all actor names occurring in an actor state or message contents must either be one of the actor names defined in the `actors` part or one of the names occurring in the `externals` part. Message contents are simply values built up from the atomic data via the pairing operation `cons`. Lambda abstractions and structures containing lambda abstractions are not allowed to be communicated in messages.

Definition (Kernel Expressions):

$$\mathbf{aOp} = \text{arithmetic operations}$$

$$\mathbf{dOp} = \{\texttt{actor?}, \texttt{boolean?}, \texttt{integer?}, \texttt{symbol?}, \texttt{cons?}, \texttt{equal?}, \texttt{cons}, \texttt{car}, \texttt{cdr}\} \cup \mathbf{aOp}$$

$$\mathbf{O} = \mathbf{dOp} \cup \{\texttt{procedure?}, \texttt{app}, \texttt{clc}\} \cup \{\texttt{self}, \texttt{ready}, \texttt{send}\}$$

$$\mathbf{At} = \mathbf{A} \cup \mathbf{Bool} \cup \mathbf{Z} \cup \mathbf{Sym}$$

$$\mathbf{E} = \mathbf{X} \cup \mathbf{At} \cup \lambda\mathbf{X}.\mathbf{E} \cup \mathbf{O}_n(\mathbf{E}_n) \cup \texttt{if}(\mathbf{E}, \mathbf{E}, \mathbf{E}) \cup \texttt{letactor}\{(\mathbf{X} := \mathbf{E})^+\}\mathbf{E}$$

In the definition of $\mathbf{E}$, $\mathbf{O}_n$ refers to the subset of $\mathbf{O}$ consisting of the operators of arity n. We let x, y, z range over $\mathbf{X}$, a ranges over $\mathbf{A}$, e ranges over $\mathbf{E}$, ${}^{\mathrm{k}}M$ ranges over ${}^{\mathrm{k}}\mathbf{M}$. The binding constructs are `letactor` and λ. $\lambda x.e$ binds the variable x in the expression e. $\texttt{letactor}\{\ldots x_i := e_i \ldots\}e$ binds the x_i in each of the e_j, and also in e. Two expressions are considered equal if they are the same up to the renaming of bound variables. For any expression e, we write $\mathrm{FV}(e)$ for the set of free variables of e. We write $e'[\bar{x} := \bar{e}]$ to denote the expression obtained from e' by simultaneously replacing all free occurrences of $\bar{x}$ by $\bar{e}$, avoiding the capture of free variables in $\bar{e}$. (We assume $\bar{x}$ and $\bar{e}$ have the same length.)
We use the following standard abbreviations:

$e_0(e_1)$	$\triangleq$	$\texttt{app}(e_0, e_1)$	
$e_0(e_1, \ldots, e_n)$	$\triangleq$	$\texttt{app}(e_0, e_1)(e_2, \ldots, e_n)$	
$\texttt{let}\{x := e_0\}e_1$	$\triangleq$	$\texttt{app}(\lambda x.e_1, e_0)$	
$\texttt{app}(e_0, e_1, \ldots, e_n)$	$\triangleq$	$\texttt{app}(\ldots\texttt{app}(e_0, e_1), \ldots, e_n)$	
$\texttt{not}(e)$	$\triangleq$	$\texttt{if}(e, \texttt{f}, \texttt{t})$	
$\texttt{or}(e_0, e_1)$	$\triangleq$	$\texttt{if}(e_0, \texttt{t}, e_1)$	
$\texttt{and}(e_0, e_1)$	$\triangleq$	$\texttt{if}(e_0, e_1, \texttt{f})$	
$\texttt{seq}(e)$	$\triangleq$	e	
$\texttt{seq}(e_0, \ldots, e_n)$	$\triangleq$	$\texttt{let}\{z := e_0\}\texttt{seq}(e_1, \ldots, e_n)$	$z \notin \mathrm{FV}(e_i)$ for $i \leq n$
$\texttt{letrec}\{fid_j = \lambda x.e\}_{1 \leq j \leq k}\ e_j$	$\triangleq$	mutual recursive definition	

We also use the following definitions for structuring message contents.

$\texttt{list}_n := \lambda x_1.\lambda x_2.\ldots.\lambda x_n \texttt{cons}(x_1, \texttt{cons}(x_2, \ldots \texttt{cons}(x_n, \texttt{nil})))$

$\texttt{msgMk} := \lambda x_{\text{mid}}.\lambda x_{\text{args}}.\lambda x_{\text{cust}}.\texttt{list}_3(x_{\text{mid}}, x_{\text{args}}, x_{\text{cust}})$

$\texttt{msgMeth} := \lambda x.\texttt{car}(x)$

$\texttt{msgArgs} := \lambda x.\texttt{car}(\texttt{cdr}(x))$

$\texttt{msgCust} := \lambda x.\texttt{car}(\texttt{cdr}(\texttt{cdr}(x)))$

3.2 Example Kernel Programs

To illustrate the kernel language, and in particular the actor primitives, we give two versions of a bounded buffer. To contrast the two languages we also present this example in the user language in §4. The example consists of a filter actor which manages two bounded buffers (of size 100) (a positive buffer and a negative buffer). The filter actor accepts two types of messages:

1. An $\texttt{item}(x)$ message, the filter then places the contents, x, into one of the bounded buffers according to whether or not x satisfies a certain property, Φ.
2. A $\texttt{get}(x_{\text{bool}}, x_{\text{cust}})$ message, the filter then forwards a corresponding $\texttt{get}(x_{\text{cust}})$ to the appropriate buffer, depending on whether or not x_{bool} is true.

The buffers are of interest, since they illustrate the built in features of the user language. To implement them in the kernel language requires some work. We provide two implementations of the buffer. The buffer is simply a bounded queue which accepts `put` and `get` messages. Both versions maintain internal queues of *disabled* put and get messages, and rely on the observation that the put queue should be examined after successfully responding to a get message, and the get queue after responding to a put message. The user language version presented in §4 is substantially simpler since it makes use of the synchronization constraints to implicitly maintain the internal message queue.

The first version, `BoundedBuffer1`, makes use of a recursive call to perform this queue manipulation whereas the second incorporates two additional insights concerning these internal queues.

$\texttt{aFilter} :=$

$\lambda b_{\text{no}}.\lambda b_{\text{yes}}.\lambda m.\texttt{if}(\texttt{equal?}(\texttt{msgMeth}(m), \texttt{item}),$
$\quad \texttt{seq}(\texttt{let}\ \{x := \texttt{msgArgs}(m)\}$
$\quad\quad \texttt{if}(\Phi(x), \texttt{send}(b_{\text{yes}}, \texttt{msgMk}(\texttt{put}, x, \texttt{nil})), \texttt{send}(b_{\text{no}}, \texttt{msgMk}(\texttt{put}, x, \texttt{nil}))),$
$\quad\quad \texttt{ready}(\texttt{aFilter}(b_{\text{no}}, b_{\text{yes}}))),$
$\quad \texttt{if}(\ \texttt{equal?}(\texttt{msgMeth}(m), \texttt{get}),$
$\quad\quad \texttt{seq}(\texttt{if}(\texttt{car}(\texttt{msgArgs}(m)),$
$\quad\quad\quad \texttt{send}(b_{\text{yes}}, \texttt{msgMk}(\texttt{get}, \texttt{nil}, \texttt{msgCust}(m))),$
$\quad\quad\quad \texttt{send}(b_{\text{no}}, \texttt{msgMk}(\texttt{get}, \texttt{nil}, \texttt{msgCust}(m)))),$
$\quad\quad\quad \texttt{ready}(\texttt{aFilter}(b_{\text{no}}, b_{\text{yes}}))),$
$\quad\quad \texttt{ready}(\texttt{aFilter}(b_{\text{no}}, b_{\text{yes}}))))$

$\texttt{BoundedBuffer1} :=$

$\lambda n.\lambda q.\lambda q_{\text{put}}.\lambda q_{\text{get}}.$

$\lambda m.\texttt{if}(\texttt{equal?}(\texttt{msgMeth}(m), \texttt{put}),$
$\quad \texttt{if}(\texttt{equal?}(\texttt{length}(q), n),$
$\quad\quad \texttt{ready}(\texttt{BoundedBuffer1}(n, q, \texttt{append}(q_{\text{put}}, \texttt{list}(m)), q_{\text{get}})),$
$\quad\quad \texttt{if}(\texttt{null?}(q_{\text{get}}),$
$\quad\quad\quad \texttt{ready}(\texttt{BoundedBuffer1}(n, \texttt{cons}(\texttt{car}(\texttt{msgArgs}(m)), q), q_{\text{put}}, q_{\text{get}})),$
$\quad\quad\quad \texttt{BoundedBuffer1}(n, \texttt{cons}(\texttt{car}(\texttt{msgArgs}(m), q), q_{\text{put}}, \texttt{cdr}(q_{\text{get}}))(\texttt{car}(q_{\text{get}}))))$
$\quad \texttt{if}(\texttt{equal?}(\texttt{msgMeth}(m), \texttt{get}),$
$\quad\quad \texttt{if}(\texttt{null?}(q),$
$\quad\quad\quad \texttt{ready}(\texttt{BoundedBuffer1}(n, q, q_{\text{put}}, \texttt{append}(q_{\text{get}}, \texttt{list}(m)))),$
$\quad\quad\quad \texttt{seq}(\ \ \texttt{send}(\texttt{msgCust}(m), \texttt{msgMk}(\texttt{element}, \texttt{car}(q), \texttt{self}())),$
$\quad\quad\quad\quad\ \ \texttt{if}(\texttt{null?}(q_{\text{put}}),$
$\quad\quad\quad\quad\quad \texttt{ready}(\texttt{BoundedBuffer1}(n, \texttt{cdr}(q), q_{\text{put}}, q_{\text{get}})),$
$\quad\quad\quad\quad\quad \texttt{BoundedBuffer1}(n, q, \texttt{cdr}(q), \texttt{cdr}(q_{\text{put}}), q_{\text{get}})(\texttt{car}(q_{\text{put}}))))$
$\quad\quad \texttt{ready}(\texttt{BoundedBuffer1}(n, q, q_{\text{put}}, q_{\text{get}}))))$

A kernel configuration using the buffer is described by following program, abbreviated by ${}^{\text{k}}P_1$.

$$
\begin{aligned}
{}^{\text{k}}P_1 \quad \triangleq \quad & \texttt{program}(\texttt{receptionists}: a_{\text{f}}, \quad \texttt{externals}: \\
& \quad \texttt{library}: \ Lib_1 \\
& \quad \texttt{actors}: \ a_{\text{y}} := \texttt{BoundedBuffer1}(100, \texttt{nil}, \texttt{nil}, \texttt{nil}), \\
& \qquad\qquad\ \ a_{\text{n}} := \texttt{BoundedBuffer1}(100, \texttt{nil}, \texttt{nil}, \texttt{nil}), \\
& \qquad\qquad\ \ a_{\text{f}} := \texttt{aFilter}(a_{\text{n}}, a_{\text{y}}) \\
& \quad \texttt{messages}: \ )
\end{aligned}
$$

The library Lib_1 includes the above definitions of `aFilter` and `BoundedBuffer1`, as well as the standard list operations `length`, `list` and `append`.

The second version, `BoundedBuffer2`, eliminates the recursive calls (not in the scope of a `ready`) by making the following two observations:

1. If the get queue is non-empty, then the buffer queue must be empty. Consequently if the current message is an enabled put message, its contents can be used to reply to the first get query in the queue. If this procedure is adopted, then no other examination of the get queue in necessary.

2. Much the same works for the put queue. If the put queue is non-empty, then the buffer must be at its maximum. Consequently if the current message is an enabled get message, then after successfully replying to this message (exactly one) put message from the put queue can be processed.

This results in the following definition.

$\texttt{BoundedBuffer2} :=$

$\lambda n.\lambda q.\lambda q_{\text{put}}.\lambda q_{\text{get}}.$

$\lambda m.$`if`(`equal?`(`msgMeth`(m), `put`),
`if`(`equal?`(`length`(q), n),
`ready`(`BoundedBuffer2`(n, q, `append`(q_{put}, `list`(m)), q_{get})),
`if`(`null?`(q_{get}),
`ready`(`BoundedBuffer2`(n, `cons`(`car`(`msgArgs`(m)), q), q_{put}, q_{get})),
`seq`(`send`(`msgCust`(`car`(q_{get})), `msgMk`(`element`, `car`(`msgArgs`(m)), `self`())),
`ready`(`BoundedBuffer2`(n, q, q_{put}, `cdr`(q_{get}))))))
`if`(`equal?`(`msgMeth`(m), `get`),
`if`(`null?`(q),
`ready`(`BoundedBuffer2`(n, q, q_{put}, `append`(q_{get}, `list`(m)))),
`seq`(`send`(`msgCust`(m), `msgMk`(`element`, `car`(q), `self`())),
`if`(`null?`(q_{put}),
`ready`(`BoundedBuffer2`(n, `cdr`(q), q_{put}, q_{get})),
`ready`(`BoundedBuffer2`(n, `cons`(`car`(`msgArgs`(`car`(q_{put}))),
`cdr`(q),
`cdr`(q_{put}),
q_{get}))))))
`ready`(`BoundedBuffer2`(n, q, q_{put}, q_{get}))))

A kernel configuration using this modified buffer is described by following program, abbreviated by ${}^{\text{k}}P_2$.

${}^{\text{k}}P_2 \triangleq$ `program`(`receptionists :` a_{f}, `externals :`
`library :` Lib_2
`actors :` $a_{\text{y}} :=$ `BoundedBuffer2`(100, `nil`, `nil`, `nil`),
$a_{\text{n}} :=$ `BoundedBuffer2`(100, `nil`, `nil`, `nil`),
$a_{\text{f}} :=$ `Filter`(a_{n}, a_{y})
`messages` :)

The library Lib_2 includes the above definitions of `aFilter` and `BoundedBuffer2`, as well as the standard list operations `length`, `list` and `append`.

For either version of bounded buffer the interactions have the following properties:

1. Each output is the response to a `get`($x_{\text{bool}}, x_{\text{cust}}$) message and the x_{item} sent in the response is such that $\Phi(x_{\text{item}})$ is true iff x_{bool} is true.
2. There is at most one response to a `get` request.
3. There will be a response to a `get`($x_{\text{bool}}, x_{\text{cust}}$) request if the number of `put` items, x_{item}, such that $\Phi(x_{\text{item}})$ has the value x_{bool} is at least the number of requests.

3.3 The Kernel Language Semantics

As indicated earlier, the semantics is given by defining an actor theory, ${}^{\text{k}}AT$. The only primitive sorts of ${}^{\text{k}}AT$ that remain to be defined are the set of kernel actor theory states, ${}^{\text{k}}\mathbf{S}$, and the labels. We define the states now, and postpone the labels till we define reduction.

${}^{\text{k}}\mathbf{S} = \{e \in \mathbf{E} \mid \text{FV}(e) = \emptyset\}$

The acquaintances of a state (or message contents) are those actor names which textually appear in the expressions involved. Renaming is simply substitution. The meaning of a kernel program is defined to be a configuration of ${}^{\mathrm{k}}AT$ as follows.

Definition ($[\![{}^{\mathrm{k}}P]\!]$ $Isem({}^{\mathrm{k}}P)$): Let ${}^{\mathrm{k}}P$ be given by

$$\begin{array}{rcl} {}^{\mathrm{k}}P & \triangleq & \texttt{program}(\texttt{receptionists} : \rho, \quad \texttt{externals} : \chi \\ & & \qquad \texttt{library} : \{{}^{\mathrm{k}}\mathit{fid}_j = \lambda x.e^j\}_{1\le j\le l} \\ & & \qquad \texttt{actors} : \{a_j := e_j\}_{1\le j\le m} \\ & & \qquad \texttt{messages} : \{a'_j \lhd {}^{\mathrm{k}}M_j\}_{1\le j\le n}) \end{array}$$

then

$$[\![{}^{\mathrm{k}}P]\!] = \left\langle\!\!\left\langle\, {}^{\mathrm{k}}I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$$

where

$${}^{\mathrm{k}}I = \{[\,e_j\,]_{a_j}\}_{1\le j\le m}\ ,\ \{a'_j \lhd {}^{\mathrm{k}}M_j\}_{1\le j\le n}$$

and

$$Isem({}^{\mathrm{k}}P) = Isem([\![{}^{\mathrm{k}}P]\!] : {}^{\mathrm{k}}AT)$$

where ${}^{\mathrm{k}}AT$ incorporates the specific library $\{{}^{\mathrm{k}}\mathit{fid}_j = \lambda x.e^j\}_{1\le j\le l}$. For the program, and hence the configuration, to be well formed we require that $\mathrm{FV}(e_j) \subseteq \{{}^{\mathrm{k}}\mathit{fid}_1, \ldots, {}^{\mathrm{k}}\mathit{fid}_l\}$.

To complete the semantics all that remains is to define the reaction rules. To do this we decompose each non-value expression as a reduction context filled with a redex. Reduction contexts identify the subexpression of an expression that is to be evaluated next, they correspond to the standard reduction strategy (left-first, call-by-value) of [Plo75] and were first introduced in [FF86]. An expression e is either a value or it can be decomposed uniquely into a reduction context filled with a redex. Thus, local actor computation is deterministic.

Definition ($\mathbf{V}$ $\mathbf{E}_{\mathrm{rdx}}$ $\mathbf{R}$): The set of *values*, $\mathbf{V}$, the set of *redexes*, $\mathbf{E}_{\mathrm{rdx}}$, and the set of *reduction contexts*, $\mathbf{R}$, are defined by

$$\mathbf{V} = \mathbf{At} \cup \texttt{cons}(\mathbf{V}, \mathbf{V}) \cup \lambda\mathbf{X}.\mathbf{E}$$

$$\mathbf{E}_{\mathrm{rdx}} = (\mathbf{O}_n(\mathbf{V}^n) - \texttt{cons}(\mathbf{V}, \mathbf{V})) \cup \texttt{if}(\mathbf{V}, \mathbf{E}, \mathbf{E}) \cup \texttt{letactor}\{(\mathbf{X} := \mathbf{E})^+\}\mathbf{E}$$

$$\mathbf{R} = \{\bullet\} \cup \mathbf{O}_{n+m+1}(\mathbf{V}^n, \mathbf{R}, \mathbf{E}^m) \cup \texttt{if}(\mathbf{R}, \mathbf{E}, \mathbf{E})$$

We let R range over $\mathbf{R}$. With the exception of the actor primitives `letactor`, `send`, and `ready`, reduction steps are silent – they only depend on information local to the executing actor and only effect the state of the executing actor. Thus we define a sequential reduction relation, $e \xrightarrow{s}_{{}^{\mathrm{k}}\zeta} e'$, on expressions that lifts uniformly to define the silent reaction rules. The decoration ${}^{\mathrm{k}}\zeta$ is an abstract context introduced to make the dependence on local context explicit. For the purposes of the kernel language the only contextual piece of information we need to assume is encoded in ${}^{\mathrm{k}}\zeta$ is the name of the actor executing the expression. We use a function $\mathit{self}({}^{\mathrm{k}}\zeta)$ that extracts the name of the executing actor from ${}^{\mathrm{k}}\zeta$. Thus to be concrete we could identify ${}^{\mathrm{k}}\zeta$ with the name of an actor. In the user language more contextual information is required and hence in the user case such a simplification is not possible. To define the sequential relation, we first define the purely functional reduction relation $r \xrightarrow{f}_{{}^{\mathrm{k}}\zeta} e$ which gives the rules for redexes that do not manipulate the reduction context.

Definition (Functional rules: $e \xrightarrow{f}_{{}^{\mathrm{k}}\zeta} e'$):

(op) $\quad \mathit{op}(\bar{v}) \xrightarrow{f}_{{}^{\mathrm{k}}\zeta} v$ if op is an arithmetic operation and $\mathit{op}(\bar{v}) = v$

(fun) $\quad \mathit{fid}(v) \xrightarrow{f}_{{}^{\mathrm{u}}\zeta} e[x := v] \qquad$ if $\mathit{fid} := \lambda x.e \in \mathit{Lib}$

(self) $\quad \texttt{self}() \xrightarrow{f}_{{}^{\mathrm{k}}\zeta} a$ if $\mathit{self}({}^{\mathrm{k}}\zeta) = a$

(if) $\texttt{if}(v, e_1, e_2) \xrightarrow{f}_{\mathrm{k}\zeta} \begin{cases} e_1 & \text{if } v \neq \texttt{f} \\ e_2 & \text{if } v = \texttt{f} \end{cases}$

(bool?) $\texttt{boolean?}(v) \xrightarrow{f}_{\mathrm{k}\zeta} \begin{cases} \texttt{t} & \text{if } v \in \mathbf{Bool} \\ \texttt{f} & \text{if } v \notin \mathbf{Bool} \end{cases}$

(sym?) $\texttt{symbol?}(v) \xrightarrow{f}_{\mathrm{k}\zeta} \begin{cases} \texttt{t} & \text{if } v \in \mathbf{Sym} \\ \texttt{f} & \text{if } v \notin \mathbf{Sym} \end{cases}$

(int?) $\texttt{integer?}(v) \xrightarrow{f}_{\mathrm{k}\zeta} \begin{cases} \texttt{t} & \text{if } v \in \mathbf{Z} \\ \texttt{f} & \text{if } v \notin \mathbf{Z} \end{cases}$

(act?) $\texttt{actor?}(v) \xrightarrow{f}_{\mathrm{k}\zeta} \begin{cases} \texttt{t} & \text{if } v \in \mathbf{A} \\ \texttt{f} & \text{if } v \notin \mathbf{A} \end{cases}$

(cons?) $\texttt{cons?}(v) \xrightarrow{f}_{\mathrm{k}\zeta} \begin{cases} \texttt{t} & \text{if } v \in \texttt{cons}(\mathbf{V}, \mathbf{V}) \\ \texttt{f} & \text{if } v \notin \texttt{cons}(\mathbf{V}, \mathbf{V}) \end{cases}$

(car) $\texttt{car}(\texttt{cons}(v_0, v_1)) \xrightarrow{f}_{\mathrm{k}\zeta} v_0$

(cdr) $\texttt{cdr}(\texttt{cons}(v_0, v_1)) \xrightarrow{f}_{\mathrm{k}\zeta} v_1$

(equal?) $\texttt{equal?}(v_0, v_1) \xrightarrow{f}_{\mathrm{k}\zeta} \begin{cases} \texttt{t} & \text{if } \{v_0, v_1\} \subseteq \mathbf{At} \text{ and } v_0 = v_1 \\ \texttt{f} & \text{if } \{v_0, v_1\} \not\subseteq \mathbf{At} \text{ or } v_0 \neq v_1 \end{cases}$

(beta-v) $\texttt{app}(\lambda x.e, v) \xrightarrow{f}_{\mathrm{k}\zeta} e[x := v]$

(lam) $\texttt{procedure?}(v) \xrightarrow{f}_{\mathrm{k}\zeta} \begin{cases} \texttt{t} & \text{if } v \in \lambda\mathbf{X}.\mathbf{E} \\ \texttt{nil} & \text{if } v \notin \lambda\mathbf{X}.\mathbf{E} \end{cases}$

The sequential reduction relation is then defined by lifting functional reduction and adding the rule for `clc`.

Definition (Sequential steps ($\xrightarrow{s}_{\mathrm{k}\zeta}$)):

(rdx) $R[\![e]\!] \xrightarrow{s}_{\mathrm{k}\zeta} R[\![e']\!]$ if $e \xrightarrow{f}_{\mathrm{k}\zeta} e'$

(clc) $R[\![\texttt{clc}(v)]\!] \xrightarrow{s}_{\mathrm{k}\zeta} \texttt{app}(v, \lambda x.R[\![x]\!]) \qquad x \notin \mathrm{FV}(R[\![\texttt{nil}]\!])$

`clc` captures the actors local continuation, R, as a function, $\lambda x.R[\![x]\!]$, and applies its argument v to this function, in the empty reduction context (the local top level). We let $\xrightarrow{s}\!\!\!\!\!\rightarrow_{\mathrm{k}\zeta}$ be the reflexive, transitive closure of $\xrightarrow{s}_{\mathrm{k}\zeta}$.

Now we are ready to define the reaction rules of ${}^{\mathrm{k}}AT$.

Definition (${}^{\mathrm{k}}RR$):

$\texttt{seq}(a) : [\![e]\!]_a \Rightarrow [\![e']\!]_a$ if $e \xrightarrow{s}_{\mathrm{k}\zeta} e'$ where $self({}^{\mathrm{k}}\zeta) = a$

$\texttt{send}(a, v_0, v_1) : [\![R[\![\texttt{send}(v_0, v_1)]\!]]\!]_a \Rightarrow [\![R[\![\texttt{nil}]\!]]\!]_a \ , \ {}^{\mathrm{k}}Emit(v_0 \triangleleft v_1)$

$\texttt{ready}(a, {}^{\mathrm{k}}M) : [\![R[\![\texttt{ready}(v)]\!]]\!]_a \ , \ a \triangleleft {}^{\mathrm{k}}M \Rightarrow [\![\texttt{app}(v, {}^{\mathrm{k}}M)]\!]_a$

$\texttt{leta}(a, \vec{a}) : [\![R[\![\texttt{letactor}\{\bar{x} := \bar{e}\}\ e]\!]]\!]_a \Rightarrow [\![R[\![e[\bar{x} := \vec{a}]]\!]]\!]_a \ , \ \{ [\![e_i[\bar{x} := \vec{a}]]\!]_{a_i} \}_{1 \leq i \leq m}$

if $\mathrm{Len}(\vec{a}) = \mathrm{Len}(\bar{x}) = m$ and $\vec{a} \cap acq(R[\![e, \bar{e}]\!]) = \emptyset$

Where

${}^{\mathrm{k}}Emit(v_0 \triangleleft v_1) = \begin{cases} v_0 \triangleleft v_1 & \text{if } v_0 \in \mathbf{A} \text{ and } v_1 \in {}^{\mathrm{k}}\mathbf{M} \\ \emptyset & \text{otherwise.} \end{cases}$

The meta-function ${}^{\mathrm{k}}Emit$ prevents ill-formed messages from getting into the system. An alternative is for an actor to block if it attempts to send an ill-formed message. The former option is chosen for compatibility with our earlier

work, but given the presence of recognizers, the choice is not important, the two semantics are inter-definable. The labels ${}^{\mathrm{k}}\mathbf{L}$ of the kernel actor theory are thus:

$${}^{\mathrm{k}}\mathbf{L} = \mathtt{seq}(\mathbf{A}) \cup \mathtt{send}(\mathbf{A},\mathbf{V},\mathbf{V}) \cup \mathtt{ready}(\mathbf{A},{}^{\mathrm{k}}\mathbf{M}) \cup \mathtt{leta}(\mathbf{A},\mathbf{A}^*)$$

where in the case of $\mathtt{leta}(a,\vec{a})$ we require $a \notin \vec{a}$. The acquaintances of a label are the actor names which textually appear in the expressions involved, for example $acq(\mathtt{ready}(a,{}^{\mathrm{k}}M)) = \{a\} \cup acq({}^{\mathrm{k}}M)$. Renaming is simply substitution.

The following lemma provides a case analysis for reduction in ${}^{\mathrm{k}}AT$. It states that, from a big step point of view, an actor is either permanently silent (**diverge.1-3**), or it is going to send a message, or create some actors, or is ready to receive a message. And precisely one of these cases holds.

Lemma (k.sred): For kernel expressions, $e \in \mathbf{E}$, and abstract contexts, ${}^{\mathrm{k}}\zeta$, exactly one of the following holds:

(diverge.1) $(\exists v \in \mathbf{V})(e \overset{s}{\longrightarrow\!\!\!\!\succ}_{{}^{\mathrm{k}}\zeta} v)$

(diverge.2) $e \overset{s}{\longrightarrow\!\!\!\!\succ}_{{}^{\mathrm{k}}\zeta} e' \quad e' \notin \mathbf{V}$ and the redex of e' does not reduce

(diverge.3) e has an infinite reduction sequence consisting solely of $\overset{s}{\longrightarrow}_{{}^{\mathrm{k}}\zeta}$ steps

(send) $(\exists v_0, v_1 \in \mathbf{V},\ R \in \mathbf{R})(e \overset{s}{\longrightarrow\!\!\!\!\succ}_{{}^{\mathrm{k}}\zeta} R[\![\mathtt{send}(v_0,v_1)]\!])$

(ready) $(\exists v \in \mathbf{V},\ R \in \mathbf{R})(e \overset{s}{\longrightarrow\!\!\!\!\succ}_{{}^{\mathrm{k}}\zeta} R[\![\mathtt{ready}(v)]\!])$

(letactor) $(\exists n \in \mathbf{N},\ e_i \in \mathbf{E}\ (1 \le i \le n),\ x_i \in \mathbf{X},\ (1 \le i \le n \text{ pairwise distinct}),\ R \in \mathbf{R})$

$$(e \overset{s}{\longrightarrow\!\!\!\!\succ}_{{}^{\mathrm{k}}\zeta} R[\![\mathtt{letactor}\{x_i := e_i\}_{1\le i\le n} e_0]\!])$$

Now we can state formally the sense in which the example kernel programs using different buffer behaviors are equivalent.

Lemma (kernel BB): The equivalence of the two versions of the bounded buffer programs is expressed by:

$$Isem([\![{}^{\mathrm{k}}P_1]\!]) = Isem([\![{}^{\mathrm{k}}P_2]\!]).$$

The two programs have the same big-step computations since they differ only in the sequential algorithm used by the buffers to maintain consistency.

4 The User Language

User language programs have the same shape as kernel language programs. However, there are several important differences:

- In the user language there is no lambda abstraction and thus no functions as values and actor behaviors are no longer simply functions to be applied to the contents of a message. Instead, the functions and behaviors used in a program are defined in a first-order manner in the program's library of (mutually recursive) definitions.
- Message contents are restricted to have the form $mid[\bar{v}]@c$ where mid is a method identifier, $\bar{v}$ is the list of method arguments and c is the customer, an actor name or `nil` if no customer is specified. In addition to the `self` primitive there is a zero-ary `customer` primitive that evaluates to the customer of the message currently being processed.
- Behaviors are a simple form of object oriented class description. They specify a set of local state variables and a set of methods.

- A *behavior definition* has the form

  `behavior` $bid(p)(methodDefs)$.

  where bid is the behavior identifier, p is a parameter list (a possibly variable length list of distinct variables described below), and $methodDefs$ is a set of method definitions.

- A *method definition* specifies when a method can be invoked and the result of invocation. It has the form

  `method` $mid(p)[$ `disable-when` $e_c,\ ]\ e$

  where mid is a method name (a symbol from $\mathbf{Sym}$), p is a user parameter list, e_c is the optional disabling condition (assumed false when not present), and e is the method body. We also incorporate the ability to handle arbitrary methods by including a *default method*.

  `otherwise`(x) `disable-when` $e_c,\ e$

  The default method is invoked when no other method in the behavior definition matches the incoming message.

- Disabling constraints (sometimes called synchronization constraints or guards) allow the expression of conditions under which method invocation might violate internal consistency or invariants. They also allow such conditions to be specified separately from the action to be performed when it is invoked. e_c is required to be functional, i.e. its evaluation involves no actor primitives other than `self` or `customer`.

- For consistency we require that a method (i.e a method identifier) should have a unique definition within a given behavior. The free variables of constraints and method bodies must be among the method parameters or the behavior parameters.

- A *function definition* has the form

  `function` $fid(p)\,e$

  where fid is a function identifier, p is a user parameter list, and e is an expression, the function body. The free variables of the function body must be among the function parameters.

- We also generalize parameter lists (of behavior, methods and functions) to incorporate variable length parameter lists. Thus user parameter lists come in two varieties:

  - Fixed length parameter lists:

    $[x_1, \ldots, x_n]$

    which are used to define abstractions of a fixed arity, in this case n.

  - Variable length parameter lists:

    $[x_1, \ldots, x_{n-1} \& x_n]$

    which are used to define abstractions of all arities greater than or equal to their fixed length part, i.e. $[x_1, \ldots, x_{n-1}]$, in this case $n-1$.

- In addition to behavior, method, and function definitions, a user library also allows for the definition of constants. A constant definition has the form

  `constant` $cid\ v$

  where cid is a constant identifier, and v is a user value expression. As mentioned above, user value expressions do not include lambda abstractions.

- In addition to asynchronous method invocation we also include remote procedure calls.

4.1 User Syntax

The user language has the same variables, basic data, actor names, and data operations as the kernel language. In addition we assume given two disjoint, countably infinite sets of identifiers: $\mathbf{FunId}$ for functions; and $\mathbf{BehId}$ for behaviors. Expressions are built from atoms and variables by the following operations: application of primitive operations to sequences of expressions, let binding, conditional branching, the `letactor` actor creation construct, and asynchronous and synchronous method invocation. The primitive operations include operations on basic data and pairs, and following user primitives: `self`, the name of the executing actor (arity 0); `customer`, the customer of the current message (arity 0); *fid*, user defined operations for $fid \in \mathbf{FunId}$; and `ready`, specifying the behavior for the next message. An *asynchronous invocation* is of the form $e_a \triangleleft mid[\bar{e}]@e_c$. The target of the request is the value of e_a and the message contents has method name mid, arguments $\bar{e}$ and customer, e_c. Once the target, arguments, and customer are evaluated, `nil` is returned as the value and the requesting actor proceeds with its computation without waiting for a reply. A *synchronous invocation* (also referred to as a *request* or *remote procedure call*) is of the form $e_a \,.\, mid[\bar{e}])$. The target of the request is the value of e_a and the message contents has method name mid, arguments $\bar{e}$. The requesting actor suspends execution until a reply is received. The value of the synchronous invocation is contents of the reply (actually it is the first member of the contents of the reply). Requests are not actually synchronous exchanges between the actor requesting the service and target, but simply a standard protocol (pattern of message passing) for making a request and receiving a reply. A *ready* expression is of the form $\texttt{ready}(bid(e_1, \ldots, e_n))$. Execution of a ready expression terminates processing of the current message and looks for the next message enabled for the behavior bid with parameters given by the values of the e_i. If there is no enabled message in the local message queue the actor waits for one to be delivered.

Definition (User Programs and Libraries):

$$
\begin{aligned}
{}^{\mathrm{u}}\mathbf{Program} = \texttt{program}(&\texttt{receptionists} : \mathcal{P}_\omega[\mathbf{A}], \quad \texttt{externals} : \mathcal{P}_\omega[\mathbf{A}] \\
&\texttt{library} : \mathcal{P}_\omega[(\mathbf{BehDef} \cup \mathbf{FunDef})] \\
&\texttt{actors} : \mathcal{P}_\omega[\mathbf{A} := \mathbf{E}] \\
&\texttt{messages} : \mathcal{M}_\omega[\mathbf{A} \triangleleft {}^{\mathrm{u}}\mathbf{M}])
\end{aligned}
$$

$$
{}^{\mathrm{u}}\mathbf{M} = \mathbf{MethId}[\mathbf{V}^*]@(\mathbf{A} \cup \{\texttt{nil}\})
$$

$$
\mathbf{V} = \mathbf{At} \cup \texttt{cons}(\mathbf{V}, \mathbf{V})
$$

where the actor names in the `actors` part must be distinct, and all actor names occurring in an actor state or message contents must either be one of the actor names defined in the `actors` part or one of the names occurring in the `externals` part. We let ${}^{\mathrm{u}}M$ range over ${}^{\mathrm{u}}\mathbf{M}$. Message contents consist of a method identifier (symbol), an argument list (a list of values), and a customer (an actor name or `nil` signifying no customer). We identify $\mathbf{MethId}$ with $\mathbf{Sym}$ and let mid range over $\mathbf{MethId}$ and use mid to stand for a symbol used as a method identifier. We let c range over $\mathbf{A} \cup \{\texttt{nil}\}$ and we may omit the customer part of a message if it is `nil`. $mid[\bar{v}]@c$ abbreviates the list construction $\texttt{msgMk}(mid, \bar{v}, c)$ defined in §3.1. We use the following meta functions: $msgMeth(M)$ selects the method component; $msgArgs(M)$ selects the arguments component; and $msgCust(M)$ selects the customer component. Thus

$$
msgMeth(msgMk(mid, \bar{v}, c)) = mid,
$$

$$
msgArgs(msgMk(mid, \bar{v}, c)) = \bar{v},
$$

$$
msgCust(msgMk(mid, \bar{v}, c)) = c.
$$

The definition of the library component is given by

$$
\mathbf{BehDef} = \texttt{behavior}\ \mathbf{BehId}(\mathbf{X}^\circ)(\mathbf{MethDef}^*[\mathbf{DefaultMethDef}]).
$$

$$
\mathbf{MethDef} = \texttt{method}\ \mathbf{MethId}(\mathbf{X}^\circ)[\ \texttt{disable\_when}\ \mathbf{E},\ ]\ \mathbf{E}
$$

$$
\mathbf{DefaultMethDef} = \texttt{otherwise}(\mathbf{X})[\ \texttt{disable\_when}\ \mathbf{E},\ ]\ \mathbf{E}
$$

$$
\mathbf{FunDef} = \texttt{function}\ \mathbf{FunId}(\mathbf{X}^\circ)\mathbf{E}
$$

where user parameter lists, $\mathbf{X}^\circ$, are defined by:

$$\mathbf{X}^\circ = \mathbf{X}^* \cup (\mathbf{X}^* \& \mathbf{X})$$

A library is well-formed if it contains at most one definition of each $fid \in \mathbf{FunId}$, and $bid \in \mathbf{BehId}$, and these definitions themselves are well-formed. We let $\bar{x}$, $\bar{y}$, and p range over variable length lists of distinct variables.

Definition (User Expressions):

$$\mathbf{O} = \mathbf{dOp} \cup \{\texttt{self}, \texttt{customer}\} \cup \mathbf{FunId}$$

$$\mathbf{At} = \mathbf{A} \cup \mathbf{Bool} \cup \mathbf{Z} \cup \mathbf{Sym}$$

$$\begin{aligned}\mathbf{E} = {} & \mathbf{X} \cup \mathbf{At} \cup \mathbf{O}(\mathbf{E}_n) \cup \texttt{ready}(\mathbf{BehId}(\mathbf{E}_n)) \quad \cup \\ & \texttt{if}(\mathbf{E},\mathbf{E},\mathbf{E}) \cup \texttt{let}\{(\mathbf{X} := \mathbf{E})^+\}\mathbf{E} \cup \texttt{letactor}\{(\mathbf{X} := \mathbf{E})^+\}\mathbf{E} \quad \cup \\ & \mathbf{E} \triangleleft \mathbf{MethId}[\mathbf{E}^*]@\mathbf{E} \cup \mathbf{E}\,.\,\mathbf{MethId}[\mathbf{E}^*]\end{aligned}$$

We let e range over $\mathbf{E}$. The binding constructs are `letactor` and `let`. $\texttt{let}\{\ldots x_i := e_i \ldots\}e$ binds the x_i in e, but not in any of the e_j. In the term $\mathbf{O}(\mathbf{E}_n)$ we allow the application of $fid \in \mathbf{FunId}$ to arbitrary sequences of expressions, but restrict the fixed arity expressions: $\mathbf{dOp} \cup \{\texttt{self}, \texttt{customer}\}$ to sequences of the appropriate length.

4.2 Example User Programs

To illustrate the user language we show how the buffer program of §3 might be expressed in the user language. We first define the library, ${}^{\mathrm{u}}Lib$, of behaviors.

$$\begin{aligned}&\texttt{behavior BoundedBuffer}(n,q)(\texttt{method put}(y)\ \texttt{disable\_when}\ n \le \texttt{length}(q),\\ &\qquad\qquad\qquad\qquad\qquad\qquad \texttt{ready}(\texttt{BoundedBuffer}(n,\texttt{cons}(y,q))),\\ &\qquad\qquad\qquad\qquad \texttt{method get}()\ \texttt{disable\_when length}(q) = 0,\\ &\qquad\qquad\qquad\qquad\qquad\qquad \texttt{seq}(\texttt{customer}() \triangleleft \texttt{element}[\texttt{car}(q)]@\texttt{self}(),\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{ready}(\texttt{BoundedBuffer}(n,\texttt{cdr}(q)))))\\ &\texttt{behavior aFilter}(b_{\mathrm{no}},b_{\mathrm{yes}})(\texttt{method item}(x)\ \texttt{seq}(\texttt{if}(\Phi(x), b_{\mathrm{yes}} \triangleleft \texttt{put}[x], b_{\mathrm{no}} \triangleleft \texttt{put}[x]),\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{ready}(\texttt{aFilter}(b_{\mathrm{no}},b_{\mathrm{yes}})))),\\ &\qquad\qquad\qquad\qquad \texttt{method get}(x)\ \texttt{seq}(\ \texttt{if}(\Phi(x),\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad b_{\mathrm{yes}} \triangleleft \texttt{get}[]@\texttt{customer}(),\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad b_{\mathrm{no}} \triangleleft \texttt{get}[]@\texttt{customer}()),\\ &\qquad\qquad\qquad\qquad\qquad\qquad \texttt{ready}(\texttt{aFilter}(b_{\mathrm{no}},b_{\mathrm{yes}})))\end{aligned}$$

A configuration corresponding to the kernel configuration described in §3 is described by the following program, abbreviated by ${}^{\mathrm{u}}P$.

$$\begin{aligned}{}^{\mathrm{u}}P \quad \triangleq \quad & \texttt{program}(\texttt{receptionists}:\ a_{\mathrm{f}},\quad \texttt{externals}:\\ & \qquad \texttt{library}: {}^{\mathrm{u}}Lib\\ & \qquad \texttt{actors}:\ a_{\mathrm{y}} := \texttt{BoundedBuffer}(100,\texttt{nil}),\\ & \qquad\qquad\quad a_{\mathrm{n}} := \texttt{BoundedBuffer}(100,\texttt{nil}),\\ & \qquad\qquad\quad a_{\mathrm{f}} := \texttt{aFilter}(a_{\mathrm{n}}, a_{\mathrm{y}})\\ & \qquad \texttt{messages}:\ )\end{aligned}$$

The library, ${}^{\mathrm{u}}Lib$, contains the above behavior definitions of `aFilter` and `BoundedBuffer`.

In this example we are using asynchronous message passing to place items in the buffer. Consequently the filter actor will have absolutely no control over the actual order in which the items appear in the buffer. Perhaps in a more

realistic version the filter actor would use the remote procedure primitive:

$$\begin{aligned}\texttt{behavior}\ \texttt{sFilter}(b_{\rm no}, b_{\rm yes})\ \texttt{method}\ \texttt{item}(x)\ \ &\texttt{seq}(\texttt{if}(\Phi(x), b_{\rm yes}\texttt{.put}[x], b_{\rm no}\texttt{.put}[x],\\ &\qquad \texttt{ready}(\texttt{sFilter}(b_{\rm no}, b_{\rm yes}))))\\ \texttt{method}\ \texttt{get}(x)\ \ &\texttt{seq}(\texttt{if}(\Phi(x),\\ &\qquad b_{\rm yes} \triangleleft \texttt{get}[]@\texttt{customer}(),\\ &\qquad b_{\rm no} \triangleleft \texttt{get}[]@\texttt{customer}()),\\ &\qquad \texttt{ready}(\texttt{aFilter}(b_{\rm no}, b_{\rm yes})))\end{aligned}$$

4.3 The User Language Semantics

As for the kernel language, the semantics is given by defining an actor theory ${}^{\rm u}AT$. Since libraries do not evolve we parameterize the actor theory by the library of definitions in force, letting the library be just part of the auxiliary axioms describing the actor theory. Thus to give the semantics we need only define user states, ${}^{\rm u}\mathbf{S}$, the user labels ${}^{\rm u}\mathbf{L}$, and give ${}^{\rm u}RR$, relative to the given library. Messages may arrive before they are enabled for processing. The simple way to deal with this is to simply resend a message that is not enabled when it arrives. This will not provide the intended level of fairness because a message could be infinitely often enabled but never be processed because it happens to always be delivered when disabled. Thus user states include additional information to manage an internal mail queue that collects messages that have arrived when disabled and this queue is walked each time the actor is ready to process a new message, before additional new messages are considered.

There are five kinds of actor states:

- $[e, c, Q_{\rm u}]$ — processing a message with customer c, with current state e. The unprocessed or unchecked messages are queued in $Q_{\rm u}$.
- $[bid(\bar{v}), Q_{\rm u}, Q_{\rm r}]$ — traversing the queue of delivered but unprocessed messages, $Q = Q_{\rm u} * Q_{\rm r}$, looking for a message that is not disabled. The current behavior is $bid(\bar{v})$. The message (contents) in $Q_{\rm r}$ have been checked and rejected (i.e. they are disabled). The messages in $Q_{\rm u}$ are yet to be checked.
- $[\varphi, e, {}^{\rm u}M, bid(\bar{v}), Q_{\rm u}, Q_{\rm r}]$ — checking disabling constraints of behavior bid for message ${}^{\rm u}M$, the method body that may or may not be disabled is e. φ represents the state of the constraint evaluation.
- $[a', R, c, Q]$ — waiting for a reply to a request. R is a reduction context – the continuation of the computation upon receipt of an answer. a' is an actor created to serve as a reply address, to distinguish the reply from other arriving messages.
- $[a]$ – the state of an actor serving as a reply address for a request sent by a;

where a, a' are actor names, e is an expression (of the user language), c is a customer — an actor name or `nil`, φ is functional expression, ${}^{\rm u}M$ is the contents of a user message, and Q is a mail queue – a sequence of messages (or more precisely their contents).

A state of the form $[e, c, Q]$ attempts to step by decomposing e into a reduction context and redex and reducing the redex. It is hung if the redex fails to reduce.

A state of the form $[bid(\bar{v}), Q_{\rm u}, Q_{\rm r}]$ is hung if $\bar{v}$ does not match the parameter list of the definition of bid. Otherwise a state of the form $[bid(\bar{v}), [\,], Q_{\rm r}]$ is waiting for delivery of a message (having already walked through its queue and found no enabled messages); and a state of the form $[bid(\bar{v}), [{}^{\rm u}M] * Q_{\rm u}, Q_{\rm r}]$ steps by starting evaluation of the constraints associated, in the behavior definition for bid, with the method name of ${}^{\rm u}M$. A state of the form $[\varphi, e, {}^{\rm u}M, bid(\bar{v}), Q_{\rm r}, Q_{\rm u}]$ steps by evaluating φ one step if it is not a value expression. If φ is the value `f`, then it starts evaluation of the method body, e, associated with the method of ${}^{\rm u}M$ in the behavior definition for bid. If φ is a value other than `f` then ${}^{\rm u}M$ is considered disabled and put on the end of rejects queue, $Q_{\rm r}$.

States of the last two forms occur in pairs $[\,a', R, c, Q\,]_a$, $[\,a\,]_{a'}$ that are waiting for the delivery of a reply to a request by a that will arrive as a message to a', serving as a unique request identifier.

The set of *user actor states*, ${}^{\rm u}\mathbf{S}$ is defined as follows.

Definition (${}^{\mathrm{u}}\mathbf{S}$):

$$\begin{aligned}{}^{\mathrm{u}}\mathbf{S} = {} & (\mathbf{E} \times (\mathbf{A} \cup \{\mathtt{nil}\}) \times {}^{\mathrm{u}}\mathbf{M}^*) \quad \cup \\ & (\mathbf{BehId}(\mathbf{V}^*) \times {}^{\mathrm{u}}\mathbf{M}^* \times {}^{\mathrm{u}}\mathbf{M}^*) \quad \cup \\ & (\mathbf{E} \times \mathbf{E} \times {}^{\mathrm{u}}\mathbf{M} \times \mathbf{BehId}(\mathbf{V}^*) \times {}^{\mathrm{u}}\mathbf{M}^* \times {}^{\mathrm{u}}\mathbf{M}^*) \quad \cup \\ & (\mathbf{A} \times \mathbf{R} \times (\mathbf{A} \cup \{\mathtt{nil}\}) \times {}^{\mathrm{u}}\mathbf{M}^*) \quad \cup \\ & \mathbf{A}\end{aligned}$$

The acquaintances of states and their parts are just the actor names occurring in the structure and renaming is substitution as usual.

The meaning of a user program is defined to be a configuration of ${}^{\mathrm{u}}AT$ as follows.

Definition ($[\![{}^{\mathrm{u}}P]\!]$ $Isem({}^{\mathrm{u}}P)$): Let ${}^{\mathrm{u}}P$ be given by

$$\begin{aligned}{}^{\mathrm{u}}P \quad \triangleq \quad & \mathtt{program}(\mathtt{receptionists}: \rho, \quad \mathtt{externals}: \chi \\ & \qquad \mathtt{library}: Lib \\ & \qquad \mathtt{actors}: \{a_j := e_j\}_{1 \le j \le m} \\ & \qquad \mathtt{messages}: \{a'_j \triangleleft {}^{\mathrm{u}}M_j\}_{1 \le j \le n})\end{aligned}$$

then

$$[\![{}^{\mathrm{u}}P]\!] = \left\langle\!\left\langle\, {}^{\mathrm{u}}I \,\right\rangle\!\right\rangle_{\chi}^{\rho}$$

where

$${}^{\mathrm{u}}I = \{[\, e_j, \mathtt{nil}, \mathtt{nil}\,]_{a_j}\}_{1 \le j \le m} \ , \ \{a'_j \triangleleft {}^{\mathrm{u}}M_j\}_{1 \le j \le n}$$

and

$$Isem({}^{\mathrm{u}}P) = Isem([\![{}^{\mathrm{u}}P]\!] : {}^{\mathrm{u}}AT)$$

To complete the definition of ${}^{\mathrm{u}}AT$ we must give the reaction rules. We first define some auxiliary meta functions and predicates to ease definition of rules concerning behaviors and methods: $parCheck$, $behMatch$, and $methMatch$.

$parCheck$ holds of a user parameter list p and a sequence of user values $\bar{v}$ iff p is of the fixed length variety and p and $\bar{v}$ have the same length, or p is of the variable length variety, and the length of $\bar{v}$ is not less than the length of the fixed length part of p.

Assuming that $parCheck(p, \bar{v})$ holds, then we write $e[p := \bar{v}]$ for the simultaneous substitution of the i-th value in $\bar{v}$ for the ith variable in p when p and $\bar{v}$ have the same length. When p is of the variable length variety we also substitute the final variable (i.e. the one appearing after the $\&$) for the remaining list of values not matched by the fixed length part of p. In symbols:

$$parCheck([x_1, \ldots x_{n-1}, \&x_n], [v_1, \ldots, v_m]) \quad \text{iff} \quad m \ge n-1$$

$$e[[x_1, \ldots x_{n-1}, \&x_n] := [v_1, \ldots, v_m]] = e[[x_1 := v_1, \ldots, x_{n-1} := v_{n-1}, x_n := \mathtt{list}_{m-n+1}(v_n, \ldots, v_m)]].$$

$behMatch$ tests whether the parameters of ready expressions match those of the behavior definition.

$methMatch$ given a behavior identifier, a parameter list, and a message extracts two expressions: the constraint associated with the method (of the message); and the method body associated with the method. We extract these using $cstrExp$ and $methExp$. $cstrExp$ extracts the constraint associated with the method; and $methExp$ extracts the method body .

Definition ($behMatch$, $methMatch$, $methExp$, $cstrExp$):

1. $behMatch(Lib, bid, \bar{v})$ holds iff $\mathtt{behavior}\ bid(p)(methodDefs) \in Lib$ for some $p, methodDefs$ such that $parCheck(p, \bar{v})$.

2. $methMatch(Lib, bid, \bar{v}, mid[\bar{v}']@c)$ is defined under the assumption that $behMatch(Lib, bid, \bar{v})$ holds. In particular we assume that `behavior` $bid(p)(methodDefs) \in Lib$ for some p, $methodDefs$ and that $parCheck(p, \bar{v})$ holds. We then define $methMatch$ in three mutually exclusive and exhaustive cases:

   (a) If `method` $mid(p')$[`disable-when` e_d,] e in $methodDefs$ and $parCheck(p', \bar{v}')$, then

   $$methMatch(Lib, bid, \bar{v}, mid[\bar{v}']@c) = [e_d[p' := \bar{v}'][p := \bar{v}], e[p' := \bar{v}'][p := \bar{v}]]$$

   If there is no disabling constraint, then the first component is taken to be `f`.

   (b) If the pre-conditions of the previous case fail, and `otherwise`(x)[`disable-when` e_d,] e in $methodDefs$, then

   $$methMatch(Lib, bid, \bar{v}, mid[\bar{v}']@c) = [e_d[[\&x] := \bar{v}'][p := \bar{v}], e[[\&x] := \bar{v}'][p := \bar{v}]]$$

   If there is no disabling constraint, then the first component is taken to be `f`.

   (c) If neither of the above cases hold then

   $$methMatch(Lib, bid, \bar{v}, mid[\bar{v}']@c) = [\mathtt{t}, \mathtt{f}]$$

Note that in the instantiation of the disabling constraint and the method body expressions the binding of method parameters shadows that of behavior parameters.

As in the kernel language, to give the reaction rules, we first define the sequential reduction relation, $e \xrightarrow{s}_{u\zeta} e'$, parameterized by an abstract context, $^u\zeta$. In addition to the *self* function, we use a function $customer(^u\zeta)$ to extract the customer of the current message. Hence in the user language the abstract context can be identified with a sequence of two actor names. We begin by defining the values $\mathbf{V}$, reduction contexts $\mathbf{R}$ and redexes $\mathbf{E}_{\mathrm{rdx}}$ of the user language, again giving the unique decomposition property for non-value expressions.

Definition ($\mathbf{V}$ $\mathbf{E}_{\mathrm{rdx}}$ $\mathbf{R}$):

$$\mathbf{V} = \mathbf{At} \cup \mathtt{cons}(\mathbf{V}, \mathbf{V})$$

$$\mathbf{E}_{\mathrm{rdx}} = (\mathbf{O}_n(\mathbf{V}^n) - \mathtt{cons}(\mathbf{V}, \mathbf{V})) \cup \mathtt{ready}(\mathbf{BehId}(\mathbf{V}^n)) \cup \mathtt{if}(\mathbf{V}, \mathbf{E}, \mathbf{E}) \cup \mathtt{let}\{(\mathbf{X} := \mathbf{V})^+\}\mathbf{E} \cup \mathtt{letactor}\{(\mathbf{X} := \mathbf{E})^+\}\mathbf{E} \cup \mathbf{V} \triangleleft \mathbf{MethId}[\mathbf{V}^*]@\mathbf{V} \cup \mathbf{V}\,.\,\mathbf{MethId}[\mathbf{V}^*]$$

$$\mathbf{R} = \{\bullet\} \cup \mathbf{O}_{n+m+1}(\mathbf{V}^n, \mathbf{R}, \mathbf{E}^m) \cup \mathtt{ready}(\mathbf{BehId}(\mathbf{V}^n, \mathbf{R}, \mathbf{E}^m)) \cup \mathtt{if}(\mathbf{R}, \mathbf{E}, \mathbf{E}) \cup \mathtt{let}\{(\mathbf{X} := \mathbf{V},)^*\mathbf{X} := \mathbf{R}(, \mathbf{X} := \mathbf{E})^*\}\mathbf{E} \cup \mathbf{R} \triangleleft \mathbf{MethId}[\mathbf{E}^*]@\mathbf{E} \cup \mathbf{V} \triangleleft \mathbf{MethId}[\mathbf{V}^*, \mathbf{R}, \mathbf{E}^*]@\mathbf{E} \cup \mathbf{V} \triangleleft \mathbf{MethId}[\mathbf{V}^*]@\mathbf{R} \cup \mathbf{R}\,.\,\mathbf{MethId}[\mathbf{E}^*] \cup \mathbf{V}\,.\,\mathbf{MethId}[\mathbf{V}^*, \mathbf{R}, \mathbf{E}^*]$$

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

Definition ($\xrightarrow{f}_{u\zeta}$): $e \xrightarrow{f}_{u\zeta} e'$ differs from the relation defined in §3 by deleting the functional clauses ((**lam**) and (**beta-v**)) and the addition of the following three clauses:

(cust) $\mathtt{customer}() \xrightarrow{f}_{u\zeta} customer(^u\zeta)$

(let) $\mathtt{let}\{\bar{x} := \bar{v}\}e \xrightarrow{f}_{u\zeta} e[\bar{x} := \bar{v}]$

(fun) $fid(\bar{v}) \xrightarrow{f}_{u\zeta} e[p := \bar{v}]$ if `function` $fid(p)e \in Lib$ and $parCheck(p, \bar{v})$ holds

In the user language all sequential redexes are purely functional. Thus $\xrightarrow{s}_{u\zeta}$ is defined by

(rdx) $R[\![e]\!] \xrightarrow{s}_{u\zeta} R[\![e']\!]$ if $e \xrightarrow{f}_{u\zeta} e'$

We let $\xrightarrow{s}\!\!\!\!\!\rightarrow_{u\zeta}$ be the reflexive, transitive closure of $\xrightarrow{s}_{u\zeta}$. Notice that the sequential rules are sufficient to evaluate functional expressions, in particular we only need the sequential rules to check constraints.

The labelled reaction rules for the user language are given by the following.

Definition (${}^{\mathrm{u}}RR$):

$\mathtt{seq}(a) : [\, e, c, Q_{\mathrm{u}} \,]_a \Rightarrow [\, e', c, Q_{\mathrm{u}} \,]_a \qquad \text{if } e \xrightarrow{s}_{a,c} e'$

$\mathtt{send}(a) : [\, R[\![v \triangleleft mid[\bar{v}]@v']\!], c, Q_{\mathrm{u}} \,]_a \Rightarrow [\, R[\![\mathtt{nil}]\!], c, Q_{\mathrm{u}} \,]_a \ , \ {}^{\mathrm{u}}Emit(v \triangleleft mid[\bar{v}]@v')$

$\mathtt{rpc}(a, a_0) : [\, R[\![v \,.\, mid[\bar{v}]]\!], c, Q_{\mathrm{u}} \,]_a \Rightarrow [\, a_0, R, c, Q_{\mathrm{u}} \,]_a \ , \ [\, a \,]_{a_0} \ , \ {}^{\mathrm{u}}Emit(v \triangleleft mid[\bar{v}]@a_0)$

if $a_0 \notin acq([\, R[\![v \,.\, mid[\bar{v}]]\!], c, Q_{\mathrm{u}} \,]_a)$

$\mathtt{rcv}(a, a_0, v) : [\, a_0, R, c, Q_{\mathrm{u}} \,]_a \ , \ [\, a \,]_{a_0} \ , \ a_0 \triangleleft mid[[v] * \bar{v}]@c' \Rightarrow [\, R[\![v]\!], c, Q_{\mathrm{u}} \,]_a \ , \ [\, \mathtt{nil}, \mathtt{nil}, [\,] \,]_{a_0}$

$\mathtt{deliver}(a, {}^{\mathrm{u}}M) : [\, bid(\bar{v}), [\,], Q \,]_a \ , \ a \triangleleft {}^{\mathrm{u}}M \Rightarrow [\, bid(\bar{v}), [{}^{\mathrm{u}}M], Q \,]_a$

$\mathtt{walk}(a) : [\, R[\![\mathtt{ready}(bid(\bar{v}))]\!], c, Q_{\mathrm{u}} \,]_a \Rightarrow [\, bid(\bar{v}), Q_{\mathrm{u}}, [\,] \,]_a \qquad \text{if } behMatch(Lib, bid, \bar{v})$

$\mathtt{leta}(a, \vec{a}) : [\, R[\![\mathtt{letactor}\{\bar{x} := \bar{e}\} e]\!], c, Q_{\mathrm{u}} \,]_a \Rightarrow [\, R[\![e[\bar{x} := \vec{a}]]\!], c, Q_{\mathrm{u}} \,]_a \ , \ \{ [\, e_i[\bar{x} := \vec{a}], c, [\,] \,]_{a_i} \}_{1 \le i \le k}$

if $\mathrm{Len}(\vec{a}) = \mathrm{Len}(\bar{x}) = k$ and $\vec{a} \cap acq(R[\![\mathtt{letactor}\{\bar{x} := \bar{e}\} e]\!], c, Q_{\mathrm{u}}) = \emptyset$

$\mathtt{cstr}(a) : [\, bid(\bar{v}), [{}^{\mathrm{u}}M] * Q_{\mathrm{u}}, Q_{\mathrm{r}} \,]_a \Rightarrow [\, e_d, e_m, {}^{\mathrm{u}}M, bid(\bar{v}), Q_{\mathrm{u}}, Q_{\mathrm{r}} \,]_a$

if $methMatch(Lib, bid, \bar{v}, {}^{\mathrm{u}}M) = [e_d, e_m]$

$\mathtt{enable}(a) : [\, \mathtt{f}, e_m, {}^{\mathrm{u}}M, bid(\bar{v}), Q_{\mathrm{u}}, Q_{\mathrm{r}} \,]_a \Rightarrow [\, e_m, msgCust({}^{\mathrm{u}}M), Q_{\mathrm{u}} * Q_{\mathrm{r}} \,]_a$

$\mathtt{disable}(a) : [\, v, e_m, {}^{\mathrm{u}}M, bid(\bar{v}), Q_{\mathrm{u}}, Q_{\mathrm{r}} \,]_a \Rightarrow [\, bid(\bar{v}), Q_{\mathrm{u}}, Q_{\mathrm{r}} * [{}^{\mathrm{u}}M] \,]_a \qquad \text{if } v \neq \mathtt{f}$

$\mathtt{check}(a) : [\, e_0, e_m, {}^{\mathrm{u}}M, bid(\bar{v}), Q_{\mathrm{u}}, Q_{\mathrm{r}} \,]_a \Rightarrow [\, e_1, e_m, {}^{\mathrm{u}}M, bid(\bar{v}), Q_{\mathrm{u}}, Q_{\mathrm{r}} \,]_a \qquad \text{if } e_0 \xrightarrow{s}_{a, msgCust({}^{\mathrm{u}}M)} e_1$

where $\quad {}^{\mathrm{u}}Emit(v \triangleleft {}^{\mathrm{u}}M) = \begin{cases} v \triangleleft {}^{\mathrm{u}}M & \text{if } v \in \mathbf{A} \text{ and } msgCust({}^{\mathrm{u}}M) \in \mathbf{A} \cup \{\mathtt{nil}\} \\ \emptyset & \text{otherwise.} \end{cases}$

As in the kernel language, the meta-function ${}^{\mathrm{u}}Emit$ prevents ill-formed messages from getting into the system. The labels of ${}^{\mathrm{u}}AT$ are

$${}^{\mathrm{u}}\mathbf{L} = \mathtt{seq}(\mathbf{A}) \cup \mathtt{send}(\mathbf{A}) \cup \mathtt{rpc}(\mathbf{A}, \mathbf{A}) \cup \mathtt{rcv}(\mathbf{A}, \mathbf{A}, \mathbf{V}) \cup \mathtt{deliver}(\mathbf{A}, {}^{\mathrm{u}}\mathbf{M}) \cup \mathtt{walk}(\mathbf{A}) \quad \cup$$
$$\mathtt{leta}(\mathbf{A}, \mathbf{A}^*) \cup \mathtt{cstr}(\mathbf{A}) \cup \mathtt{enable}(\mathbf{A}) \cup \mathtt{disable}(\mathbf{A}) \cup \mathtt{check}(\mathbf{A})$$

where in $\mathtt{leta}(a, \vec{a})$ and $\mathtt{rpc}(a, a')$ a' and the elements of $\vec{a}$ must be distinct and different from a. The set of acquaintances of a label is the union of the actor parameters mentioned in the label, except $acq(\mathtt{rcv}(a, a_0, v)) = \{a, a_0\} \cup acq(v)$ and $acq(\mathtt{deliver}(a, {}^{\mathrm{u}}M)) = \{a\} \cup acq({}^{\mathrm{u}}M)$.

Lemma (well-formedness invariant): The following conditions are invariants of user configuration computation steps (assuming that `behavior` $bid(p)(methodDefs) \in Lib$):

uc.1 if $[\, a', R, c, Q_{\mathrm{u}} \,]_a \in {}^{\mathrm{u}}I$, then $[\, a \,]_{a'} \in {}^{\mathrm{u}}I$;

uc.2 if $[\, bid(\bar{v}), Q_{\mathrm{u}}, Q_{\mathrm{r}} \,]_a \in {}^{\mathrm{u}}I$, then $parCheck(p, \bar{v})$ holds.

uc.3 if $[\, e_0, e_m, M, bid(\bar{v}), Q_{\mathrm{u}}, Q_{\mathrm{r}} \,]_a \in {}^{\mathrm{u}}I$, then $parCheck(p, \bar{v})$ holds, and if $M = mid[\bar{v}_m]@c$, then there is some `method` $mid(p_m)e \in methodDefs$ and $parCheck(p_m, \bar{v}_m)$ holds.

The following lemma is the user version of (**k.sred**), it provides a case analysis for reduction in ${}^{\mathrm{u}}AT$. From a big step point of view, an actor is either permanently silent (**diverge.1-3**), is going to send a message, do a remote procedure call, create some actors, is ready to receive a message. And exactly one of these cases holds.

Lemma (u.sred): For user expressions, $e \in \mathbf{E}$, and abstract context, ${}^{\mathrm{u}}\zeta$, exactly one of the following holds:

(diverge.1) $\quad (\exists v \in \mathbf{V})(e \xrightarrow{s}\!\!\!\!\succ_{{}^{\mathrm{u}}\zeta} v)$

(diverge.2) $\quad e \xrightarrow{s}\!\!\!\!\succ_{{}^{\mathrm{u}}\zeta} e' \quad e' \notin \mathbf{V}$ and the redex of e' does not reduce

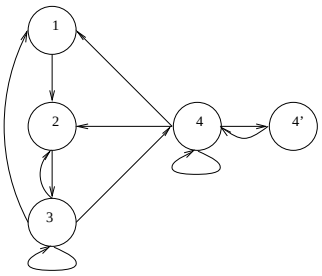

Figure 1: User Language FSM

(diverge.3) e has an infinite reduction sequence consisting solely of $\xrightarrow{s}_{\mathrm{u}\zeta}$ steps

(send) $(\exists v \in \mathbf{V}, \bar{v} \in \mathbf{V}*, c \in \mathbf{A} \cup \{\mathtt{nil}\} R \in \mathbf{R}, mid \in \mathbf{MethId})(e \xrightarrow{s}\!\!\!\!\succ_{\mathrm{u}\zeta} R[\![v \triangleleft mid[\bar{v}]@c]\!])$

(rpc) $(\exists v \in \mathbf{V}, \bar{v} \in \mathbf{V}*, R \in \mathbf{R}, mid \in \mathbf{MethId})(e \xrightarrow{s}\!\!\!\!\succ_{\mathrm{u}\zeta} R[\![v\,.\,mid[\bar{v}]]\!])$

(ready) $(\exists \bar{v} \in \mathbf{V}^*, R \in \mathbf{R}, bid \in \mathbf{MethId})(e \xrightarrow{s}\!\!\!\!\succ_{\mathrm{u}\zeta} R[\![\mathtt{ready}(bid(\bar{v}))]\!]))$

(letactor) $(\exists n \in \mathbf{N}, e_i \in \mathbf{E}\ (1 \le i \le n), x_i \in \mathbf{X}\ (1 \le i \le n \text{ pairwise distinct}), R \in \mathbf{R})$

$(e \xrightarrow{s}\!\!\!\!\succ_{\mathrm{u}\zeta} R[\![\mathtt{letactor}\{x_i := e_i\}_{1 \le i \le n} e_0]\!])$

Finally we may relate the behavior of the user version of the bounded buffer with the two kernel versions given in §3.

Lemma (user kernel BB): The equivalence of the kernel and user versions of the bounded buffer is expressed by:

$$Isem([\![^{\mathrm{k}}P_1]\!]) \lceil^{\mathrm{u}}\mathbf{M} = Isem([\![^{\mathrm{k}}P_2]\!]) \lceil^{\mathrm{u}}\mathbf{M} = Isem([\![^{\mathrm{u}}P]\!]).$$

We can represent the user language reduction system as a parameterized finite state machine. The parameterized finite state machine is represented diagrammatically in figure 1. The states and transitions are defined below.

Definition (User Semantics FSM States):

State Number	Actor
(**State 1**)	$[\,bid(\bar{v}), [\,], Q_{\mathrm{r}}\,]_a$
(**State 2**)	$[\,bid(\bar{v}), [^{\mathrm{u}}M] * Q_{\mathrm{u}}, Q_{\mathrm{r}}\,]_a$
(**State 3**)	$[\,e_d, e_m, bid(\bar{v}), Q_{\mathrm{u}}, Q_{\mathrm{r}}\,]_a$
(**State 4**)	$[\,e, c, Q_{\mathrm{u}}\,]_a$
(**State 4′**)	$[\,a_1, R, c, Q_{\mathrm{u}}\,]_a$, $[\,a\,]_{a_1}$

Definition (User Semantics FSM Transitions):

State 1 → State 2: requires the delivery of a $a \triangleleft {}^{\mathrm{u}}M$.

State 2 → State 3: requires no side conditions, other than that e_d and e_m are the appropriate disabling constraint and method body, respectively.

State 3 → State 3: requires only that the constraint checking does not terminate yet.

State 3 → State 4: requires that the constraint checking terminates and that the message is enabled.

State 3 → State 2: requires that the constraint checking terminates, the message is disabled, and that the remaining unchecked messages is not empty, $Q_{\mathrm{u}} \neq []$.

State 3 → State 1: requires that the constraint checking terminates, the message is disabled, and that the remaining unchecked messages is empty, $Q_{\mathrm{u}} = []$.

State 4 → State 4: requires that computation continues, but does not execute either a `ready` or `request` expression.

State 4 → State 4': requires the execution of a `request` expression. This creates an auxiliary actor to handle the `request` reply.

State 4' → State 4: requires delivery of a message to the auxiliary reply actor.

State 4 → State 1: requires that a `ready` expression is executed and that the remaining unchecked messages is empty, $Q_{\mathrm{u}} = []$.

State 4 → State 2: requires that a `ready` expression is executed and that the remaining unchecked messages is not empty, $Q_{\mathrm{u}} \neq []$.

Lemma (user fsm): The actors is a user configuration have states corresponding to one of the user FSM states and steps involving these actors are FSM transitions.

5 A Semantics Preserving User to Kernel Translation

In this section we define a translation, $u2k : {}^{\mathrm{u}}\mathcal{L} \to {}^{\mathrm{k}}\mathcal{L}$ and show that this translation preserves interaction semantics. We do this in three stages. We first describe and define the translation formally. We then give a formal statement of the correctness of $u2k$ together with an outline of the proof the correctness in sufficient detail for the reader to understand its structure. Finally we fill in some of the details in the proof outline.

5.1 The Translation $u2k$ on Syntactic Entities

$u2k$ is a family of maps, one for each syntactic category. The members of the family are distinguished by context of application rather than by name. Programs are translated by translating the library, actors, and messages parts. A library is translated by translating the constant, function and behavior definitions, producing a kernel language library. A user actor description is translated into a closely coupled pair of actors consisting of a mail–queue actor and a behavior actor. The mail–queue actor accepts messages, manages the internal mail queue, and interacts with the behavior actor according to a fixed uniform protocol. The behavior actor embodies the actual behavior of the user actor and is known only to the mail–queue actor and any auxiliary remote procedure call actor created for doing remote procedure calls. The behavior of the behavior actor is obtained by translating the expression part assuming it executes in a local context in which the current message has no customer and the mail–queue actor is initialized with an empty mail queue. The behavior of the mail–queue actor is independent of the behavior of the user actor, it simply requires the address of the behavior actor it is fronting for. A message is translated by treating the user message syntax as syntactic sugar in the kernel language, as explained in §3.1.

The core of the translation is its action on expressions. Expressions are translated in the context of a user library. In order to leave this dependence implicit, we adopt a standard convention about converting user constant, function and behavior identifiers into variables and assume sufficient renaming has been done to avoid conflicts. These translate to kernel function identifiers. We write ${}^{\mathrm{k}}cid$, ${}^{\mathrm{k}}fid$, and ${}^{\mathrm{k}}bid$ for the translation of cid, fid, and bid respectively. The translation $u2k(e)$ of a user expression is a lambda term of the form $\lambda c.\lambda q.e$ which when applied to a *customer*, c, and a *mail–queue* actor, q, reduces to a kernel expression that corresponds to the user expression executing in a local context where the current message has customer c and the internal mail queue corresponds to that held by the mail–queue actor q. We use the following abbreviation

$$u2k^*(e, c, q) \triangleq \mathtt{app}(u2k(e), c, q)$$

in defining the translation.

The translation of the expression forms that are common to the two languages as well as `customer` and asynchronous send are straightforward. It amounts to passing the customer and mail–queue parameters to the translated subexpressions. The translation of synchronous invocations (requests) and `ready` are where care is needed. In the user language, the transition that delivers the reply to a request involves two actors, the actor requesting the reply and the actor created to serve as the reply address, as well as the reply message. Kernel actor transitions involve at most one actor. The three-body interaction is replaced by a delivery to the reply address followed by a forwarding and delivery to the requesting behavior actor. This works because the address of the behavior actor is known only to its mail–queue actor and the currently active reply actor. Furthermore, while waiting for a request reply, there are no undelivered messages for the behavior actor, and there will be none, until the reply actor sends one.

The translation of a `ready` expression must initiate the queue walking protocol between the behavior actor and its mail–queue to walk the message queue, checking the disabling constraints for the method of each message. If an enabled message is found, then the translated method body is executed.

We begin by defining the mapping on programs, and work our way down to expressions.

$u2k$ on Programs

Programs are translated as follows:

$$
\begin{array}{l}
u2k(\texttt{program}(\texttt{receptionists}:\rho \quad \texttt{externals}:\chi \\
\qquad\qquad \texttt{library}:{}^{\mathrm{u}}Lib \\
\qquad\qquad \texttt{actors}:\{a_i := e_i\}_{1\le i\le k} \\
\qquad\qquad \texttt{messages}:\{a'_j \triangleleft M_j\}_{1\le i\le n})) \\
\triangleq \\
\texttt{program}(\texttt{receptionists}:\rho \quad \texttt{externals}:\chi \\
\qquad\quad \texttt{library}:u2k({}^{\mathrm{u}}Lib) \\
\qquad\quad \texttt{actors}: \quad \{a_i := \texttt{Qwaiting}(a_i^*,\texttt{nil},\texttt{nil})\}_{1\le i\le k} \\
\qquad\qquad\qquad\quad \{a_i^* := u2k^*(e_i,\texttt{nil},a_i)\}_{1\le i\le k} \\
\qquad\quad \texttt{messages}:\{a'_j \triangleleft M_j\}_{1\le i\le n})
\end{array}
$$

where a_i^* are fresh, pairwise distinct actor names $1 \le i \le k$

Note that $u2k$ is the identity on $\mathbf{M}$, since as described in §3.1 we use the following definitions for structuring message contents in the kernel language

$$\texttt{list}_n := \lambda x_1.\lambda x_2.\ldots.\lambda x_n\texttt{cons}(x_1,\texttt{cons}(x_2,\ldots\texttt{cons}(x_n,\texttt{nil})))$$

$$\texttt{msgMk} := \lambda x_{\mathrm{mid}}.\lambda x_{\mathrm{args}}.\lambda x_{\mathrm{cust}}.\texttt{list}_3(x_{\mathrm{mid}},x_{\mathrm{args}},x_{\mathrm{cust}})$$

$$\texttt{msgMeth} := \lambda x.\texttt{car}(x)$$

$$\texttt{msgArgs} := \lambda x.\texttt{car}(\texttt{cdr}(x))$$

$$\texttt{msgCust} := \lambda x.\texttt{car}(\texttt{cdr}(\texttt{cdr}(x)))$$

In the user language, as described in §4, we assume that $mid[\bar{v}]@c$ abbreviates the list construction $\texttt{msgMk}(mid,\bar{v},c)$.

$u2k$ on Library Definitions

The translation on library definition is pointwise:

$$u2k([D_1\ldots D_n]) \triangleq [u2k(D_1),\ldots,u2k(D_n)]$$

where each D_i is a constant, function or behavior definition.

Since user values are also kernel values, constants translate to thunks which when applied to anything return the value:

$$u2k(\texttt{constant}\ cid\ v) \quad \triangleq \quad {}^{\mathrm{k}}cid := \lambda c.\lambda q.\lambda x.v$$

They are translated to thunks simply because kernel libraries are restricted to function definitions. The translation of a user definition of fid yields a kernel definition of the associated function identifier ${}^{\mathrm{k}}fid$. The translation of behavior definitions is a little more complex. The translation of functions and behaviors must account for several differences between the user and kernel language. User functions, behaviors and methods have user parameter lists. The translated operations will be applied to a list of arguments and must check if the number of arguments is correct and then bind these to the individual parameters. For this purpose, we define a family of abbreviations: $\texttt{parCheck}[p]$ determines whether a list of values is of the appropriate length, while $\texttt{parBind}[p, v, e]$ binds the elements of p to appropriate values determined by v in the expression e. `length`, `listtail`, and `listref`, are kernel procedures that compute the correspondingly named functions on lists á la Scheme. `length` computes the length of a list with the empty list having length 0. $\texttt{listtail}(v, i)$ returns the sublist of v obtained by omitting the first i elements, thus if v is a list then $\texttt{listtail}(v, 0) = v$. $\texttt{listref}(v, i)$ returns the ith entry of the list v, with the first element being the 0th entry.

$$\texttt{parCheck}[p] \triangleq \begin{cases} \lambda y.\texttt{equal?}(\texttt{length}(y), n) & \text{if } p = [x_1, \ldots, x_n] \\ \lambda y.\texttt{length}(y) \geq n & \text{if } p = [x_1, \ldots, x_n \& x_{n+1}] \end{cases}$$

$$\texttt{parBind}[p, v, e] \triangleq \begin{cases} \texttt{let}\{x_i := \texttt{listref}(v, i)\}_{1 \leq i \leq n} e & \text{if } p = [x_1, \ldots, x_n] \\ \texttt{let}\{x_i := \texttt{listref}(v, i), x_n := \texttt{listtail}(v, n)\}_{1 \leq i \leq n-1} e & \text{if } p = [x_1, \ldots, x_{n-1} \& x_n] \end{cases}$$

Note that the definition of $\texttt{parBind}[p, v, e]$ assumes that v and p have the same length.

The translation of a function definition is given by:

$$\begin{aligned} u2k(\texttt{function}\ fid(p)\, e) \quad \triangleq \quad {}^{\mathrm{k}}fid := \lambda c.\lambda q.\lambda y.\texttt{if}(&\texttt{parCheck}[p](y), \\ &\texttt{parBind}[p, y, u2k^*(e, c, q)], \\ &\texttt{hang}) \end{aligned}$$

where `hang` is some functional redex that fails to reduce, for example $\texttt{app}(\texttt{nil}, \texttt{nil})$, thus hanging the computation if the arguments do not match the parameters.

The translation of a behavior definition is:

$$\begin{aligned} &u2k(\texttt{behavior}\ bid(p)(methodDefs)) \quad \triangleq \\ &\quad {}^{\mathrm{k}}bid := \lambda q.\lambda y.\texttt{if}(\texttt{parCheck}[p](y), \\ &\qquad \texttt{seq}(\texttt{send}(q, \texttt{WALK}[]@\texttt{self}()), \\ &\qquad\quad \texttt{ready}(\texttt{Wait4Q}(u2k(methodDefs, p), q, y))), \\ &\qquad \texttt{nil}) \end{aligned}$$

Note that once the instantiated parameters have been determined to be syntactically correct, the behavior actor notifies its mail-queue actor that it has become ready for a new message. This aspect of the translation is part of the behavior actor mail-queue actor protocol which we will treat in detail shortly.

The addition of methods involves the translation of method definitions. $u2k(methodDefs)$ is a data-structure that encodes the appropriate responses to the appropriate methods. In particular it is an alist (a Lisp style association list) that pairs method identifiers with the translated methods. The translation of each method, including the default method, consists of a triple (a list of length three). The first element of the triple is a predicate that determines whether or not a list of arguments matches the user parameter list of the method. The second element of the triple is a function that corresponds to the disabling constraint of the method, while the third element is the functional embodiment of the method body. We call these triples *method entries* and use the following abstract syntax to make their manipulation more readable:

$$\texttt{methEntryMk} := \lambda x_{\mathrm{arity}}.\lambda x_{\mathrm{cstr}}.\lambda x_{\mathrm{body}}.\texttt{list}_3(x_{\mathrm{arity}}, x_{\mathrm{cstr}}, x_{\mathrm{body}})$$

$\texttt{methEntryArity} := \lambda x.\texttt{car}(x)$

$\texttt{methEntryCstr} := \lambda x.\texttt{car}(\texttt{cdr}(x))$

$\texttt{methEntryBody} := \lambda x.\texttt{car}(\texttt{cdr}(\texttt{cdr}(x)))$

Similarly to make the manipulation of the alist legible, we use:

$\texttt{methId} := \lambda x.\texttt{car}(x)$

$\texttt{methEntry} := \lambda x.\texttt{cdr}(x)$

Let *methodDefs* be

$$\{\texttt{method}\ mid_i(p_i)\ \texttt{disable-when}\ e_i^d,\ e_i^m, \texttt{otherwise}(x_{bid})\ \texttt{disable-when}\ e^d,\ e^m \mid 1 \leq i \leq m_{bid}\}$$

(e_i^d and e^d are taken to be `f` if no disabling constraint is present). Then the alist that associates each method identifier with its method entry is:

$$\begin{array}{l} u2k(methodDefs, p) \triangleq \\ \quad \texttt{list}_{m_{bid}+1}(\texttt{cons}(mid_1, u2k(p, \texttt{method}\ mid_1(p_1)\ \texttt{disable-when}\ e_1^d,\ e_1^m)) \\ \qquad\qquad \vdots \\ \qquad\qquad \texttt{cons}(mid_i, u2k(p, \texttt{method}\ mid_i(p_i)\ \texttt{disable-when}\ e_i^d,\ e_i^m)) \\ \qquad\qquad \vdots \\ \qquad\qquad \texttt{cons}(mid_{m_{bid}}, u2k(p, \texttt{method}\ mid_{m_{bid}}(p_{m_{bid}})\ \texttt{disable-when}\ e_{m_{bid}}^d,\ e_{m_{bid}}^m)) \\ \qquad\qquad \texttt{cons}(\texttt{nil}, u2k(p, \texttt{otherwise}(x_{bid})\ \texttt{disable-when}\ e^d,\ e^m))) \end{array}$$

The method entries are defined to be the translation of the corresponding method:

$$\begin{array}{l} u2k(p, \texttt{method}\ mid_i(p_i)\ \texttt{disable-when}\ e_i^d,\ e_i^m) \triangleq \\ \quad \texttt{methEntryMk}(\lambda y.\texttt{parCheck}[p_i](y), \\ \qquad\qquad \lambda y.\lambda m.\texttt{let}\{c := \texttt{msgCust}(m), v := \texttt{msgArgs}(m)\} \\ \qquad\qquad\quad \texttt{parBind}[p, y, \texttt{parBind}[p_i, v, u2k^*(e_i^d, c, \texttt{nil})]], \\ \qquad\qquad \lambda y.\lambda m.\texttt{let}\{c := \texttt{msgCust}(m), v := \texttt{msgArgs}(m)\} \\ \qquad\qquad\quad \texttt{parBind}[p, y, \texttt{parBind}[p_i, v, u2k^*(e_i^m, c, q)]]) \end{array}$$

and

$$\begin{array}{l} u2k(p, \texttt{otherwise}(x_{bid})\ \texttt{disable-when}\ e^d,\ e^m)) \triangleq \\ \quad \texttt{methEntryMk}(\lambda y.\texttt{t}, \\ \qquad\qquad \lambda y.\lambda m.\texttt{let}\{c := \texttt{msgCust}(m), v := \texttt{msgArgs}(m)\} \\ \qquad\qquad\quad \texttt{parBind}[p, y, \texttt{parBind}[[\&x_{bid}], v, u2k^*(e_c^d, \texttt{nil})]], \\ \qquad\qquad \lambda y.\lambda m.\texttt{let}\{c := \texttt{msgCust}(m), v := \texttt{msgArgs}(m)\} \\ \qquad\qquad\quad \texttt{parBind}[p, y, \texttt{parBind}[[\&x_{bid}], v, u2k^*(e^m, c, q)]]) \end{array}$$

The main function for manipulating these data structures is the lookup procedure `MethodMatch`. This procedure attempts to find the triple associated with a method identifier and a list of values. For an entry to match, the identifier must be the same, and the list of values must be of the appropriate length. To determine if the latter is the case the first function in the triple is used. If a match is found then the associated triple is returned. Otherwise the default triple is used.

$\texttt{MethodMatch} :=$

$\lambda MD, mid, v.$`if(null?(`MD`),`
`nil,`
`let{`$I :=$ `methId(car(`MD`)), `$A :=$ `methEntryArity(methEntry(car(`MD`)))}`
`if(or(null?(`I`),`
`and(equal?(`I, mid`), `$A(v)$`)),`
`methEntry(car(`MD`)),`
`MethodMatch(cdr(`MD`), `mid, v`)))`

The following lemma states that `MethodMatch` works in the same way as the meta function $methMatch$.

Lemma (MethodMatch): If `behavior` $bid(p)(methodDefs) \in Lib$, $parCheck(p, \bar{v})$, ${}^{\mathtt{u}}M = mid[\bar{v}']@c$, and $MD = u2k(methodDefs, p)$, then $methMatch(Lib, bid, \bar{v}, {}^{\mathtt{u}}M) = e_d, e_m$ iff `MethodMatch`$(MD, mid, \bar{v}')$ silently reduces to `methEntryMk`(A, C, B) where `app`$(C, \bar{v}', {}^{\mathtt{u}}M)$ is the same as $u2k^*(e_d, c,$ `nil`$)$ and `app`$(B, \bar{v}', {}^{\mathtt{u}}M, q)$ is the same as $u2k^*(e_m, c, q)$ (up to silent reductions).

In translating the behavior of a user actor, we replace each user actor by a pair of actors: the mail–queue actor and the real behavior, so to speak. The mail–queue actor is the actor that may be known by other actors in the system or externally. The only actors that know the behavior actor are the mail–queue actor and any auxiliary remote procedure call actor created for doing remote procedure calls. A behavior actor only directly receives messages from: its mail–queue actor in answer to a request; an auxiliary remote procedure call actor serving as one-time receptionist for an remote procedure call reply.

The behavior actor

We begin by describing the simpler of the two, the behavior actor. The behavior actor is either:

- in state `Wait4Q` waiting for a message from its mail–queue actor to check whether it is disabled or not. If it receives a message, it checks to see whether it is enabled or not. If it is disabled, it merely requests another message from its mail–queue actor, and returns to the `Wait4Q` state. If the message is enabled, it reports this fact to the mail–queue actor and then waits for an acknowledgement before processing the body, it waits for acknowledgement in the `OK?` state. When an acknowledgement is received it then executes the (translation of) the body of the method associated with the enabled message. This of course is the third component of the method entry.
- in state `OK?` having just reported that a message is enabled and is waiting for an acknowledgement from the mail–queue behavior.

`Wait4Q`(MD, q, y) $:=$
$\lambda m.$`let{`$z :=$ `MethodMatch(`MD`, msgMeth(`m`), msgArgs(`m`))}`
`let{`$C :=$ `methEntryCstr(`z`), `$B :=$ `methEntryBody(`z`)}`
`if(`$C(y, m)$`,`
`seq(send(`q`, NEXT[]@self()),`
`ready(Wait4Q(`MD, q, y`))),`
`seq(send(`q`, ENABLED[]@self()),`
`ready(OK?(`$\lambda().B(y, m, q)$`)))))`

`OK?` $:=$ $\lambda B.\lambda m.$`if(equal?(msgMeth(`m`), OK), `$B()$`, nil)`

The mail–queue actor

A mail–queue actor is in one of three states `Qidle`, `Qwaiting`, or `Qwalking`. These state are described as follows.

In a `Qidle` state it has a single queue consisting of the collection of currently disabled messages. Consequently no action can be taken until another possibly enabled message arrives. In this state the mail–queue actor will not receive an unsolicited message from its associated behavior actor. In this state it is simply waiting for an external message to forward to the behavior actor.

In a `Qwaiting` state it has forwarded a message to the behavior actor and received a reply to the effect that the message is enabled. It is thus waiting for the behavior actor to complete its processing of this message. Once done, the behavior actor will signal its completion by requesting new messages. The mail–queue actor in the `Qwaiting` state has two queues, the to-be-checked queue and the pending queue.

In a `Qwalking` state, the mail–queue actor has three queues – rejected, to-be-checked, and pending plus the current message. The current message has been forwarded to the behavior actor. The rejected queue consists of those messages already sent (in this walking period) to the behavior actor and rejected. The to-be-checked are messages that are still to be processed. The pending messages are those that have arrived since this walking period began. We use infix $*$ as a list appending operation, and $[m]$ to represent the singleton list containing m.

To represent these states the following definitions are added to the kernel library of each translated user program.

$$\texttt{Qidle}(b, q_{\text{reject}}) \quad := \quad \lambda m.\texttt{seq}(\texttt{send}(b, m), \texttt{ready}(\texttt{Qwalking}(b, \texttt{nil}, m, q_{\text{reject}}, \texttt{nil})))$$

$$\begin{array}{l}
\texttt{Qwaiting}(b, q_{\text{unchecked}}, q_{\text{pending}}) \quad := \\
\quad \lambda m.\texttt{if}(\texttt{not}(\texttt{equal?}(\texttt{msgCust}(m), b)), \\
\qquad \texttt{ready}(\texttt{Qwaiting}(b, q_{\text{unchecked}}, q_{\text{pending}} * [m])) \\
\qquad \texttt{if}(\texttt{equal?}(\texttt{msgMeth}(m), \texttt{WALK}) \\
\qquad\quad \texttt{if}(\texttt{null?}(q_{\text{unchecked}}), \\
\qquad\qquad \texttt{if}(\texttt{null?}(q_{\text{pending}}), \\
\qquad\qquad\quad \texttt{ready}(\texttt{Qidle}(b, \texttt{nil})), \\
\qquad\qquad\quad \texttt{let}\ \{m := \texttt{car}(q_{\text{pending}}), pq := \texttt{cdr}(q_{\text{pending}})\} \\
\qquad\qquad\qquad \texttt{seq}(\texttt{send}(b, m), \\
\qquad\qquad\qquad\quad \texttt{ready}(\texttt{Qwalking}(b, \texttt{nil}, m, \texttt{nil}, pq)))) \\
\qquad\qquad \texttt{let}\{m := \texttt{car}(q_{\text{unchecked}}), q := \texttt{cdr}(q_{\text{unchecked}})\} \\
\qquad\qquad\quad \texttt{seq}(\texttt{send}(b, m), \\
\qquad\qquad\qquad \texttt{ready}(\texttt{Qwalking}(b, q, m, \texttt{nil}, q_{\text{pending}})))) \\
\qquad\quad \texttt{nil}))
\end{array}$$

$$\texttt{Qwalking}(b, q_{\text{unchecked}}, m_{\text{c}}, q_{\text{reject}}, q_{\text{pending}}) \quad :=$$

$$
\begin{array}{l}
\lambda m.\texttt{if}(\texttt{not}(\texttt{equal?}(\texttt{msgCust}(m), b)),\\
\quad \texttt{ready}(\texttt{Qwalking}(b, q_{\text{unchecked}}, m_{\text{c}}, q_{\text{reject}}, q_{\text{pending}} * [m]))\\
\quad \texttt{if}(\texttt{equal?}(\texttt{msgMeth}(m), \texttt{NEXT})\\
\quad\quad \texttt{if}(\texttt{null?}(q_{\text{unchecked}}),\\
\quad\quad\quad \texttt{if}(\ \texttt{null?}(q_{\text{pending}}),\\
\quad\quad\quad\quad \texttt{ready}(\texttt{Qidle}(b, q_{\text{reject}} * [m_{\text{c}}])),\\
\quad\quad\quad\quad \texttt{let}\ \{m := \texttt{car}(q_{\text{pending}}), pq := \texttt{cdr}(q_{\text{pending}})\}\\
\quad\quad\quad\quad\quad \texttt{seq}(\texttt{send}(b, m),\\
\quad\quad\quad\quad\quad\quad \texttt{ready}(\texttt{Qwalking}(b, \texttt{nil}, m, q_{\text{reject}} * [m_{\text{c}}], pq))))\\
\quad\quad\quad \texttt{let}\ \{m := \texttt{car}(q_{\text{unchecked}}), q := \texttt{cdr}(q_{\text{unchecked}})\}\\
\quad\quad\quad\quad \texttt{seq}(\texttt{send}(b, m),\\
\quad\quad\quad\quad\quad \texttt{ready}(\texttt{Qwalking}(b, q, m, q_{\text{reject}} * [m_{\text{c}}], q_{\text{pending}}))))\\
\quad\quad \texttt{if}(\texttt{equal?}(\texttt{msgMeth}(m), \texttt{ENABLED}),\\
\quad\quad\quad \texttt{seq}(\texttt{send}(b, \texttt{OK[]@self}()),\\
\quad\quad\quad\quad \texttt{ready}(\texttt{Qwaiting}(b, q_{\text{unchecked}} * q_{\text{reject}}, q_{\text{pending}}))),\\
\quad\quad\quad \texttt{nil})))
\end{array}
$$

$u2k$ on Expressions

The translation of expressions is now quite straightforward, given the preceding picture.

$$u2k(cid) \triangleq \lambda c.\lambda q.{}^{\text{k}}cid(c, q, \texttt{nil}) \qquad \text{if } cid \in \mathbf{ConstId}$$

$$u2k(e) \triangleq \lambda c.\lambda q.e \qquad \text{if } e \in \mathbf{X} \cup \mathbf{At}$$

$$u2k(op(e_1, \ldots, e_n)) \triangleq \lambda c.\lambda q.op(u2k^*(e_1, c, q), \ldots, u2k^*(e_n, c, q)) \qquad \text{for } op \in \mathbf{dOp}$$

$$u2k(\texttt{customer}()) \triangleq \lambda c.\lambda q.c$$

$$u2k(\texttt{self}()) \triangleq \lambda c.\lambda q.q$$

$$u2k(fid(e_1, \ldots, e_n)) \triangleq \lambda c.\lambda q.\texttt{app}({}^{\text{k}}fid, c, q, \texttt{list}_n(u2k^*(e_1, c, q), \ldots u2k^*(e_n, c, q)))$$

$$u2k(\texttt{if}(e_0, e_1, e_2)) \triangleq \lambda c.\lambda q.\texttt{if}(u2k^*(e_0, c, q), u2k^*(e_1, c, q), u2k^*(e_2, c, q))$$

$$u2k(\texttt{let}\{x_i := e_i\}_{1 \le i \le n}\ e) \triangleq \lambda c.\lambda q.\texttt{let}\{x_i := u2k^*(e_i, c, q)\}_{1 \le i \le n} u2k^*(e, c, q)$$

$$u2k(\texttt{letactor}\{a_i := e_i\}_{1 \le i \le n}\ e) \triangleq \lambda c.\lambda q.\texttt{letactor}\{a_i := \texttt{ready}(\texttt{Qidle}(a_i^*, \texttt{nil})),\ a_i^* := u2k^*(e_i, c, a_i)\}_{1 \le i \le n} \quad u2k^*(e, c, q)$$

for a_i^* pairwise distinct and fresh $1 \le i \le n$

$$u2k(\texttt{ready}(bid(e_1, \ldots, e_n))) \triangleq \lambda c.\lambda q.\texttt{let}\{x_i := u2k^*(e_i, c, q)\}_{1 \le i \le n}\ \texttt{let}\{y := \texttt{list}_n(x_1, \ldots, x_n)\}\ \texttt{clc}(\lambda f.\texttt{app}({}^{\text{k}}bid, q, y))$$

`clc` is used to discard any enclosing reduction context

$$u2k(e_0 \triangleleft mid[e_1, \ldots, e_n]@e_{n+1}) \triangleq \lambda c.\lambda q.\texttt{let}\{x_i := u2k^*(e_i, c, q)\}_{0 \le i \le n+1}\ \texttt{send}(x_0, \texttt{msgMk}(mid, \texttt{list}_n(x_1, \ldots, x_n), x_{n+1}))$$

$$u2k(e_0 \,.\, mid[e_1, \ldots, e_n]) \triangleq$$
$$\lambda c.\lambda q.\texttt{let}\{x_i := u2k^*(e_i, c, q)\}_{0 \le i \le n}$$
$$\texttt{letactor}\{a_{\text{aux}} := \texttt{ready}(\texttt{RpcAux}(\texttt{self}()))\}$$
$$\texttt{seq}(\texttt{send}(x_0, \texttt{msgMk}(mid, \texttt{list}_n(x_1, \ldots, x_n), a_{\text{aux}})),$$
$$\texttt{clc}(\lambda k.\texttt{ready}(\texttt{RpcWait}(k))))$$

where the following definitions are also added to the generated kernel library of any program translation

$$\texttt{RpcAux} := \lambda x_a.\lambda m.\texttt{send}(x_a, \texttt{msgMk}(\texttt{nil}, \texttt{list}_1(\texttt{car}(\texttt{msgArgs}(m))), x_a)$$

$$\texttt{RpcWait} := \lambda k.\lambda m.\texttt{app}(k, \texttt{car}(\texttt{msgArgs}(m)))$$

Analysis of the $u2k$ Mail–Queue Actor & Behavior Actor Protocol

To help in understanding the translation, we establish two lemmas about configurations that arise in kernel computations starting from the translation of a user program (assuming input is restricted to user messages). The first states that the internal actors in a configuration are partitioned into groups associated with the actors coming from the user configuration. The second states the invariant obeyed by the local configurations of these groups.

Lemma (kernel groups): In any path in ${}^{\text{k}}AT$ whose initial configuration is the translation of a user program in which only user messages are input, the configurations satisfy the following:

1. the actors in the configuration can be partitioned into three (disjoint) sets: $Queue$ the mail–queue actors, Beh the behavior actors, and Aux the auxiliary actors.
2. there is a bijection $B : Queue \to Beh$ that pairs the mail–queue actors with their companion behavior actor.
3. there is a surjection $beh : Aux \to Beh$ that assigns to each auxiliary actor, the behavior actor it is associated with. There is at most one active auxiliary per behavior actor, i.e. with state that is not `nil` (modulo silent moves). For any b in Beh we call $\{b\} \cup B^{-1}(b) \cup beh^{-1}(b)$ the *group* associated with b, and denote it by $G(b)$.
4. only actors in $Queue \cup Aux$ can be receptionists or even acquaintances of other actors in the configuration other than those in the actor's group. Thus any messages to an actor in Beh must come from an actor in its group – its mail–queue actor or one of its auxiliaries. Messages to the mail–queue actor from within the group have the behavior actor as customer. Auxiliary actors are not sent messages by group members.

We can describe the configuration and transitions local to the group associated to a behavior actor by a *parametrized* finite state machine. This captures both the steps corresponding to actual computation on the user side and the protocol for interactions between the mail–queue actor and the behavior actor. In what follows the mail–queue actor is named a and its corresponding behavior actor is called $B(a)$. The method definition data-structure associated with the current behavior is MD. The parameterized finite state machine is represented diagrammatically in figure 2.

Definition (Group Protocol States):

Protocol State	Mail − Queue Actor State	Behavior Actor State	Internal Messages
(**State 1**)	$\texttt{Qidle}(B(a), Q)$	$\texttt{Wait4Q}(MD, a, y)$	
(**State 2**)	$\texttt{Qwalking}(B(a), Q_{\text{u}}, m_{\text{c}}, Q_{\text{r}}, Q_{\text{p}})$	$\texttt{Wait4Q}(MD, a, y)$	$B(a) \triangleleft m_{\text{c}}$
(**State 3**)	$\texttt{Qwalking}(B(a), Q_{\text{u}}, m_{\text{c}}, Q_{\text{r}}, Q_{\text{p}})$	constraint checking	
(**State 4**)	$\texttt{Qwalking}(B(a), Q_{\text{u}}, m_{\text{c}}, Q_{\text{r}}, Q_{\text{p}})$	$\texttt{Wait4Q}(MD, a, y)$	$a \triangleleft \texttt{NEXT}[]@B(a)$
(**State 5**)	$\texttt{Qwalking}(B(a), Q_{\text{u}}, m_{\text{c}}, Q_{\text{r}}, Q_{\text{p}})$	$\texttt{OK?}(b)$	$a \triangleleft \texttt{ENABLED}[]@B(a)$
(**State 6**)	$\texttt{Qwaiting}(B(a), Q_{\text{u}}, Q_{\text{p}})$	$\texttt{OK?}(b)$	$B(a) \triangleleft \texttt{OK}[]@a$
(**State 7**)	$\texttt{Qwaiting}(B(a), Q_{\text{u}}, Q_{\text{p}})$	computing reply	
(**State 7′**)	$\texttt{Qwaiting}(B(a), Q_{\text{u}}, Q_{\text{p}})$	$\texttt{RpcWait}(k)$, $\texttt{RpcAux}(B(a))$	
(**State 8**)	$\texttt{Qwaiting}(B(a), Q_{\text{u}}, Q_{\text{p}})$	$\texttt{Wait4Q}(MD, a, y)$	$a \triangleleft \texttt{WALK}[]@B(a)$

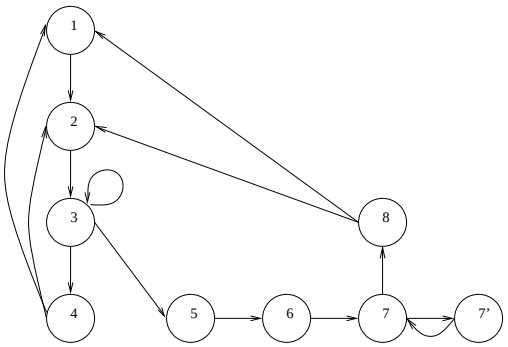

Figure 2: *u2k* Protocol FSM

Definition (Mail–queue & Behavior Actor Protocol Transitions):

State 1 → State 2: requires that an external message $a \triangleleft m_{\mathrm{c}}$ is delivered.

State 2 → State 3: requires no side conditions.

State 3 → State 3: assumes that the constraint checking continues.

State 3 → State 4: assumes that the constraint checking terminates, and that the message is disabled.

State 3 → State 5: assumes that the constraint checking terminates, and that the message is enabled.

State 4 → State 1: requires that there are no more unchecked or pending messages, i.e. $Q_{\mathrm{u}} = Q_{\mathrm{p}} = \mathtt{nil}$.

State 4 → State 2: requires that there is either are unchecked messages (and the top unchecked message is then checked), or else $Q_{\mathrm{u}} = \mathtt{nil}$ and there are pending messages, and the top one is then checked.

State 5 → State 6: requires no side conditions.

State 6 → State 7: requires no side conditions.

State 7 → State 7: requires that computation proceeds without executing either a `ready` or `request` expression.

State 7 → State 7': requires that the behavior actor executes a `request` expression. This creates an auxiliary mail–queue actor with state $\mathtt{RpcAux}(B(a))$.

State 7' → State 7: requires that the auxiliary mail–queue actor receives a reply from the external actor.

State 7 → State 8: requires that the behavior actor executes a `ready` expression.

State 8 → State 1: requires that there are no more unchecked or pending messages, i.e. $Q_{\mathrm{u}} = Q_{\mathrm{p}} = \mathtt{nil}$.

State 8 → State 2: requires that there is either are unchecked messages (and the top unchecked message is then checked), or else $Q_{\mathrm{u}} = \mathtt{nil}$ and there are pending messages, and the top one is then checked.

Lemma (kernel fsm): The actors and internal messages (with target or customer a behavior actor) associated to a give behavior actor have combined states corresponding to one of the *u2k* protocol FSM states and steps involving these actors are FSM transitions.

5.2 Sketch of the Correctness of $u2k$

In the previous sections we have given the syntax and semantics (as actor theories) of ${}^{\mathrm{u}}\mathcal{L}$ and ${}^{\mathrm{k}}\mathcal{L}$, and we have defined a translation, $u2k$ from the user language, ${}^{\mathrm{u}}\mathcal{L}$, to the kernel language, ${}^{\mathrm{k}}\mathcal{L}$. Now we want to prove that this translation preserves the semantics:

Theorem ($u2k$):

$$Isem({}^{\mathrm{u}}P) = Isem(u2k({}^{\mathrm{u}}P))\lceil {}^{\mathrm{u}}\mathbf{M}$$

where ${}^{\mathrm{u}}P$ is a user language program, ${}^{\mathrm{u}}AT$ is the user actor theory, ${}^{\mathrm{k}}AT$ is the kernel actor theory, $Isem$ maps programs to their interaction semantics in the appropriate actor theory, and $\lceil {}^{\mathrm{u}}\mathbf{M}$ restricts the kernel interactions to user language messages. More generally we extend $u2k$ to configurations and show that:

($u2k$**.1**) $\quad Isem({}^{\mathrm{u}}K : {}^{\mathrm{u}}AT) = Isem(u2k({}^{\mathrm{u}}K) : {}^{\mathrm{k}}AT)\lceil {}^{\mathrm{u}}\mathbf{M}.$

and

($u2k$**.2**) $\quad [\![u2k({}^{\mathrm{u}}P)]\!] = u2k([\![{}^{\mathrm{u}}P]\!])$

To extend $u2k$ to configurations, we extend $u2k$ to all the semantic entities that configurations are built up from. Then ($u2k$**.2**) is a direct calculation from these definitions. The ($u2k$**.1**) equivalence is established by a series of actor theory transformations that preserve the interaction semantics. This series of transformations yield a pair of *isomorphic* actor theories. In the remainder of this section we give an overview of the transformations and state the lemmas leading to the correctness result.

Definition ($AT_0 \gg AT_1$): We use the notation $AT_0 \gg AT_1$ to mean that actor theory AT_0 transforms to AT_1 in a semantics preserving manner: $Isem(K : AT_0) = Isem(K : AT_1)$ for relevant K.

User transform 1: On the user side we simply move to the big step semantics. ${}^{\mathrm{u}}AT^{\dagger}$ is the big step transform of ${}^{\mathrm{u}}AT$, consequently this equivalence follows from the lemma (**big step form**) of §2.5.

Lemma (user transform.1): $\quad {}^{\mathrm{u}}AT \gg {}^{\mathrm{u}}AT^{\dagger}.$

Kernel transform 1: The first transformation on the kernel side is to restrict attention to configurations in the image of $u2k$. We call the result ${}^{\mathrm{k}}AT_{\mathrm{r}}$. The equivalence for this step

$$Isem([\![u2k({}^{\mathrm{u}}P)]\!] : {}^{\mathrm{k}}AT)\lceil {}^{\mathrm{u}}\mathbf{M} = Isem([\![u2k({}^{\mathrm{u}}P)]\!] : {}^{\mathrm{k}}AT_{\mathrm{r}})$$

follows from the lemma (**cfig.restr**) of §2.5.

Definition (${}^{\mathrm{k}}AT_{\mathrm{r}}$): $\quad {}^{\mathrm{k}}AT_{\mathrm{r}} = ({}^{\mathrm{k}}AT\lceil {}^{\mathrm{u}}\mathbf{M})\lceil u2k(\mathbf{K} : {}^{\mathrm{u}}AT^{\dagger}).$

Lemma (kernel transform.1): $\quad {}^{\mathrm{k}}AT \gg {}^{\mathrm{k}}AT_{\mathrm{r}}.$

Kernel transform 2: In the second transformation on the kernel side we restrict attention to paths in which messages are delivered to the mail-queue actors only when they are in the `Qidle` state. At the same time we relax the fairness constraint so that a message may be delayed forever if the behavior actor becomes permanently silent. The resulting actor theory is called ${}^{\mathrm{k}}AT_{\mathrm{d}}$. The lemmas (**kernel groups**) and (**kernel fsm**) of §5 hold for ${}^{\mathrm{k}}AT_{\mathrm{d}}$ and furthermore mail-queue actor pending queues are always empty and can be ignored. Thus the computations correspond closely to the user level computations where messages are only delivered when the unchecked queue becomes empty.

Lemma (kernel transform.2): $\quad {}^{\mathrm{k}}AT_{\mathrm{r}} \gg {}^{\mathrm{k}}AT_{\mathrm{d}}.$

Kernel transform 3: To make the correspondence between user and kernel computations (for a given user configuration and its translation) easy to see, we make one final transformation. This transformation is a generalization of the transformation to big step form where communication steps local to a group (messages from the behavior actor to its mail-queue and from the mail-queue and auxiliaries to the behavior actor) are also considered silent.

Lemma (kernel transform.3): $\quad {}^{\mathrm{k}}AT_{\mathrm{d}} \gg {}^{\mathrm{k}}AT_{\mathrm{g}}$

Final step The final step of the proof is to show that the transformed user and kernel theories agree on user configurations and their translations.

Lemma (final step): $Isem({}^{\mathrm{u}}K : {}^{\mathrm{u}}AT^{\dagger}) = Isem(u2k({}^{\mathrm{u}}K) : {}^{\mathrm{k}}AT_{\mathrm{g}})$

This is proved by establishing the correspondence between admissible computations of the two theories.

5.3 Details of the Correctness of $u2k$

We first define the extension of $u2k$ and prove the lemma ($u2k$**.2**). Then we fill in missing details for the actor theory transformations. Then we establish the final correspondence by making a careful analysis of the user and kernel computations.

5.3.1 The Translation $u2k$ on Semantic Entities

To establish correctness of the user–kernel translation, we extend $u2k$ to actor theory configurations and show that this mapping preserves interaction semantics. The translation of a user actor, a, requires an additional actor, $B(a)$, the behavior actor. For this reason the translation on interiors takes an additional argument B which is a finite bijection between actor names. It is implicitly assumed that the domain and range of B are disjoint, and the names of the user actors occurring in the configuration lie in the domain of B, and the external actors of the configuration are disjoint from the range of B.

Definition ($u2k$ on Configurations, Interiors, Message Queues, & Messages):

Configurations: $u2k(\left\langle\!\left\langle\, {}^{\mathrm{u}}I \,\right\rangle\!\right\rangle^{\rho}_{\chi}) \triangleq \left\langle\!\left\langle\, u2k(B, {}^{\mathrm{u}}I) \,\right\rangle\!\right\rangle^{\rho}_{\chi}$

Interiors: $u2k(B, \emptyset) \triangleq \emptyset$ $\qquad u2k(B, {}^{\mathrm{u}}I_0\ ,\ {}^{\mathrm{u}}I_1) \triangleq u2k(B, {}^{\mathrm{u}}I_0)\ ,\ u2k(B, {}^{\mathrm{u}}I_1)$

Messages: $u2k(B, a \lhd {}^{\mathrm{u}}M) \triangleq a \lhd {}^{\mathrm{u}}M$

Message Queues: $u2k([\,]) \triangleq \mathtt{nil}$ $\qquad u2k([{}^{\mathrm{u}}M] * Q) \triangleq \mathtt{cons}({}^{\mathrm{u}}M, u2k(Q))$

Definition ($u2k$ on States): The translation on states also requires the extra parameter, B.

$u2k(B, [\![e, c, Q]\!]_a) \triangleq$

$\quad [\![\mathtt{ready}(\mathtt{Qwaiting}(B(a), u2k(Q), \mathtt{nil}))]\!]_a\ ,\ [\![u2k^*(e, c, a)]\!]_{B(a)}$

$u2k(B, [\![bid(\bar{v}), [\,], Q_r]\!]_a) \triangleq$

$\quad [\![\mathtt{ready}(\mathtt{Qidle}(B(a), u2k(Q_r)))]\!]_a\ ,\ [\![{}^{\mathrm{k}}bid(a, \mathtt{list}(\bar{v}))]\!]_{B(a)}$

$u2k(B, [\![bid(\bar{v}), [{}^{\mathrm{u}}M] * Q_l, Q_r]\!]_a) \triangleq$

$\quad [\![\mathtt{ready}(\mathtt{Qwalking}(B(a), u2k(Q_l), {}^{\mathrm{u}}M, u2k(Q_r)))]\!]_a\ ,\ B(a) \lhd {}^{\mathrm{u}}M\ ,\ [\![{}^{\mathrm{k}}bid(a, \mathtt{list}(\bar{v}))]\!]_{B(a)}$

$u2k(B, [\![\varphi, e, {}^{\mathrm{u}}M, bid(\bar{v}), Q_l, Q_r]\!]_a) \triangleq$

$\quad [\![\mathtt{ready}(\mathtt{Qwalking}(B(a), u2k(Q_l), {}^{\mathrm{u}}M, u2k(Q_r)))]\!]_a\ ,\ B(a) \lhd {}^{\mathrm{u}}M\ ,\ [\![{}^{\mathrm{k}}bid(a, \mathtt{list}(\bar{v}))]\!]_{B(a)}$

$u2k(B, ([\![a', R, c, Q]\!]_a\ ,\ [\![a]\!]_{a'})) \triangleq$

$\quad [\![\mathtt{ready}(\mathtt{Qwaiting}(B(a), u2k(Q), \mathtt{nil}))]\!]_a\ ,\ [\![\mathtt{ready}(\mathtt{RpcWait}(k))]\!]_{B(a)}\ ,\ [\![\mathtt{ready}(\mathtt{RpcAux}(B(a)))]\!]_{a'}$

where $k = \lambda x.u2k^*(R[\![x]\!], c, a)$

The final clause above relies on the first clause (**uc.1**) of (**well-formedness invariant**) of user configuration computation steps, found in §4. The following lemma says that the user-to-kernel translation commutes with the meaning function on programs. This formalizes the commuting diagram of §1.

Lemma ($u2k$.2): For any user program, ${}^{\mathrm{u}}P$, $[\![u2k({}^{\mathrm{u}}P)]\!] = u2k([\![{}^{\mathrm{u}}P]\!])$

Proof : Let ${}^{\mathrm{u}}P$ be given by

$$\begin{aligned}{}^{\mathrm{u}}P \quad \triangleq \quad & \texttt{program}(\texttt{receptionists} : \rho, \quad \texttt{externals} : \chi \\ & \qquad \texttt{library} : Lib \\ & \qquad \texttt{actors} : \{a_j := e_j\}_{1\le j\le m} \\ & \qquad \texttt{messages} : \{a'_j \lhd {}^{\mathrm{u}}M_j\}_{1\le j\le n})\end{aligned}$$

then

$$[\![{}^{\mathrm{u}}P]\!] = \left\langle\!\!\left\langle\, {}^{\mathrm{u}}I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} \quad \text{where} \quad {}^{\mathrm{u}}I = \{[\![e_j, \texttt{nil}, \texttt{nil}]\!]_{a_j}\}_{1\le j\le m} \ , \ \{a'_j \lhd {}^{\mathrm{u}}M_j\}_{1\le j\le n}.$$

So by the above definition

$$u2k([\![{}^{\mathrm{u}}P]\!]) = \left\langle\!\!\left\langle\, u2k(B, {}^{\mathrm{u}}I) \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$$

where

$$u2k(B, {}^{\mathrm{u}}I) = \{[\![\texttt{Qwaiting}(B(a_j), \texttt{nil}, \texttt{nil})]\!]_{a_j} \ , \ [\![u2k^*(e_j, \texttt{nil}, a_j)]\!]_{B(a_j)}\}_{1\le j\le m} \ , \ \{a'_j \lhd {}^{\mathrm{u}}M_j\}_{1\le j\le n}.$$

On the other hand

$$\begin{aligned}[\![u2k({}^{\mathrm{u}}P)]\!] = [\![\texttt{program}(&\texttt{receptionists} : \rho \quad \texttt{externals} : \chi \\ &\texttt{library} : u2k({}^{\mathrm{u}}Lib) \\ &\texttt{actors} : \quad \{a_i := \texttt{Qwaiting}(B(a_i), \texttt{nil}, \texttt{nil})\}_{1\le i\le m} \\ &\qquad\quad \{B(a_i) := u2k^*(e_i, \texttt{nil}, a_i)\}_{1\le i\le m} \\ &\texttt{messages} : \{a'_j \lhd M_j\}_{1\le i\le n})]\!] \\ = \left\langle\!\!\left\langle\, {}^{\mathrm{k}}I \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}&\end{aligned}$$

where

$${}^{\mathrm{k}}I = \{[\![\texttt{Qwaiting}(B(a_j), \texttt{nil}, \texttt{nil})]\!]_{a_j} \ , \ [\![u2k^*(e_j, \texttt{nil}, a_j)]\!]_{B(a_j)}\}_{1\le j\le m} \ , \ \{a'_j \lhd {}^{\mathrm{u}}M_j\}_{1\le j\le n}$$

□

5.3.2 Second Kernel Transform 2

Definition (${}^{\mathrm{k}}AT_{\mathrm{d}}$): ${}^{\mathrm{k}}AT_{\mathrm{d}}$ is the extended actor theory obtained from ${}^{\mathrm{k}}AT_{\mathrm{r}}$ by defining the admissible path set ${}^{\mathrm{k}}\mathcal{A}$ to be:

$${}^{\mathrm{k}}\mathcal{A} = \{delay(\pi) \ \mid \ \pi \in \mathcal{F} : {}^{\mathrm{k}}AT_{\mathrm{r}}\}$$

Where $delay(\pi)$ is defined for $\pi \in \mathcal{F} : {}^{\mathrm{k}}AT_{\mathrm{r}}$ as follows. First let $D(\pi) = \{[i, j] \in \mathbf{N} \times (\mathbf{N} \cup \{\infty\})\}$ such that $\pi(i) = {}^{\mathrm{k}}K_i \xrightarrow{l_i} {}^{\mathrm{k}}K_{i+1}$ where l_i is the delivery of an external message to a non-`Qidle` mail-queue actor, and $\pi(j)$ is similar with l_j being the transition moving the message delivered at stage i from the pending queue of the mail-queue actor to the unchecked queue (or there is no such transition, in which case $j = \infty$). Thus j corresponds to a transition either of the form $\mathbf{State\ 4} \rightarrow \mathbf{State\ 2}$ where there are no unchecked messages, or $\mathbf{State\ 4} \rightarrow \mathbf{State\ 2}$ again where there are no unchecked messages in the parameterized finite state machine of figure 2. In either case the message delivered at stage i must be on the top of the pending queue immediately prior to the jth transition. $delay(\pi)$ is obtained from π by omitting the transitions l_i for $[i, j] \in D(\pi)$, leaving the message packet in the undelivered pool until the jth step. The jth transition now will be to the `Qidle` state (either of the $\mathbf{State\ 4} \rightarrow \mathbf{State\ 1}$ or $\mathbf{State\ 8} \rightarrow \mathbf{State\ 1}$ variety) since all prior pending messages are now undelivered. The delivery of the message is now inserted immediately after the jth transition (a $\mathbf{State\ 1} \rightarrow \mathbf{State\ 2}$ transition).

Lemma (kernel transform.2): ${}^{\mathrm{k}}AT_{\mathrm{r}} \gg {}^{\mathrm{k}}AT_{\mathrm{d}}$.

Proof : The is easy to see, since the delay operation does not move or modify any interactions. □

Lemma (${}^{\mathrm{k}}AT_{\mathrm{d}}$ invariance): (**kernel group**) and (**kernel fsm**) holds for ${}^{\mathrm{k}}AT_{\mathrm{d}}$ and the pending queues of mail-queue actors are always `nil` and hence can be ignored.

5.3.3 Third Kernel Transform 3

The final kernel theory transformation is a generalized big step transformation, treating communications local to a group as silent.

Definition (${}^{\mathrm{k}}AT_{\mathrm{g}}$): The actor theory ${}^{\mathrm{k}}AT_{\mathrm{g}}$ has the same actors, messages, states and rules as ${}^{\mathrm{k}}AT_{\mathrm{d}}$. The admissible paths $\mathcal{A} : {}^{\mathrm{k}}AT_{\mathrm{g}}$ are those paths of $\mathcal{A} : {}^{\mathrm{k}}AT_{\mathrm{d}}$ that can be express as sequences of macro steps of the form IO or $Local; Eff$ where

1. IO is an interaction step — input, output, or `idle`
2. $Local$ is a sequence of local steps – silent, or sends/delivers of messages internal to a group associated to a single behavior actor.
3. Eff is one of the following:
   (a) a translated `letactor` – creating mail–queue/behavior pairs;
   (b) a translated asynchronous user message `send` – a single kernel `send`;
   (c) a translated remote procedure call – a single `letactor` followed by `send`;
   (d) delivery to an idle mail–queue actor of a message from outside the group, followed by the forwarding of the message to the behavior actor;
   (e) delivery to an auxiliary actor of a message from outside the group, followed by forwarding to and receipt of the message by the behavior actor.

Each such path is observationally fair:

— all messages to external actors are output;

— if an auxiliary actor is enabled for delivery, and there is a message to be delivered, then some message is delivered;

— if the group associated with $b \in Beh$ is enabled then it will progress;

— if a group is enabled for delivery to the mail–queue actor infinitely often, then all pending messages to that actor get delivered.

A group $G(b)$ is enabled if there is a sequence of local steps leading to an effect step and the effect is not delivery, or there is an available message. Progress means that there is a macro step carrying out the local sequence and effect. Locally permanently silent actors — behavior actors that either hang or go on silently forever are ignored. The local sequence will be unique. The effect is unique upto choice of created actor names or choice of delivered message.

Lemma (kernel transform.3): ${}^{\mathrm{k}}AT_{\mathrm{d}} \gg {}^{\mathrm{k}}AT_{\mathrm{g}}$

Proof : The proof is analogous to the proof that the big step transform preserves interaction semantics. □

5.3.4 Final Step

To complete the proof of theorem ($u2k$) we establish (**final step**):

Lemma (final step):

$$Isem({}^{\mathrm{u}}K : {}^{\mathrm{u}}AT^{\dagger}) = Isem(u2k({}^{\mathrm{u}}K) : {}^{\mathrm{k}}AT_{\mathrm{g}})$$

This is proved by establishing a correspondence between the macro steps of ${}^{\mathrm{u}}AT^{\dagger}$ and those of ${}^{\mathrm{k}}AT_{\mathrm{g}}$.

We let ${}^{\mathrm{u}}L$ range over sequences of labels that make up the steps of a computation in ${}^{\mathrm{u}}AT^{\dagger}$ and ${}^{\mathrm{k}}L$ range over sequences of labels that make up the steps of a computation in ${}^{\mathrm{k}}AT_{\mathrm{g}}$. The heart of the argument is showing that computations of ${}^{\mathrm{u}}K : {}^{\mathrm{u}}AT^{\dagger}$ and $u2k({}^{\mathrm{u}}K) : {}^{\mathrm{k}}AT_{\mathrm{g}}$ have corresponding macro steps. This is expressed by lemmas (*u2k*.**sim.1**) and (*u2k*.**sim.2**). We first show how these lemmas are used to prove (**final step**) Then we prove the lemmas.

Lemma (*u2k*.sim.1): If ${}^{\mathrm{u}}K \xrightarrow{{}^{\mathrm{u}}L} {}^{\mathrm{u}}K'$ with ${}^{\mathrm{u}}L$ a macro step label of the user big step form then there is ${}^{\mathrm{k}}L$ a macro step label of the user-kernel canonical form such that $isem({}^{\mathrm{u}}L) = isem({}^{\mathrm{k}}L)$ and $u2k({}^{\mathrm{u}}K) \xrightarrow{{}^{\mathrm{k}}L} u2k({}^{\mathrm{u}}K')$.

Lemma (*u2k*.sim.2): If $u2k({}^{\mathrm{u}}K) \xrightarrow{{}^{\mathrm{k}}L} {}^{\mathrm{k}}K'$ with ${}^{\mathrm{k}}L$ a macro step label of the user-kernel canonical form then there is L a macro step label of the user big step form such that $isem({}^{\mathrm{u}}L) = isem({}^{\mathrm{k}}L)$ and a user configuration ${}^{\mathrm{u}}K'$ such that $u2k({}^{\mathrm{u}}K') = {}^{\mathrm{k}}K'$ and ${}^{\mathrm{u}}K \xrightarrow{{}^{\mathrm{u}}L} {}^{\mathrm{u}}K'$.

Lemma (final step):

$$Isem({}^{\mathrm{u}}K : {}^{\mathrm{u}}AT^{\dagger}) = Isem(u2k({}^{\mathrm{u}}K) : {}^{\mathrm{k}}AT_{\mathrm{g}})$$

Proof : To show $Isem({}^{\mathrm{u}}K : {}^{\mathrm{u}}AT^{\dagger}) \supseteq Isem(u2k({}^{\mathrm{u}}K) : {}^{\mathrm{k}}AT_{\mathrm{g}})$, let

$${}^{\mathrm{u}}\pi = [{}^{\mathrm{u}}K_i \xrightarrow{{}^{\mathrm{u}}L_i} {}^{\mathrm{u}}K_{i+1} \mid i \in \mathbf{N}]$$

be a user path in big-step form, define $u2k({}^{\mathrm{u}}\pi)$ by

$$u2k({}^{\mathrm{u}}\pi) = [u2k({}^{\mathrm{u}}K_i) \xrightarrow{{}^{\mathrm{k}}L_i} u2k({}^{\mathrm{u}}K_{i+1}) \mid i \in \mathbf{N}]$$

where ${}^{\mathrm{k}}L_i$is given by lemma (*u2k*.**sim.1**). We claim $u2k({}^{\mathrm{u}}\pi)$ is an admissible path of ${}^{\mathrm{k}}AT_{\mathrm{g}}$ starting from $u2k({}^{\mathrm{u}}K)$. Clearly $isem({}^{\mathrm{u}}\pi) = isem(u2k({}^{\mathrm{u}}\pi))$.

Conversely, to show $Isem({}^{\mathrm{u}}K : {}^{\mathrm{u}}AT) \subseteq Isem(u2k({}^{\mathrm{u}}P) : {}^{\mathrm{k}}AT_{\mathrm{g}})$, let

$${}^{\mathrm{k}}\pi = [{}^{\mathrm{k}}K_i \xrightarrow{{}^{\mathrm{k}}L_i} {}^{\mathrm{k}}K_{i+1} \mid i \in \mathbf{N}]$$

be a kernel path (in ${}^{\mathrm{k}}AT_{\mathrm{g}}$) in big-step form where ${}^{\mathrm{k}}K_0 = u2k({}^{\mathrm{u}}K_0)$ for some ${}^{\mathrm{u}}K_0 : {}^{\mathrm{u}}AT$. Define

$${}^{\mathrm{u}}\pi = [{}^{\mathrm{u}}K_i \xrightarrow{{}^{\mathrm{u}}L_i} {}^{\mathrm{u}}K_{i+1} \mid i \in \mathbf{N}]$$

where each ${}^{\mathrm{u}}L_i$ is obtained via induction and lemma (*u2k*.**sim.2**). Again $isem({}^{\mathrm{u}}\pi) = isem(u2k({}^{\mathrm{u}}\pi))$. □

5.3.5 Stepwise Correspondence

To establish the macro step correspondence lemmas we begin by examining ${}^{\mathrm{u}}AT^{\dagger}$ and ${}^{\mathrm{k}}AT_{\mathrm{g}}$ in a little more detail. Firstly observe that the non-silent transitions are labelled by `send`, `rpc`, `rcv`, `deliver`, and `leta`. The silent transitions are labelled `seq`, `walk`, `cstr`, `enable`, `disable` and `check`. Since we are interested in configurations in which initially all actors are in states of the form $[\,e, c, Q\,]$, we can restrict the states of ${}^{\mathrm{u}}AT^{\dagger}$ to those of the form

(**State 4**) $[\,e, c, Q\,]_a$

(**State 4'**) $[\,a', R, c, Q\,]_a$, $[\,a\,]_{a'}$

(**State 2**) $[\,bid(\bar{v}), [{}^{\mathrm{u}}M], Q_{\mathrm{r}}\,]_a$

since these are the states that result from carrying out a big step. The names for these states correspond to those of the user finite state machine, figure 1, of §4. (**State 4**) arises either from the initial starting configuration, or after

completing a `send`, an `rcv`, or a `leta`. (**State 4'**) arises after an `rpc`. While (**State 2**) arises after a `deliver`. The corresponding states, via $u2k$, on the kernel side are:

(**State 7**) $[\texttt{ready}(\texttt{Qwaiting}(B(a), u2k(Q), \texttt{nil}))]_a$, $[u2k^*(e, c, a)]_{B(a)}$

(**State 7'**) $[\texttt{ready}(\texttt{Qwaiting}(B(a), u2k(Q), \texttt{nil}))]_a$, $[\texttt{ready}(\texttt{RpcWait}(k))]_{B(a)}$, $[\texttt{ready}(\texttt{RpcAux}(B(a)))]_{a'}$

where $k = \lambda x.u2k^*(R[\![x]\!], c, a)$

(**State 2**) $[\texttt{ready}(\texttt{Qwalking}(B(a), \texttt{nil}, {}^{\texttt{u}}M, Q_r, \texttt{nil}))]_a$, $B(a) \triangleleft {}^{\texttt{u}}M$, $[{}^{\texttt{k}}bid(a, \texttt{list}(\bar{v}))]_{B(a)}$

The names for these states correspond to those of the $u2k$ protocol finite state machine, figure 2, of §5 (upto β-reduction of ${}^{\texttt{k}}bid(a, \texttt{list}(\bar{v}))$ in the case of (**State 2**)).

The (big)steps of ${}^{\texttt{u}}AT^{\dagger}$ can be classified as sequences of ${}^{\texttt{u}}AT$ steps with labels of one of the following forms (where $\texttt{nextmsg}(a)$ is either $\texttt{walk}(a)$ or $\texttt{disable}(a)$), let

$${}^{\texttt{u}}\mathrm{L}_{\mathrm{bs}}(a) \triangleq [\texttt{seq}(a)^*; [\texttt{nextmsg}(a); \texttt{cstr}(a); \texttt{check}(a)^*]^+; \texttt{enable}(a)]^*; \texttt{seq}(a)^*$$

Note that the first $\texttt{nextmsg}(a)$ in such a sequence must be a $\texttt{walk}(a)$, while subsequent ones generated by the term $[\texttt{nextmsg}(a); \texttt{cstr}(a); \texttt{check}(a)^*]^+$ will be the $\texttt{disable}(a)$ variety. Then the bigsteps are of one of the following forms, where in (1–4) the actor a starts in (**State 4**) or (**State 2**):

1. ${}^{\texttt{u}}\mathrm{L}_{\mathrm{bs}}(a); \texttt{send}(a)$ that has the effect of sending a message, the actor a finishes in (**State 4**).
2. ${}^{\texttt{u}}\mathrm{L}_{\mathrm{bs}}(a); \texttt{rpc}(a)$ that has the effect sending an remote procedure call request, and creating an auxiliary actor to handle the reply. The actor a finishes in (**State 4'**).
3. ${}^{\texttt{u}}\mathrm{L}_{\mathrm{bs}}(a); \texttt{leta}(a, \vec{a})$ that has the effect of creating the actors $\vec{a}$, the actor a finishes in (**State 4**). These three big steps correspond to a path of the form

   $$((\mathbf{State\ 4} \rightarrow)^+(\mathbf{State\ 2} \rightarrow (\mathbf{State\ 3} \rightarrow)^+)^*)^*(\mathbf{State\ 4} \rightarrow)^*\mathbf{State\ 4}$$

   in the user FSM of figure 1.
4. ${}^{\texttt{u}}\mathrm{L}_{\mathrm{bs}}(a); [\texttt{nextmsg}(a); \texttt{cstr}(a); \texttt{check}(a)^*]^*; \texttt{disable}(a); \texttt{deliver}(a, {}^{\texttt{u}}M)$ that has a delivery effect. The actor a finishes in (**State 2**).

   $$(\mathbf{State\ 4} \rightarrow (\mathbf{State\ 2} \rightarrow (\mathbf{State\ 3} \rightarrow)^+)^*(\mathbf{State\ 4} \rightarrow)^*)^* \rightarrow \mathbf{State\ 1} \rightarrow \mathbf{State\ 2}$$

   in the user FSM of figure 1.
5. $\texttt{rcv}(a, a_0, v)$ leading from (**State 4'**) to (**State 4**). This correspond to the transition:

   $$\mathbf{State\ 4'} \rightarrow \mathbf{State\ 4}$$

We can make the correspondence between internal user transitions and local transitions within the corresponding group explicit:

$\texttt{Seq}(B, a) \triangleq \texttt{seq}(B(a))$

$\texttt{Walk}(B, a) \triangleq \texttt{send}(B(a), a, \texttt{WALK[]@}B(a)); \texttt{ready}(a, \texttt{WALK[]@}B(a))$

$\texttt{Disable}(a) \triangleq \texttt{send}(B(a), a, \texttt{NEXT[]@}B(a)); \texttt{ready}(a, \texttt{NEXT[]@}B(a))$

$\texttt{Enable}(a) \triangleq \texttt{send}(B(a), a, \texttt{ENABLED[]@}B(a)); \texttt{ready}(a, \texttt{ENABLED[]@}B(a)); \texttt{seq}(a)^*;$
$\texttt{send}(a, B(a), \texttt{OK[]@}a); \texttt{ready}(B(a), \texttt{OK[]@}a)$

$\texttt{Cstr}(a) \triangleq \texttt{seq}(a)^*; \texttt{send}(a, B(a), M); \texttt{ready}(B(a), M)$ for some appropriate M

$\texttt{Check}(a) \triangleq \texttt{seq}(B(a))$

Note that we are making a harmless identification between the user auxiliary actor, and the corresponding unique active kernel auxiliary actor within the group. We can similarly make the correspondence between the non-silent internal user transitions and local transitions within the corresponding group explicit:

$\texttt{Send}(B, a) \triangleq \texttt{send}(B(a), v_0, v_1)$ for some appropriate v_0, v_1

$\texttt{Rpc}(B, a, a_0) \triangleq \texttt{leta}(B(a), a_0); \texttt{send}(B(a), a_0, M); \texttt{seq}(B(a))^*$ for some appropriate M

$\texttt{Rcv}(a, a_0, M) \triangleq \texttt{ready}(a_0, M); \texttt{seq}(a_0)^*; \texttt{send}(a_0, B(a), M); \texttt{ready}(B(a), M)$

$\texttt{Leta}(B, a, \vec{a}) \triangleq \texttt{leta}(a, \vec{a} * B(\vec{a}))$

Note that some kernel labels contain more information than their corresponding user counterparts, the actual values for these can be easily gleaned from either the user computation, or the corresponding kernel computation. Using this correspondence we can make transparent the correspondence between the (big)steps of ${}^{\mathrm{u}}AT^{\dagger}$ and the (big)steps of ${}^{\mathrm{k}}AT_{\mathrm{g}}$.

The (big)steps of ${}^{\mathrm{k}}AT_{\mathrm{g}}$ can be classified as sequences of these macro steps with labels of one of the following forms (leaving the B argument implicit for simplicity). As in the user case $\texttt{Nextmsglab}(a)$ is either $\texttt{Walk}(a)$ or $\texttt{Disable}(a)$). Then

${}^{\mathrm{k}}\mathrm{L}_{\mathrm{bs}}(a) \triangleq [\texttt{Seq}(a)^*; [\texttt{Nextmsglab}(a); \texttt{Cstr}(a); \texttt{Check}(a)^*]^+; \texttt{Enable}(a)]^*; \texttt{Seq}(a)^*$

Note that the first $\texttt{Nextmsglab}(a)$ in such a sequence must be a $\texttt{Walk}(a)$, while subsequent ones generated by the term $[\texttt{Nextmsglab}(a); \texttt{Cstr}(a); \texttt{Check}(a)^*]^+$ will be the $\texttt{Disable}(a)$ variety. Then the bigsteps are of one of the following forms, where in (1–4) the group corresponding to a starts in (**State 7**) or (**State 2**):

1. ${}^{\mathrm{k}}\mathrm{L}_{\mathrm{bs}}(a); \texttt{Send}(a)$ has the effect of sending a message, the group finishes in (**State 7**).
2. ${}^{\mathrm{k}}\mathrm{L}_{\mathrm{bs}}(a); \texttt{Rpc}(a)$ has the effect sending an remote procedure call request, and creating an auxillary actor to handle the reply. The group finishes in (**State 7'**).
3. ${}^{\mathrm{k}}\mathrm{L}_{\mathrm{bs}}(a); \texttt{Leta}(a, \vec{a})$ has the effect of creating the mail-queue actors $\vec{a}$ and the corresponding behavior actors $B(\vec{a})$. The group finishes in (**State 7**).
4. ${}^{\mathrm{k}}\mathrm{L}_{\mathrm{bs}}(a); [\texttt{Nextmsglab}(a); \texttt{Cstr}(a); \texttt{Check}(a)^*]^*; \texttt{Disable}(a); \texttt{Deliver}(a, {}^{\mathrm{u}}M)$ has a delivery effect. The group finishes in (**State 2**).
5. $\texttt{Rcv}(a, a_0, v)$ leading from (**State 7'**) to (**State 7**).

Next we establish some properties relating reduction and the user to kernel translation. In what follows we fix a user library Lib. We extend $u2k$ to reduction contexts by defining:

$u2k(\bullet) \triangleq \bullet$

An *administrative reduction* is a β reduction of an expression of the form $\texttt{app}(\lambda c.\lambda q.e, c, q)$. In the following we consider kernel expressions in the image of the translation to be equal if they differ only by administrative reductions. The following lemma asserts that the translation of an expression commutes with the decomposition of the expression into a reduction context filled with a redex.

Lemma ($u2k$.rcx.1): $u2k(R[\![e]\!]) \equiv u2k(R)[\![u2k(e)]\!]$

Proof ($u2k$.rcx.1): An easy induction of the construction of R, since $\bullet$'s always appear in places where $u2k$ applies the result of a simple recursive call to a c and q. $\square_{u2k.\mathbf{rcx.1}}$

Now we establish two lemmas to show that either a user actor and its kernel translation group are both permanently silent, or there are silent/local steps leading to corresponding effects. The first deals with the purely functional part of big steps. The second deals with the user `ready` redex, which because of the internal queue management is either hung, or silently moves to an idle state waiting for a message delivery, or to the execution of a method body.

Lemma ($u2k$.big.1): Let ${}^{\mathrm{u}}e$ be a user expression, ${}^{\mathrm{u}}\zeta$ be a user context with $self({}^{\mathrm{u}}\zeta) = a$ and $customer({}^{\mathrm{u}}\zeta) = c$, ${}^{\mathrm{k}}e = u2k^*({}^{\mathrm{u}}e, c, a)$ and ${}^{\mathrm{k}}\zeta$ a kernel context with $self({}^{\mathrm{k}}\zeta) = B(a)$

(1) ${}^{\mathrm{u}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{u}\zeta} v$ iff ${}^{\mathrm{k}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{k}\zeta} v$.

(2) ${}^{\mathrm{u}}e$ reduces via $\xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{u}\zeta}$ steps to a hung state, or has infinite $\xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{u}\zeta}$ steps iff ${}^{\mathrm{k}}e$ reduces via $\xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{k}\zeta}$ steps to a hung state, or has infinite $\xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{k}\zeta}$ steps.

(3) ${}^{\mathrm{u}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{u}\zeta} {}^{\mathrm{u}}R[\![v \triangleleft mid[\bar{v}]@c']\!]$ iff ${}^{\mathrm{k}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{k}\zeta} u2k^*({}^{\mathrm{u}}R, c, a)[\![\mathtt{send}(v, mid[\bar{v}]@c')]\!]$

(4) ${}^{\mathrm{u}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{u}\zeta} {}^{\mathrm{u}}R[\![v\,\mathtt{.}\,mid[\bar{v}]]\!]$ iff ${}^{\mathrm{k}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{k}\zeta} u2k^*({}^{\mathrm{u}}R, c, a)[\![{}^{\mathrm{k}}e_r]\!]$

where ${}^{\mathrm{k}}e_r = \mathtt{letactor}\{a_{\mathrm{aux}} := \mathtt{ready}(\mathtt{RpcAux}(\mathtt{self}()))\}$
$\mathtt{seq}(\mathtt{send}(v, mid[\bar{v}]@a_{\mathrm{aux}}), \mathtt{clc}(\lambda k.\mathtt{ready}(\mathtt{RpcWait}(k))))$

(5) ${}^{\mathrm{u}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{u}\zeta} {}^{\mathrm{u}}R[\![\mathtt{letactor}\{a_i := {}^{\mathrm{u}}e_i\}_{1\le i\le n}\ {}^{\mathrm{u}}e_0]\!]$ iff ${}^{\mathrm{k}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{k}\zeta} u2k^*({}^{\mathrm{u}}R, c, a)[\![{}^{\mathrm{k}}e_r]\!]$

where ${}^{\mathrm{k}}e_r = \mathtt{letactor}\{a_i := \mathtt{Qidle}(a_i^*, \mathtt{nil}),$
$a_i^* := u2k^*({}^{\mathrm{u}}e_i, c, a_i)\}_{1\le i\le n}$
$u2k^*({}^{\mathrm{u}}e_0, c, a)$

for a_i^* pairwise distinct and fresh $1 \le i \le n$

(6) ${}^{\mathrm{u}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{u}\zeta} {}^{\mathrm{u}}R[\![\mathtt{ready}(bid(\bar{v}))]\!] \wedge behMatch(Lib, bid, \bar{v})$ iff ${}^{\mathrm{k}}e \xrightarrow{s}\!\!\!\!\rightarrow_{\mathrm{k}\zeta} \mathtt{app}({}^{\mathrm{k}}bid, a, \bar{v})$

In **(2)** we consider $\mathtt{ready}(bid(\bar{v}))$ to be hung if $behMatch(Lib, bid, \bar{v})$ fails to hold. Note that if ${}^{\mathrm{u}}e$ is a disabling constraint, and hence uses no actor primitives other than `customer` and `self`, then **(1)** and **(2)** show that ${}^{\mathrm{u}}e$ and ${}^{\mathrm{k}}e$ have the same evaluation properties.

Proof : This is a routine case analysis on the decomposition of ${}^{\mathrm{u}}e$ into a redex and a reduction context, and the fact that modulo the actor primitives, $u2k$ preserves this reduction.

□

Now, assume `behavior` $bid(p)(methodDefs) \in Lib$, $parCheck(p, \bar{v})$, and $MD = u2k(methodDefs, p)$. Then

$[\![\mathtt{ready}(bid(\bar{v})), c, Q]\!]_a$ moves silently to $[\![bid(\bar{v}), Q, [\,]]\!]_a$

and

$[\![\mathtt{ready}(\mathtt{Qwaiting}(B(a), u2k(Q), \mathtt{nil}))]\!]_a$, $[\![\mathtt{app}({}^{\mathrm{k}}bid, a, \bar{v})]\!]_{B(a)}$

moves by local steps to (assuming $Q = [M] * Q'$)

$[\![\mathtt{ready}(\mathtt{Qwalking}(B(a), u2k(Q'), M, \mathtt{nil}, \mathtt{nil}))]\!]_a$, $[\![\mathtt{ready}(\mathtt{Wait4Q}(MD, a, \bar{v}))]\!]_{B(a)}$, $B(a) \triangleleft M$

thus we are interested in the correspondence between user states

$$[\![bid(\bar{v}), [M] * Q_u, Q_r]\!]_a$$

and the kernel translation

$[\![\mathtt{ready}(\mathtt{Qwalking}(B(a), u2k(Q_u), M, u2k(Q_r), \mathtt{nil}))]\!]_a$, $[\![\mathtt{ready}(\mathtt{Wait4Q}(MD, a, \bar{v}))]\!]_{B(a)}$, $B(a) \triangleleft M$

Lemma (*u2k*.big.2): Assume

$$\mathtt{behavior}\ bid(p)(methodDefs) \in Lib, parCheck(p, \bar{v}), \quad \text{and} \quad MD = u2k(methodDefs, p).$$

Define

${}^{\mathrm{u}}I(bid, \bar{v}, M, Q_u, Q_r) = [\![bid(\bar{v}), [M] * Q_u, Q_r]\!]_a$

${}^{\mathrm{k}}I(bid, \bar{v}, M, u2k(Q_u), u2k(Q_r)) =$

$[\![\mathtt{ready}(\mathtt{Qwalking}(B(a), u2k(Q_u), M, u2k(Q_r), \mathtt{nil}))]\!]_a$, $[\![\mathtt{ready}(\mathtt{Wait4Q}(MD, a, \bar{v}))]\!]_{B(a)}$, $B(a) \triangleleft M$

Then

1 ${}^{\mathrm{u}}I(bid, \bar{v}, M, Q_u, Q_r)$ is permanently silent iff ${}^{\mathrm{k}}I(bid, \bar{v}, M, u2k(Q_u), u2k(Q_r))$

2 ${}^{\mathrm{u}}I(bid, \bar{v}, M, Q_u, Q_r)$ moves silently to $[\,bid(\bar{v}), [\,], Q_r\,]_a$ iff ${}^{\mathrm{k}}I(bid, \bar{v}, M, u2k(Q_u), u2k(Q_r))$ moves locally to $[\,\mathtt{ready}(\mathtt{Qidle}(B(a), u2k(Q_r)))\,]_a$, $[\,\mathtt{ready}(\mathtt{Wait4Q}(MD, a, \bar{v}))\,]_{B(a)}$

3 ${}^{\mathrm{u}}I(bid, \bar{v}, M, Q_u, Q_r)$ moves silently to $[\,{}^{\mathrm{u}}e_m, c, Q\,]_a$ iff ${}^{\mathrm{k}}I(bid, \bar{v}, M, u2k(Q_u), u2k(Q_r))$ moves locally to $[\,\mathtt{ready}(\mathtt{Qwaiting}(B(a), u2k(Q), \mathtt{nil}))\,]_a$, $[\,u2k^*({}^{\mathrm{u}}e_m, c, a)\,]_{B(a)}$

And these three cases are exhaustive.

Proof : The proof is by induction on Q_u, using the lemmas (**MethodDef**) and (*u2k*.**big.1**) to show that user and kernel method lookup and constraint checking give corresponding results. □

We complete the details by indicating how the macro step correspondence lemmas are proved.

Proof (*u2k*.sim.1): Assume ${}^{\mathrm{u}}K \xrightarrow{{}^{\mathrm{u}}L} {}^{\mathrm{u}}K'$ with L a macro step label of the user big step form. We consider cases according to the analysis of the macro steps in ${}^{\mathrm{u}}AT^{\dagger}$. If there are no `nextmsg`... sequences, then we use the corresponding case of (*u2k*.**big.1**) to show the correspondence. Otherwise, by the ready case of (*u2k*.**big.1**) both user and translated states step silently to a `ready` and we use lemma (*u2k*.**big.2**) to complete the argument. □

Proof (*u2k*.sim.2): The proof is similar to the proof of (*u2k*.**sim.2**) using the analysis of user-kernel canonical steps. □

6 Conclusions

The main technical contribution of this paper is to present a method for establishing equivalence of actor systems, or more generally for distributed object-based systems. The main result of this paper is a proof of correctness of what is essentially a stage of compilation of a high-level actor language.

In [PT94] high-level object programming constructs are explained by expansion in the Pict language. In [Wal95] a semantics for a variant of POOL is given via translation to a sorted Pi calculus. This is shown to be a simulation (up to bisimulation) of a direct transition system operational semantics of POOL. Core Facile is a synthesis of the typed lambda calculus and pi-calculus style concurrency primitives. In [Ama94] a translation from Core Facile to a variant based on asynchronous communication is given. The translation of a process is shown to preserve barbed bisimilarity and barbed congruence of the translation of two expressions implies congruence of the expressions. The converse is left open. The translation goes by an intermediate language obtained by adding a control operator to the asynchronous Facile much as we have done in the kernel language. In [AP94] an extension of the Pi-calculus to model locality and failure is translated in to a simply sorted Pi-calculus and similar properties are proved for the translation. Our approach differs in giving both languages an abstract, composable semantics in the same semantic domain and showing that the translation preserves the abstract semantics. The notion of barbed bisimulation seems to share with abstract actor structures and interaction semantics the objective of hiding details of internal computation. More detailed investigation of the relation between these approaches is an interesting topic for future work.

Acknowledgements

The authors would like to thank both Gul Agha and Scott Smith for numerous discussions concerning the contents of this paper. Gul Agha was kind enough to point out a fairness problem in an early version of the translation, while Scott Smith provided detailed corrections to an earlier draft. An early version of this paper appears as [MT97] and we thank the three anonymous ICALP referees and the three anonymous TCS referees for many helpful comments and corrections. This work was done while the first author was partially supported by Australian Research Council grant IA131.84 and U.N.E University Research Grant IB34.1. The second author was partially supported by ONR grant N00014-94-1-0857, NSF grant CCR-9312580, and ARPA/SRI subcontract C-Q0483, ARPA/AirForce grant F30602-96-1-0300, NSF grant CRR-9633419.

References

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

[Agh90] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[Ama94] R. M. Amadio. Translating core facile. Technical Report ECRC-1994-3, European Computer-Industry Research Centre, 1994.

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

[AP94] R. M. Amadio and S. Prasad. Localities and failures. Technical Report ECRC-1994-18, European Computer-Industry Research Centre, 1994.

[BB92] G Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BH77] Henry G. Baker and Carl Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, August 1977.

[Cli81] W. D. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[FF86] M. Felleisen and D.P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

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

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes96] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Concur96*, 1996.

[MT97] I. A. Mason and C. L. Talcott. A semantics preserving actor translation. In *ICALP'97*, 1997. proceedings of the 24th International Colloquium on Automata, Languages, and Programming.

[Plo75] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PT94] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan*, Lecture Notes in Computer Science. Springer-Verlag, November 1994. To appear, 1995.

[Tal96a] C. L. Talcott. An actor rewriting theory. In *Workshop on Rewriting Logic*, number 4 in Electronic Notes in Theoretical Computer Science, 1996.

[Tal96b] 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.

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

[Tal98] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3), 1998.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.