An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language

Gul Agha and Prasanna Thati

University of Illinois at Urbana-Champaign, USA
{agha,thati}@cs.uiuc.edu
http://osl.cs.uiuc.edu/

1 Introduction

The development of Simula by Ole-Johan Dahl and Kristen Nygaard introduced a number of important programming language concepts – *object* which supports modularity in programming through encapsulation of data and procedures, the concept of *class* which organizes behavior and supports Abstract Data Types, and the concept *inheritance* which provides subtyping relations and reuse [6]. Peter Wegner terms programming languages which use objects as *object-based languages*, and reserves the term object-oriented languages for languages which also support classes and inheritance [58].

Concurrency provides a natural model for the execution of objects: in fact, Simula uses co-routines to simulate a simple form of concurrency in a sequential execution environment. The resulting execution is tightly synchronized and, while this execution model is appropriate for simulations which use a global virtual clock, it is not an adequate model for distributed systems. The Actor Model unifies the notion of objects with concurrency; an actor is a concurrent object which operates asynchronously and interacts with other actors by sending asynchronous messages [2].

Many models for concurrent and distributed computation have been developed. An early and influential model is *Petri Nets* developed by Carl Adam Petri [44]. In the Petri Net model, there are two kinds of elements – nodes and tokens. Nodes are connected to other nodes by fixed (static) links. Tokens are passed between nodes using these links. The behavior of each node is governed by reactive rules which are triggered based on the presence of tokens at the nodes.

Another popular model of concurrency is based on communicating processes. Two exponents of this sort of model are Robin Milner who defined the *Calculus of Communicating Systems (CCS)* [38], and Tony Hoare who defined the programming language, *Communicating Sequential Processes (CSP)* [20]. In both these systems, asynchronous processes have a fixed communication topology (processes which can communicate with each other are statically determined) and the communication is synchronous – i.e. a message exchange involves an explicit handshake between the sender and the receiver.

In contrast to these models, the notion of actors is very flexible. In the earliest formulation of the Actor Model, an actor was defined by Carl Hewitt as an

O. Owe et al. (Eds.): From OO to FM (Dahl Festschrift), LNCS 2635, pp. 26–57, 2004.

autonomous agent which has intentions, resources, contain message monitors and a scheduler [19]. Later work by Hewitt and his associates developed a more abstract model of parallelism based on causal relations between asynchronous events at different actors – where an event represents the sending or receiving of a message [10, 16]. The formulation of the Actor Model that most people refer to is based on the transition system in Agha [1]. In particular, this formulation provides a basis for the operational semantics developed in [3].

The Actor Model is more flexible than Petri Nets, CCS or CSP. Petri Nets have been generalized to Colored Petri Nets which allow tokens to carry data. It is possible to encode actor computations in this more general model by interpreting actor behaviors although it is not clear how useful such an encoding is [42].

In fact, the work on actors inspired Robin Milner to develop the π-calculus [41], a model which is more general than CCS. As Milner reports: "... the pure λ-calculus is built with just two kinds of things: terms and variables. Can we achieve the same economy for a process calculus? Carl Hewitt, with his Actor Model, responded to this challenge a long ago; he declared that a value, an operator on values, and a process should all be the same kind of thing: an *actor*. This goal impressed me, because it implies a homogeneity and completeness of expression .... But it was long before I could see how to attain the goal in terms of an algebraic calculus .... So, in the spirit of Hewitt, our first step is to demand that all things denoted by terms or accessed by names–values, registers, operators, processes, objects–are all the same kind of thing; they should *all* be processes. Thereafter we regard access-by-name as the raw material of computation ...." [39].

The π-calculus allows names to be communicable – thus capturing an essential aspect of actors which provides it greater flexibility. However, there are number of differences between the two models that are caused by differing goals and ontological commitments. The goal of explicitly modeling distributed systems has motivated the development of actors, while the goal of providing an algebraic formulation has been central to work on π-calculus. As a consequence, the Actor Model uses asynchronous communication which is natural in distributed systems, while the π-calculus uses synchronous communication which results in a simpler algebra. As in object-based systems, each actor has a distinct identity which is bound to a unique name which does not change. By contrast, in the π-calculus, different processes can have the same name, and these names can disappear.

This paper develops a formal calculus for actors by imposing suitable type restrictions on the π-calculus. Our aim is to gain a better understanding of the implications of the different ontological commitments of the Actor Model. We present a typed variant of π-calculus, called Aπ, which is an accurate representation of the Actor Model, and we investigate a basic theory of process equivalence in Aπ. We then illustrate how Aπ can be used to provide formal semantics for actor-based concurrent programming languages. The Actor Model has served as the basis of a number of object-based languages [4, 59]. Since our aim is to investigate the effects of only the basic ontological commitments of the Actor

Model, we focus our presentation on a simple actor-based language which was first defined in [1].

Following is the layout of the rest of this paper. In Section 2, we give a brief and informal description of the Actor Model, and in Section 3, we describe a simple actor language (SAL). In Section 4, we present the calculus Aπ, and in Section 5, we investigate a basic theory of process equivalence in Aπ. In Section 6, we provide a formal semantics for SAL by translating its programs to Aπ. In Section 7, we conclude the paper with an overview of several other research directions that have been pursued on the basic Actor Model over the last two decades.

2 The Actor Model

A computational system in the Actor Model, called a *configuration*, consists of a collection of concurrently executing actors and a collection of messages in transit [1]. Each actor has a unique name (the *uniqueness* property) and a behavior, and communicates with other actors via asynchronous messages. Actors are reactive in nature, i.e. they execute only in response to messages received. An actor's behavior is deterministic in that its response to a message is uniquely determined by the message contents. Message delivery in the Actor Model is fair [10]. The delivery of a message can only be delayed for a finite but unbounded amount of time.

An actor can perform three basic actions on receiving a message (see Figure 1): (a) create a finite number of actors with universally fresh names, (b) send a finite number of messages, and (c) assume a new behavior. Furthermore, all actions performed on receiving a message are concurrent; there is no ordering between any two of them. The following observations are in order here. First, actors are persistent in that they do not disappear after processing a message (the *persistence* property). Second, actors cannot be created with well known names or names received in a message (the *freshness* property).

The description of a configuration also defines an interface between the configuration and its environment, which constrains interactions between the two. An interface is a set of names ρ, called the *receptionist* set, that contains names of all the actors in the configuration that are visible to the environment. The only way an environment can effect the actors in a configuration is by sending messages to the receptionists of the configuration; the non-receptionist actors are all hidden from the environment. Note that uniqueness of actor names automatically prevents the environment from receiving messages in configuration that are targeted to the receptionists, because to receive such messages the environment should have an actor with the same name as that of a receptionist. The receptionist set may evolve during interactions, as the messages that the configuration sends to the environment may contain names of actors are not currently in the receptionist set.

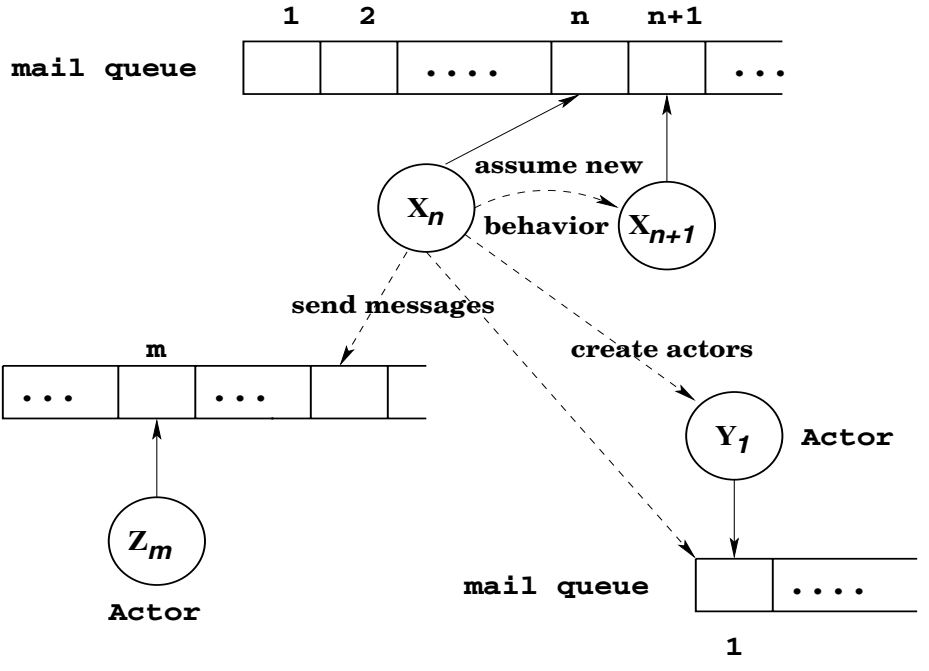

Fig. 1. A diagram illustrating computation in an actor system.

3 A Simple Actor Language (SAL)

A SAL program consists of a sequence of behavior definitions followed by a single (top level) command.

$$Pgm ::= BDef_1 \quad ... \quad BDef_n \quad Com$$

The behavior definitions are templates for actor behaviors. The top level command creates an initial collection of actors and messages, and specifies the interface of the configuration to its environment.

3.1 Expressions

Three types of primitive values - booleans, integers and names - are presumed. There are literals for boolean and integer constants, but none for names. Primitive operations include $\wedge, \vee, \neg$ on booleans, $+, -, *, =$ on integers. Expressions always evaluate to values of one of the primitive types. An expression may contain identifiers which may be bound by formal parameters of the behavior definition in which the expression occurs (see behavior definitions in Section 3.3). Identifiers are lexically scoped in SAL. We let e range over the syntactic domain of expressions, and u, v, w, x, y, z over actor names.

3.2 Commands

Following is the syntax for SAL commands.

Com ::=	**send** $[e_1, ..., e_n]$ **to** x	**(message send)**
	become $B(e_1, ..., e_n)$	**(new behavior)**
	let $x_1 =$ [**recep**] **new** $B_1(e_1, ..., e_{i_1})$,	
	$\ldots\ x_k =$ [**recep**] **new** $B_k(e_1, ..., e_{i_k})$	
	in *Com*	**(actor creations)**
	if e **then** Com_1 **else** Com_2	**(conditional)**
	case x **of** $(y_1 : Com_1, \ldots, y_n : Com_n)$	**(name matching)**
	$Com_1 \mid\mid Com_2$	**(composition)**

Message send: Expressions e_1 to e_n are evaluated, and a message containing the resulting tuple of values is sent to actor x. A message send is asynchronous: execution of the command does not involve actual delivery of the message.

New behavior: This specifies a new behavior for the actor which is executing the command. The identifier B should be bound by a behavior definition (see behavior definitions in Section 3.3). Expressions e_1 to e_n are evaluated and the results are bound to the parameters in the acquaintance list of B. The resulting closure is the new behavior of the actor. A `become` command cannot occur in the top level command of an actor program, because the top level command specifies the initial system configuration and not the behavior of a single actor.

Actor creations: Actors with the specified behaviors are created. The identifiers $x_1, \ldots, x_n$, which are all required to be distinct, denote names of the actors, and the command *Com* is executed under the scope of these identifiers. In the top level command of a program, the identifiers can be optionally tagged with the qualifier `recep`. The corresponding actors will be receptionists of the program configuration, and can thus receive messages from the environment; all the other actors are (at least initially) private to the configuration. The set of receptionists can of course expand during the execution, as messages containing the names of non-receptionists are sent to the environment.

While the scope of the identifiers declared as receptionists is the entire top level command, the scope of the others is only the `let` command. Because actor names are unique, a name can not be declared as a receptionist more than once in the entire top level command. An actor creates new actors with universally fresh names, and these names must be communicated before they can be used by any actor other than the creator. This freshness property would be violated if any of the new actors is declared as a receptionist. Therefore, the `recep` qualifier can not be used in behavior definitions.

Conditional: The expression e should evaluate to a boolean. If the result is true, command Com_1 is executed, else Com_2 is executed.

Name matching: The name x is matched against the names $y_1, \ldots, y_n$. If there is a match, the command corresponding to one of the matches is non-deterministically chosen and executed. If there is no match, then there is no further execution of the command. Note that the mismatch capability on names is not available, i.e. it is not possible to take an action based on the failure of a match.

Composition: The two composed commands are executed concurrently.

A couple of observations are in order here. First, there is no notion of sequential composition of commands. This is because all the actions an actor performs on receiving a message, other than the evaluation order dependencies imposed by the semantics, are concurrent. Second, message passing in SAL is analogous to *call-by-value* parameter passing; expressions in a `send` command are first evaluated and a message is created with the resulting values. Alternately, we can think of a *call-by-need* message passing scheme. But both the mechanisms are semantically equivalent because expressions do not involve recursions and hence their evaluations always terminate.

3.3 Behavior Definitions

The syntax of behavior definitions is as follows.

$BDef$ ::= **def** $\langle beh\ name \rangle (\langle acquaintence\ list \rangle)[\langle input\ list \rangle]$
Com
end def

The identifier $\langle beh\ name \rangle$ is bound to an abstraction and the scope of this binding is the entire program. The identifiers in acquaintance list are formal parameters of this abstraction, and their scope is the body *Com*. These parameters are bound during a behavior instantiation, and the resulting closure is an actor behavior. The identifiers in input list are formal parameters of this behavior, and their scope is the body *Com*. They are bound at the receipt of a message. The acquaintance and input lists contain all the free identifiers in *Com*. The reserved identifier *self* can be used in *Com* as a reference to the actor which has (an instance of) the behavior being defined. The execution of *Com* should always result in the execution of at most a single `become` command, else the behavior definition is said to be erroneous. This property is guaranteed statically by requiring that in any concurrent composition of commands, at most one of the commands contains a `become`. If the execution of *Com* does not result in the execution of a `become`, then the corresponding actor is assumed to take on a '*sink*' behavior that simply ignores all the messages it receives.

3.4 An Example

SAL is not equipped with high-level control flow structures such as recursion and iteration. However, such structures can be encoded as patterns of message passing [18]. The following implementation of the factorial function (adapted from [1]) shows how recursion can be encoded. The example also illustrates *continuation passing style* of programming common in actor systems.

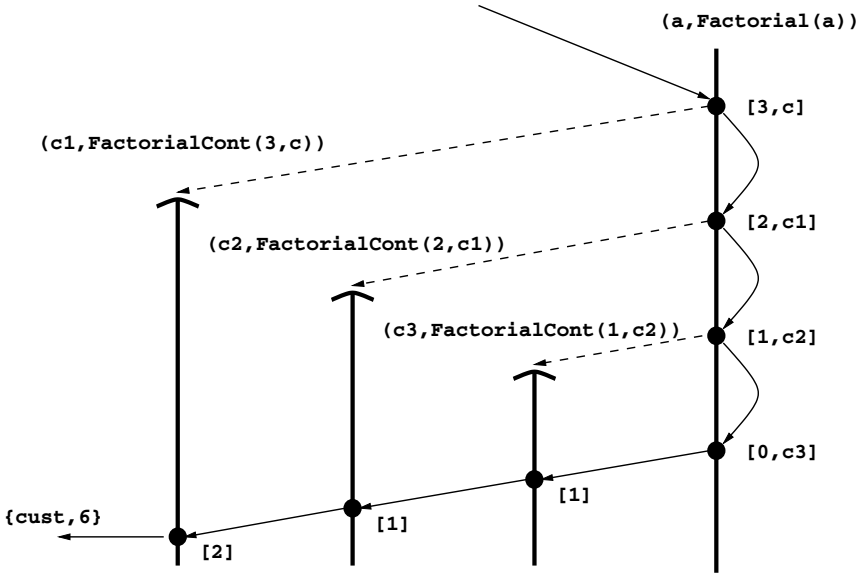

Fig. 2. A diagram illustrating computation of factorial 3, whose result is to be sent back to the actor *c*. The vertical lines denote time lines of actors. An arrow to the top of a time line denotes an actor creation. Other arrows denote messages.

```
def Factorial ()[val,cust]
      become Factorial ()  ||
      if val = 0
        then send [1] to cust
        else let cont = new FactorialCont (val,cust)
                 in send [val − 1, cont] to self
end def

def FactorialCont (val,cust)[arg]
      send [val ∗ arg] to cust
end def
```

A request to factorial actor includes a positive integer n and the actor name *cust* to which the result has to be sent. On receiving a message the actor creates a continuation actor *cont* and sends itself a message with contents $n - 1$ and *cont*. The continuation actor has n and *cust* as its acquaintances. Eventually a chain of continuation actors will be created each knowing the name of the next in the chain (see Figure 2). On receiving a message with an integer, the behavior of each continuation actor is to multiply the integer with the one it remembers and send the reply to its customer. The program can be proved correct by a simple induction on n. Note that since the factorial actor is stateless, it can

process different factorial requests concurrently, without affecting the result of a factorial evaluation.

Following is a top level command that creates a factorial actor that is also a receptionist and sends a message with value 5 to it.

$$\begin{array}{l}\textbf{let } x = [\textbf{recep}]\ \textbf{new } \mathit{Factorial}() \\ \quad \textbf{in send } [5]\ \textbf{to } x\end{array}$$

4 The Calculus Aπ

The Actor Model and π-calculus have served as the basis of a large body of research on concurrency. In this section, we represent the Actor Model as a typed asynchronous π-calculus [7, 21], called Aπ. The type system imposes a certain discipline on the use of names to capture actor properties such as uniqueness, freshness and persistence. This embedding of the Actor Model in π-calculus not only provides a direct basis for comparison between the two models, but also enables us to apply concepts and techniques developed for π-calculus to the Actor Model. As an illustration of how the theory of behavioral equivalences for π-calculus can be adapted to the Actor Model, we develop a theory of *may* testing for Aπ in Section 5. In the interest of space and simplicity, we skip the proofs of all the propositions we state. In fact, the proofs are variations of the ones presented in [53, 54].

4.1 Syntax

We assume an infinite set of names $\mathcal{N}$, and let $u, v, w, x, y, z, \ldots$ range over $\mathcal{N}$. The set of configurations, ranged over by P, Q, R, is defined by the following grammar.

$$\begin{array}{rcl} P := & 0 \mid x(y).P \mid \overline{x}y \mid (\nu x)P \mid P_1|P_2 \\ & \mid \texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, \ldots, y_n : P_n) \mid B\langle \tilde{x}; \tilde{y} \rangle \end{array}$$

The order of precedence among the constructs is the order in which they are listed. The reader may note that, as in the π-calculus, only names are assumed to be primitive in Aπ. As we will see in Section 6, datatypes such as booleans and integers, and operations on them, can be encoded as Aπ processes. These encodings are similar to those for π-calculus [40]; the differences arise mainly due to the typing constraints imposed by Aπ.

Following is the intended interpretation of Aπ terms as actor configurations. The *nil* term 0, represents an empty configuration. The output term $\overline{x}y$, represents a configuration with a single message targeted to x and with contents y. We call x the *subject* of the output term. Note that unlike in SAL, where tuples of arbitrary length can be communicated, only a single name can be communicated per message in Aπ. As we will explain in Section 4.3, *polyadic* communication (communication of tuples of arbitrary length) can be encoded in Aπ, although only after relaxing the persistence property. The input term $x(y).P$ represents

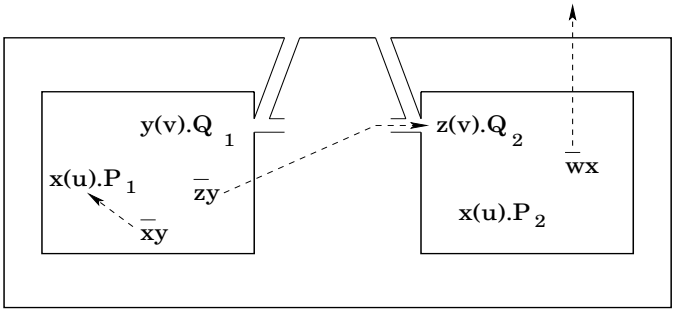

Fig. 3. A visualization of the Aπ term $R = (\nu x)(x(u).P_1|y(v).Q_1|\overline{x}y|\overline{z}y) \mid (\nu x)(x(u).P_2|z(v).Q_2|\overline{w}x)$. A box around subterms indicates a restriction operator. An outlet next to an actor inside the box indicates that the actor is a receptionist for the configuration.

a configuration with an actor x whose behavior is $(y)P$. The parameter y constitutes the formal parameter list of the behavior $(y)P$, and binds all the free occurrences of y in P. The actor x can receive an arbitrary name z and substitute it for y in the definition of P, and then behave like $P\{z/y\}$ (see below for the definition of substitution). We call x the subject of the input term.

The restricted process $(\nu x)P$ is the same as P, except that x is no longer a receptionist of P. All free occurrence of x in P are bound by the restriction. Thus, the receptionists of a configuration P, are simply those actors whose names are not bound by a restriction. The composition $P_1|P_2$ is a configuration containing all the actors and messages in P_1 and P_2. The configuration `case` x `of` $(y_1 : P_1, \ldots, y_n : P_n)$ behaves like P_i if $x = y_i$, and like 0 if $x \neq y_i$ for $1 \leq i \leq n$. If more than one branch is true, one of them is non-deterministically chosen. Note that this construct does not provide mismatch capability on names, i.e. it does not allow us to take an action based on the failure of a match. Thus, this construct is much like the `case` construct of SAL.

The term $B\langle\tilde{u};\tilde{v}\rangle$ is a behavior instantiation. The identifier B has a single defining equation of the form $B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P$, where $\tilde{x}$ is a tuple of distinct names of length 1 or 2, and x_1 denotes the first component of $\tilde{x}$. This definition, like a behavior definition in SAL, provides a template for an actor behavior. The tuples $\tilde{x}$ and $\tilde{y}$ together contain exactly the free names in $x_1(z).P$, and constitute the acquaintance list of the behavior definition. The reason behind the constraint on length of $\tilde{x}$ will be clear in Section 4.2. For an instantiation $B\langle\tilde{u};\tilde{v}\rangle$, we assume $len(\tilde{u}) = len(\tilde{x})$, and $len(\tilde{v}) = len(\tilde{y})$. In the case where $\tilde{v}$ is the empty tuple, we write $B\langle\tilde{u}\rangle$ as a shorthand for $B\langle\tilde{u};\rangle$.

For example, the configuration

$$R = (\nu x)(x(u).P_1|y(v).Q_1|\overline{x}y|\overline{z}y) \mid (\nu x)(x(u).P_2|z(v).Q_2|\overline{w}x)$$

is a composition of two sub-configurations (see Figure 3). The first consists of two actors x and y, a message targeted to x, and a message targeted to an actor

z that is external to the sub-configuration. The actor y is a receptionist, while x is hidden. The second sub-configuration, also contains two actors x and z, and a message targeted to an external actor w. Note that although the name x is used to denote two different actors, the uniqueness property of actor names is not violated in R because the scopes of the two restrictions of x do not intersect. The actors y and z are receptionists of the configuration R.

The reader may note that we use the `case` construct and recursive definitions instead of the standard match ($[x = y]P$) and replication ($!P$) operators of π-calculus. We have chosen these constructs mainly because they are more convenient in expressing actor systems. However, both these constructs can be encoded using the match and replication operators. For instance, the reader can find an encoding of recursive definitions using the standard π-calculus constructs in [40].

Before presenting the type system, a few notational conventions are in order. For a tuple $\tilde{x}$, we denote the set of names occurring in $\tilde{x}$ by $\{\tilde{x}\}$. We denote the result of appending $\tilde{y}$ to $\tilde{x}$ by $\tilde{x}, \tilde{y}$. We assume the variable $\hat{z}$ ranges over $\{\emptyset, \{z\}\}$. By $\tilde{x}, \hat{z}$ we mean $\tilde{x}, z$ if $\hat{z} = \{z\}$, and $\tilde{x}$ otherwise. By $(\nu\hat{z})P$ we mean $(\nu z)P$ if $\hat{z} = \{z\}$, and P otherwise. We define the functions for free names, bound names and names, $fn(.)$, $bn(.)$ and $n(.)$, of a process as expected. As usual, we do not distinguish between alpha-equivalent processes, i.e. between processes that differ only in the use of bound names. A name substitution is a function on names that is almost always the identity. We write $\{\tilde{y}/\tilde{x}\}$ to denote a substitution that maps x_i to y_i and is identity on all other names, and let σ range over substitutions. We denote the result of simultaneous substitution of y_i for x_i in P by $P\{\tilde{y}/\tilde{x}\}$. As usual, we define substitution on processes only modulo alpha-equivalence, with the usual renaming of bound names to avoid captures.

4.2 Type System

Not all terms represent actor configurations. For example, the term $x(u).P|x(v).Q$ violates the uniqueness property of actor names, as it contains two actors with name x. The term $x(u).(u(v).P|x(v).Q)$ violates the freshness property because it creates an actor with name u that is received in a message. Uniqueness of actor names and freshness of names of newly created actors, capture essential aspects of object identity. We enforce such constraints by imposing a type system.

Enforcing all actor properties directly in Aπ results in a language that is too weak to express certain communication patterns. For example, consider expressing polyadic communication in Aπ, where tuples of arbitrary length can be communicated. Since communication in Aπ is monadic, both the sending and receiving actors have to exchange each component of the tuple one at a time, and delay the processing of other messages until all the arguments are transfered. But on the other hand, the persistence property implies that both the actors are always ready to process any message targeted to them. We therefore relax the persistence requirement, so that instead of assuming a new behavior immediately after receiving a message, an actor can wait until certain synchronization

conditions are met before processing the next message. Specifically, we allow an actor to assume a series of fresh names, one at a time, and resume the old name at a later point. Basically, the synchronization task is delegated from one new name to another until the last one releases the actor after the synchronization conditions are met.

We assume $\perp, * \notin \mathcal{N}$, and for $X \subset \mathcal{N}$ define $X^* = X \cup \{\perp, *\}$. For $f : X \to X^*$, we define $f^* : X^* \to X^*$ as $f^*(x) = f(x)$ for $x \in X$ and $f^*(\perp) = f^*(*) = \perp$. A typing judgment is of the form $\rho; f \vdash P$, where ρ is the receptionist set of P, and $f : \rho \to \rho^*$ is a temporary name mapping function that relates actors in P to the temporary names they have currently assumes. Specifically

- $f(x) = \perp$ means that x is a regular actor name and not a temporary one,
- $f(x) = *$ means x is the temporary name of an actor with a private name (bound by a restriction), and
- $f(x) = y \notin \{\perp, *\}$ means that actor y has assumed the temporary name x.

The function f has the following properties. For all $x, y \in \rho$,

- $f(x) \neq x$: This holds for obvious reasons.
- $f(x) = f(y) \notin \{\perp, *\}$ implies $x = y$: This holds because an actor cannot assume more than one temporary name at the same time.
- $f^*(f(x)) = \perp$: This holds because temporary names are not like regular actor names in that they themselves cannot temporarily assume new names, but can only delegate their capability of releasing the original actor to new names.

We define a few functions and relations on the temporary name mapping functions, that will be useful in defining the type rules.

Definition 1. *Let $f_1 : \rho_1 \to \rho_1^*$ and $f_2 : \rho_2 \to \rho_2^*$.*

1. *We define $f_1 \oplus f_2 : \rho_1 \cup \rho_2 \to (\rho_1 \cup \rho_2)^*$ as*

$$(f_1 \oplus f_2)(x) = \begin{cases} f_1(x) \text{ if } x \in \rho_1, \text{ and } f_1(x) \neq \perp \text{ or } x \notin \rho_2 \\ f_2(x) \text{ otherwise} \end{cases}$$

   Note that $\oplus$ is associative.
2. *If $\rho \subset \rho_1$ we define $f|\rho : \rho \to \rho^*$ as*

$$(f|\rho)(x) = \begin{cases} * & \text{if } f(x) \in \rho_1 - \rho \\ f(x) & \text{otherwise} \end{cases}$$

3. *We say f_1 and f_2 are compatible if $f = f_1 \oplus f_2$ has following properties: $f = f_2 \oplus f_1$, and for all $x, y \in \rho_1 \cup \rho_2$, $f(x) \neq x$, $f^*(f(x)) = \perp$, and $f(x) = f(y) \notin \{\perp, *\}$ implies $x = y$.* □

Definition 2. *For a tuple $\tilde{x}$, we define $ch(\tilde{x}) : \{\tilde{x}\} \to \{\tilde{x}\}^*$ as $ch(\epsilon) = \{\}$, and if $len(\tilde{x}) = n$, $ch(\tilde{x})(x_i) = x_{i+1}$ for $1 \leq i < n$ and $ch(\tilde{x})(x_n) = \perp$.* □

$$\textit{NIL: } \emptyset; \{\} \vdash 0 \qquad \textit{MSG: } \emptyset; \{\} \vdash \overline{x}y$$

$$\textit{ACT: } \frac{\rho; f \vdash P}{\{x\} \cup \hat{z}; ch(x, \hat{z}) \vdash x(y).P} \texttt{ if } \begin{array}{l} \rho - \{x\} = \hat{z},\ y \notin \rho, \texttt{ and} \\ f = \begin{cases} ch(x, \hat{z}) \texttt{ if } x \in \rho \\ ch(\epsilon, \hat{z}) \text{ otherwise} \end{cases} \end{array}$$

$$\textit{CASE: } \frac{\forall 1 \leq i \leq n \quad \rho_i; f_i \vdash P_i}{(\cup_i \rho_i); (f_1 \oplus f_2 \oplus \ldots \oplus f_n) \vdash \texttt{case } x \texttt{ of } (y_1 : P_1, \ldots, y_n : P_n)} \texttt{ if } f_i \texttt{ are mutually compatible}$$

$$\textit{COMP: } \frac{\rho_1; f_1 \vdash P_1 \qquad \rho_2; f_2 \vdash P_2}{\rho_1 \cup \rho_2; f_1 \oplus f_2 \vdash P_1 | P_2} \texttt{ if } \rho_1 \cap \rho_2 = \phi$$

$$\textit{RES: } \frac{\rho; f \vdash P}{\rho - \{x\}; f|(\rho - \{x\}) \vdash (\nu x)P}$$

$$\textit{INST: } \{\tilde{x}\}; ch(\tilde{x}) \vdash B\langle \tilde{x}; \tilde{y} \rangle \texttt{ if } len(\tilde{x}) = 2 \texttt{ implies } x_1 \neq x_2$$

Table 1. Type rules for Aπ.

The type rules are shown in Table 1. Rules *NIL* and *MSG* are obvious. In the *ACT* rule, if $\hat{z} = \{z\}$ then actor z has assumed temporary name x. The condition $y \notin \rho$ ensures that actors are not created with names received in a message. This is what is commonly referred to as the *locality* property in the π-calculus literature [35] [1]. The conditions $y \notin \rho$ and $\rho - \{x\} = \hat{z}$ together guarantee the freshness property by ensuring that new actors are created with fresh names. Note that it is possible for x to be a regular name, i.e. $\rho - \{x\} = \emptyset$, and disappear after receiving some message, i.e. $x \notin \rho$. We interpret this as the actor x having assumed a *sink* behavior, i.e. that it simply consumes all the messages that it now receives. With this interpretation the intended persistence property is not violated. Note that a similar interpretation was adopted to account for the case where the body of a SAL behavior definition does not execute a `become` command (see Section 3.3).

The compatibility check in *CASE* rule prevents errors such as: two actors, each in a different branch, assuming the same temporary name; or, the same actor assumes different temporary names in different branches. The *COMP* rule guarantees the uniqueness property by ensuring that the two composed configurations do not contain actors with the same name. In the *RES* rule, f is updated so that if x has assumed a temporary name y in P, then y's role as a temporary name is remembered but x is forgotten. The *INST* rule states that if $len(\tilde{x}) = 2$, then $B\langle \tilde{x}; \tilde{y} \rangle$ denotes a configuration containing a single actor x_2 that has assumed temporary name x_1.

[1] In the context of π-calculus, the locality constraint stipulates that a processes can not receive a name and listen to it; the constraint is enforced by the simple syntactic rule that in a term $x(y).P$, the name y can not occur as the subject of an input.

Type checking a term involves checking the accompanying behavior definitions. For *INST* rule to be sound, for every definition $B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P$ and substitution $\sigma = \{\tilde{u},\tilde{v}/\tilde{x},\tilde{y}\}$ that is one-to-one on $\{\tilde{x}\}$, the judgment $\{\tilde{u}\}; ch(\tilde{u}) \vdash (x_1(z).P)\sigma$ should be derivable. From Lemma 1, it follows that this constraint is satisfied if $\{\tilde{x}\}; ch(\tilde{x}) \vdash x_1(z).P$ is derivable. Thus, a term is well-typed only if for each accompanying behavior definition $B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P$, the judgment $\{\tilde{x}\}; ch(\tilde{x}) \vdash x_1(z).P$ is derivable.

The following theorem states a soundness property of the type system.

Theorem 1. *If $\rho; f \vdash P$ then $\rho \subset fn(P)$, and for all $x, y \in \rho$, $f(x) \neq x$, $f^*(f(x)) = \perp$, and $f(x) = f(y) \notin \{\perp, *\}$ implies $x = y$. Furthermore, if $\rho'; f' \vdash P$ then $\rho = \rho'$ and $f = f'$.* □

Not all substitutions on a term P yield terms. A substitution σ may identify distinct actor names in P, and therefore violate the uniqueness property. But, if σ renames different actors in P to different names, then $P\sigma$ will be well typed. This is formally stated in Lemma 1, where we have used the following notation. For a set of names X, $\sigma(X)$ denotes the set obtained by applying the substitution σ to each element of X. Further, if σ is a substitution which is one-to-one on X, $f\sigma : \sigma(X) \to \sigma(X)^*$ is defined as $f\sigma(\sigma(x)) = \sigma(f(x))$, where $\sigma(\perp) = \perp$ and $\sigma(*) = *$.

Lemma 1. *If $\rho; f \vdash P$ and σ is one-to-one on ρ then $\sigma(\rho); f\sigma \vdash P\sigma$.* □

A consequence of Lemma 1 is that the type system respects alpha-equivalence, i.e. if P_1 and P_2 are alpha-equivalent, then $\rho; f \vdash P_1$ if and only if $\rho; f \vdash P_2$. For a well-typed term P, we define $rcp(P) = \rho$ if $\rho; f \vdash P$ for some f.

4.3 Operational Semantics

We specify the operational semantics of Aπ using a labeled transition system (see Table 2). The rules are obtained by simple modifications to the usual rules for asynchronous π-calculus [7]. The modifications simply account for the use of `case` construct and recursive definitions instead of the standard match and replication operators.

The transition system is defined modulo alpha-equivalence on processes, i.e. alpha-equivalent processes are declared to the same transitions. The symmetric versions of *COM, CLOSE*, and *PAR*, where the roles of P_1 and P_2 are interchanged, are not shown. Transition labels, which are also called actions, can be of five forms: τ (a silent action), $\overline{x}y$ (free output of a message with target x and content y), $\overline{x}(y)$ (bound output), xy (free input of a message), and $x(y)$ (bound input). We denote the set of all visible (non-τ) actions by $\mathcal{L}$, let α range over $\mathcal{L}$, and let β range over all the actions.

The interpretation of these rules in terms of the Actor Model, is as follows. The *INP* rule represents the receipt of a message by an actor, and the *OUT* rule represents the emission of a message. The *BINP* rule is used to infer bound

$$INP\ \ x(y).P \xrightarrow{xz} P\{z/y\} \qquad BINP\ \frac{P \xrightarrow{xy} P'}{P \xrightarrow{x(y)} P'}\ y \notin fn(P)$$

$$OUT\ \ \overline{x}y \xrightarrow{\overline{x}y} 0$$

$$RES\ \frac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'}\ y \notin n(\alpha) \qquad OPEN\ \frac{P \xrightarrow{\overline{x}y} P'}{(\nu y)P \xrightarrow{\overline{x}(y)} P'}\ x \neq y$$

$$PAR\ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1|P_2 \xrightarrow{\alpha} P_1'|P_2}\ bn(\alpha) \cap fn(P_2) = \emptyset \qquad COM\ \frac{P_1 \xrightarrow{\overline{x}y} P_1'\ \ P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'}$$

$$CLOSE\ \frac{P_1 \xrightarrow{\overline{x}(y)} P_1'\ \ P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} (\nu y)(P_1'|P_2')}\ y \notin fn(P_2)$$

$$BRNCH\ \texttt{case}\ x\ \textbf{of}\ (y_1 : P_1, \ldots, y_n : P_n) \xrightarrow{\tau} P_i\ \ \texttt{if}\ x = y_i$$

$$BEHV\ \frac{(x_1(z).P)\{(\tilde{u},\tilde{v})/(\tilde{x},\tilde{y})\} \xrightarrow{\alpha} P'}{B\langle \tilde{u};\tilde{v}\rangle \xrightarrow{\alpha} P'}\ B \stackrel{def}{=} (\tilde{x};\tilde{y})x_1(z).P$$

Table 2. A labeled transition system for Aπ.

inputs, i.e. receipt of messages that contain actor names that were previously unknown to the receiving configuration. The *RES* rule states that an action α performed by P can also be performed by $(\nu x)P$, provided x does not occur in α. This condition disallows the emission of a message which contains the name of a hidden actor in the configuration (a non-receptionist), and prevents confusing a received name with the name of a hidden actor. The *OPEN* rule accounts for the former type of actions, while the latter can be accounted for by an alpha-conversion of the recepient $(\nu x)P$ to a term $(\nu y)P\{y/x\}$, where y does not occur in α, and then applying the *RES* rule. Note that in the *OPEN* rule, the hidden actor name that is being emitted is bound in the output action, but is no longer bound by a restriction in the transition target. Thus, the actor which was hidden in the transition source, becomes a receptionist in the target. The side condition of the *OPEN* rule prevents the emission of messages that are targeted to the hidden actor.

The *PAR* rule captures the concurrent composition of configurations. The side condition of the rule prevents erroneous inferences of bound inputs and outputs. For example, if P_1 performs a bound input $x(y)$, and $y \in fn(P_2)$, then the entire configuration $P_1|P_2$ can *not* perform the bound input $x(y)$ as it already 'knows' the name y. Similarly, it would be erroneous to allow bound outputs of P_1 with the output argument occuring free in P_2; such behavior would confuse the name of a previously hidden actor with the name of another actor.

The *COM* rule is used to infer the communication of a receptionist name between two composed configurations. The *CLOSE* rule is used to infer the communication of non-receptionist names between the configurations. The side condition prevents confusion of the private name that is communicated, with

other names in the recipient P_2. Note that the transition target has a top-level restriction of the communicated name; thus the actor whose name is communicated (internally) is still a non-receptionist in the transition target.

The *BRNCH* and *BEHV* rules are self explanatory. The following theorem states that the type system respects the transition rules.

Theorem 2. *If P is well-typed and $P \xrightarrow{\alpha} P'$ then P' is well-typed.*

Since well-typed terms are closed under transitions, it follows that actor properties are preserved during a computation. However, note that the source and the target of a transition need not have the same typing judgment. Specifically, both the receptionist set and the function that relates actors to the temporary names they have assumed, may change. For instance the receptionist set changes when the name of a hidden actor is emitted to the environment, or an actor disappears after receiving a message. (The reader may recall that the latter case is interpreted as the actor assuming a sink behavior.) Similarly, the temporary name map function changes when an actor with a temporary name re-assumes its original name.

Example 1 (polyadic communication). We show how the ability to temporarily assume a fresh name can be used to encode polyadic communication in Aπ. Suppose that the subject of a polyadic receive is not a temporary name. In particular, in the encoding below, x cannot be a temporary name. The idea behind translation is to let x temporarily assume a fresh name z which is used to receive all the arguments without any interference from other messages, and re-assume x after the receipt. For fresh u, z we have

$$
\begin{aligned}
[\![\overline{x}\langle y_1,\ldots,y_n\rangle]\!] &= (\nu u)(\overline{x}u \mid S_1\langle u; y_1,\ldots,y_n\rangle)\\
S_i &\stackrel{def}{=} (u; y_i,\ldots,y_n)u(z).(\overline{z}y_i \mid S_{i+1}\langle u; y_{i+1},\ldots,y_n\rangle) \quad 1 \le i < n\\
S_n &\stackrel{def}{=} (u; y_n)u(z).\overline{z}y_n\\
[\![x(y_1,\ldots,y_n).P]\!] &= x(u).(\nu z)(\overline{u}z \mid R_1\langle z, \hat{x}; u, \tilde{a}\rangle)\\
R_i &\stackrel{def}{=} (z, \hat{x}; u, \tilde{a})z(y_i).(\overline{u}z \mid R_{i+1}\langle z, \hat{x}; u, \tilde{a}\rangle) \qquad 1 \le i < n\\
R_n &\stackrel{def}{=} (z, \hat{x}; u, \tilde{a})z(y_n).(\overline{u}z \mid [\![P]\!])
\end{aligned}
$$

where $\tilde{a} = fn(x(y_1,\ldots,y_n).P) - \{x\}$, and $\hat{x} = \{x\}$ if for some ρ, f, we have $\rho \cup \{x\}; f \vdash [\![P]\!]$, and $\hat{x} = \emptyset$ otherwise. □

Before we proceed any further, a few definitions and notational conventions are in order. The functions $fn(.)$, $bn(.)$ and $n(.)$ are defined on $\mathcal{L}$ as expected. As a uniform notation for free and bound actions we adopt the following convention from [7]: $(\emptyset)\overline{x}y = \overline{x}y$, $(\{y\})\overline{x}y = \overline{x}(y)$, and similarly for input actions. We define a complementation function on $\mathcal{L}$ as $\overline{(\hat{y})xy} = (\hat{y})\overline{x}y$, $\overline{(\hat{y})\overline{x}y} = (\hat{y})xy$. The variables s, r, t are assumed to range over $\mathcal{L}^*$. The functions $fn(.)$, $bn(.)$, $n(.)$, and complementation on $\mathcal{L}$ are extended to $\mathcal{L}^*$ the obvious way. Elements in $\mathcal{L}^*$ are

called traces. Alpha-equivalence over traces is defined as expected, and alpha-equivalent traces are not distinguished. The relation $\Longrightarrow$ denotes the reflexive transitive closure of $\stackrel{\tau}{\longrightarrow}$, and $\stackrel{\beta}{\Longrightarrow}$ denotes $\Longrightarrow\stackrel{\beta}{\longrightarrow}\Longrightarrow$. For $s = l.s'$, $P \stackrel{l}{\longrightarrow}\stackrel{s'}{\longrightarrow} Q$ is compactly written as $P \stackrel{s}{\longrightarrow} Q$, and similarly $P \stackrel{l}{\Longrightarrow}\stackrel{s'}{\Longrightarrow} Q$ as $P \stackrel{s}{\Longrightarrow} Q$. The assertion, $P \stackrel{s}{\Longrightarrow} P'$ for some P', is written as $P \stackrel{s}{\Longrightarrow}$, and similarly $P \stackrel{s}{\longrightarrow}$ and $P \stackrel{\tau}{\longrightarrow}$.

Not every trace produced by the transition system corresponds to an actor computation. For instance, we have

$$(\nu x)(x(u).P|\overline{x}x|\overline{y}x) \stackrel{\overline{y}(x)}{\longrightarrow} x(u).P|\overline{x}x \stackrel{\overline{x}x}{\longrightarrow}$$

But the message $\overline{x}x$ is not observable; due to the uniqueness property of actor names, there can never be an actor named x in the environment. To account for this, we define for any set of names ρ, the notion of a ρ-well-formed trace such that only ρ-well-formed traces can be exhibited by an actor configuration with ρ as its initial receptionist set.

Definition 3. *For a set of names ρ and trace s we define $rcp(\rho, s)$ inductively as*

$$rcp(\rho, \epsilon) = \rho \qquad rcp(\rho, s.(\hat{y})xy) = rcp(\rho, s) \qquad rcp(\rho, s.(\hat{y})\overline{x}y) = rcp(\rho, s) \cup \hat{y}$$

We say s is ρ-well-formed if $s = s_1.(\hat{y})\overline{x}y.s_2$ implies $x \notin rcp(\rho, s_1)$. We say s is well-formed if it is $\emptyset$-well-formed. □

The following lemma captures our intuition.

Lemma 2. *Let $P|Q$ be a well-typed $A\pi$ term with $rcp(P) = \rho_1$ and $rcp(Q) = \rho_2$. Then $P|Q \Longrightarrow$ can be unzipped into $P \stackrel{s}{\Longrightarrow}$ and $Q \stackrel{\overline{s}}{\Longrightarrow}$ such that s is ρ_1-well-formed and $\overline{s}$ is ρ_2-well-formed.* □

For convenience, since we work only modulo alpha-equivalence on traces, we adopt the following hygiene condition. Whenever we are interested in ρ-well-formed traces, we will only consider traces s such that if $s = s_1.\alpha.s_2$, then $(\rho \cup n(s_1) \cup fn(\alpha)) \cap bn(\alpha.s_2) = \emptyset$.

The transition sequences are further constrained by a fairness requirement which requires messages to be eventually delivered, if they can be. For example, the following transition sequences are unfair.

$$\begin{aligned} Diverge\langle x\rangle|\overline{x}u|y(v).\overline{v}v|\overline{y}v &\stackrel{\tau}{\longrightarrow} Diverge\langle x\rangle|\overline{x}u|y(v).\overline{v}v|\overline{y}v \\ &\stackrel{\tau}{\longrightarrow} Diverge\langle x\rangle|\overline{x}u|y(v).\overline{v}v|\overline{y}v \\ &\stackrel{\tau}{\longrightarrow} \cdots \end{aligned}$$

$$\text{where } Diverge \stackrel{def}{=} (x)x(u).(\overline{x}u \mid Diverge\langle x\rangle)$$

In every transition above, the message $\overline{x}y$ is delivered to its target; but the message $\overline{y}v$ is never delivered.

Fairness in actors requires that the delivery of a message is not delayed infinitely long; but it can be delayed for any finite number of steps. Thus, only infinite transition sequences can be unfair. However, note that our fairness constraint does *not* require that *every* message is eventually delivered to its target. Because we have relaxed the persistence property, an actor may disappear during a computation, after which all the message targeted to it become permanently disabled. Thus, the fairness criteria only requires that there is no message that is infinitely often enabled, but not delivered. This is consistent with our convention that an actor that disappears is assumed to take on a *sink* behavior.

The fairness requirement can be enforced by defining a predicate on sequences of transitions as described in [48] such that only fair sequences satisfy the predicate. However, we do not pursue this any further in this paper, as fairness does not effect the theory we are concerned with. The reader is referred to Section 5.4 for further discussion.

4.4 Discussion

There has been considerable research on actor semantics in the past two decades. We set Aπ in the context of some of the salient work. A significant fraction of the research has been in formal semantics for high level concurrent programming languages based on the Actor Model, e.g. [3, 13] where a core functional language is extended with actor coordination primitives. The main aim of these works has been to design concurrent languages that could be useful in practice. Accordingly, the languages assume high-level computational notions as primitives, and are embellished with type systems that guarantee useful properties in object-based settings. In contrast, Aπ is a basic calculus that makes only the ontological commitments inherent in the Actor Model, thus giving us a simpler framework for further theoretical investigations. In Section 6, we show how Aπ can be used to give a translational semantics for SAL.

In [48, 49], actors are modeled in rewriting logic which is often considered as a universal model of concurrency [33, 36]. An actor system is modeled as a specific rewrite theory, and established techniques are used to derive the semantics of the specification and prove its properties. In a larger context, this effort belongs to a collection of works that have demonstrated that rewriting logic provides a good basis to unify many different concurrency theories. For example, we have also a rewrite theory formulation of the π-calculus [52]. In comparison, Aπ establishes a connection between two models of concurrency that is deeper than is immediately available from representing the two models in a unified basis. Specifically, the theory that we have developed in Section 5, can be seen as a more elaborate investigation of the relationship between two specific rewrite theories, and provides a formal connection that helps in adapting and transferring results in one theory to the other.

There are several calculi that are inspired by the Actor Model and the π-calculus [15, 22, 45]. But these are neither entirely faithful to the Actor Model, nor directly comparable to the π-calculus. For example, they are either equipped with primitives intrinsic to neither of the models [15, 22], or they ignore actor

properties such as uniqueness and persistence [45]. These works are primarily intended for investigation of object-oriented concepts.

5 A Theory of May Testing for Aπ

Central to any process calculus is the notion of behavioral equivalence which is concerned with the question of when two processes are equal. Typically, a notion of success is defined, and two processes are considered equivalent if they have the same success properties in all contexts. Depending on the chosen notion of context and success one gets a variety of equivalences [8, 12, 46].

The *may testing* equivalence is one such instance [17, 12], where the context consists of an observing process that runs in parallel and interacts with the process being tested, and success is defined as the observer signaling a special event. The possible non-determinism in execution leads to at least two possibilities for the definition of equivalence. In may testing, a process is said to pass a test proposed by an observer, if there exists *at least one* run that leads to a success. By viewing a success as something bad happening, may testing can be used for reasoning about *safety* properties. An alternate definition, where a process is said to pass a test if *every* run leads to a success, is called the *must testing* equivalence. By viewing a success as something good happening, must testing can be used for reasoning about *liveness* properties. In this paper, we will be develop only with the theory may testing for Aπ.

Context-based behavioral equalities like may testing suffer from the need for universal quantification over all possible contexts; such quantification makes it very hard to prove equalities directly from the definition. One solution is to find an alternate characterization of the equivalence which involves only the processes being compared. We provide an alternate characterization of may testing in Aπ that is *trace* based and directly builds on the known characterization for asynchronous π-calculus.

5.1 A Generalized May Preorder

As in any typed calculus, may testing in Aπ takes typing into account; an observer O can be used to test a configuration P only if $P|O$ is well-typed. Note that $P|O$ is well-typed only if $rcp(P) \cap rcp(O) = \emptyset$. Thus, O can be used to test the equivalence between P and Q only if $rcp(O) \cap (rcp(P) \cup rcp(Q)) = \emptyset$.

The uniqueness property of actor names naturally leads to a generalized version of may testing, where the equivalence $\simeq_\rho$ is tagged with a parameter ρ. All possible observers O that do not listen on names in ρ, i.e. $rcp(O) \cap \rho = \emptyset$, are used for deciding $\simeq_\rho$. Of course, for processes P and Q to be compared with $\simeq_\rho$, it has to be the case that $rcp(P), rcp(Q) \subset \rho$.

Definition 4 (may testing). *Observers are processes that can emit a special message $\overline{\mu}\mu$. We let O range over the set of observers. We say O accepts a trace s if $O \stackrel{\overline{s}.\overline{\mu}\mu}{\Longrightarrow}$. For P, O, we say P may O if $P|O \stackrel{\overline{\mu}\mu}{\Longrightarrow}$. For ρ such that*

(L1)	$s_1.(\hat{y})s_2$	$\prec$	$s_1.(\hat{y})xy.s_2$	if $(\hat{y})s_2 \neq \perp$
(L2)	$s_1.(\hat{y})(\alpha.xy.s_2)$	$\prec$	$s_1.(\hat{y})xy.\alpha.s_2$	if $(\hat{y})(\alpha.xy.s_2) \neq \perp$
(L3)	$s_1.(\hat{y})s_2$	$\prec$	$s_1.(\hat{y})xy.\overline{x}y.s_2$	if $(\hat{y})s_2 \neq \perp$
(L4)	$s_1.\overline{x}w.(s_2\{w/y\})$	$\prec$	$s_1.\overline{x}(y).s_2$	

Table 3. A preorder relation on traces.

$rcp(P), rcp(Q) \subset \rho$, we say $P \stackrel{\sqsubseteq}{\sim}_\rho Q$, if for every O such that $rcp(O) \cap \rho = \emptyset$, P may O implies Q may O. We say $P \simeq_\rho Q$ if $P \stackrel{\sqsubseteq}{\sim}_\rho Q$ and $Q \stackrel{\sqsubseteq}{\sim}_\rho P$. □

The relation $\stackrel{\sqsubseteq}{\sim}_\rho$ is a preorder, i.e. reflexive and transitive, and $\simeq_\rho$ is an equivalence relation. Further, note that the larger the parameter ρ, the smaller the observer set that is used to decide $\stackrel{\sqsubseteq}{\sim}_\rho$. Hence if $\rho_1 \subset \rho_2$, we have $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$ implies $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$. However, $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$ need not imply $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$. For instance, $0 \simeq_{\{x\}} \overline{x}x$, but only $0 \stackrel{\sqsubseteq}{\sim}_\emptyset \overline{x}x$ and $\overline{x}x \not\stackrel{\sqsubseteq}{\sim}_\emptyset 0$. Similarly, $\overline{x}x \simeq_{\{x,y\}} \overline{y}y$, but $\overline{x}x \not\stackrel{\sqsubseteq}{\sim}_\emptyset \overline{y}y$ and $\overline{y}y \not\stackrel{\sqsubseteq}{\sim}_\emptyset \overline{x}x$. However, $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$ implies $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$ if $fn(P) \cup fn(Q) \subset \rho_1$.

Theorem 3. *Let $\rho_1 \subset \rho_2$. Then $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$ implies $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$. Furthermore, if $fn(P) \cup fn(Q) \subset \rho_1$ then $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$ implies $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$.* □

5.2 An Alternate Characterization of May Testing

We now build on the trace-based characterization of may testing for asynchronous π-calculus presented in [7] to obtain a characterization of may testing in Aπ. Following is a summary of the alternate characterization of may testing in asynchronous π-calculus. To account for asynchrony, the trace semantics is modified using a trace preorder $\preceq$ that is defined as the reflexive transitive closure of the laws shown in Table 3, where the notation $(\hat{y})\cdot$ is extended to traces as follows.

$$(\hat{y})s = \begin{cases} s & \text{if } \hat{y} = \emptyset \text{ or } y \notin fn(s) \\ s_1.x(y).s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ s.t.} \\ & \quad s = s_1.xy.s_2 \text{ and } y \notin fn(s_1) \cup \{x\} \\ \perp & \text{otherwise} \end{cases}$$

The expression $(\hat{y})s$ returns $\perp$, if $\hat{y} = \{y\}$ and y is used in s before it is received for the first time, i.e. the first free occurrence of y in s is *not* as the argument of an input. Otherwise, the expression returns the trace s with the first such free input changed to a bound input. The (unparameterized) may preorder $\stackrel{\sqsubseteq}{\sim}$ in asynchronous π-calculus (which corresponds to $\stackrel{\sqsubseteq}{\sim}_\emptyset$ in our setting) is then characterized as: $P \stackrel{\sqsubseteq}{\sim} Q$ if and only if $P \stackrel{s}{\Longrightarrow}$ implies $Q \stackrel{r}{\Longrightarrow}$ for some $r \preceq s$.

The intuition behind the preorder is that if an observer accepts a trace s, then it also accepts any trace $r \preceq s$. Laws *L1-L3* capture asynchrony, and *L4*

captures the inability to mismatch names. Laws *L1* and *L2* state that an observer cannot force inputs on the process being tested. Since outputs are asynchronous, the actions following an output in a trace exhibited by an observer need not be causally dependent on the output. Hence the observer's outputs can be delayed until a causally dependent action (*L2*), or dropped if there are no such actions (*L1*). Law *L3* states that an observer can consume its own outputs unless there are subsequent actions that depend on the output. Law *L4* states that without mismatch an observer cannot discriminate bound names from free names, and hence can receive any name in place of a bound name. The intuition behind the trace preorder is formalized in the following lemma that is proved in [7] for asynchronous π-calculus.

Lemma 3. *If* $P \stackrel{\overline{s}}{\Longrightarrow}$, *then* $r \preceq s$ *implies* $P \stackrel{\overline{r}}{\Longrightarrow}$. □

We note that, the lemma above also holds for Aπ with very simple modifications to the proof.

Actor properties such as uniqueness and freshness "weaken" may equivalence in Aπ, in comparison to asynchronous π-calculus. Specifically, the type system of Aπ reduces the number of observers that can be used to test actor configurations. For example, the following two processes are distinguishable in asynchronous π-calculus, but equivalent in Aπ:

$$P = (\nu x)(x(z).0|\overline{x}x|\overline{y}x) \qquad Q = (\nu x)(x(z).0|\overline{y}x)$$

The observer $O = y(z).z(w).\overline{\mu}\mu$ can distinguish P and Q in asynchronous π-calculus, but is not a valid Aπ term as it violates the freshness property (*ACT* rule of Table 1). In fact, no Aπ term can distinguish P and Q, because the message $\overline{x}x$ is not observable.

The following alternate preorder on configurations characterizes the may preorder in Aπ.

Definition 5. *We say* $P \ll_\rho Q$, *if for every* ρ*-well-formed trace* s, $P \stackrel{s}{\Longrightarrow}$ *implies there is* $r \preceq s$ *such that* $Q \stackrel{r}{\Longrightarrow}$. □

To prove the characterization, we define an observer $O(s)$ for a well-formed trace s, such that P *may* $O(s)$ implies $P \stackrel{r}{\Longrightarrow}$ for some $r \preceq s$.

Definition 6 (canonical observer). *For a well-formed trace* s, *we define an observer*

$$O(s) = (\nu \tilde{x}, z)(|_{y_i \in \chi} Proxy(s, y_i, z) \mid O'(s, z)), \text{ where } z \text{ fresh}$$

$\{\tilde{x}\}$ = *set of names occurring as argument of bound input actions in* s
χ = *set of names occuring as subject of output actions in* s

$O'(\epsilon, z) \stackrel{\triangle}{=} \overline{\mu}\mu$
$O'((\hat{v})uv.s, z) \stackrel{\triangle}{=} \overline{u}v|O'(s, z)$

$O'(\overline{u}v.s, z) \triangleq z(w_1, w_2).\texttt{case}\ w_1\ \texttt{of}\ (u : \texttt{case}\ w_2\ \texttt{of}\ (v : O'(s, z)))\quad w_1, w_2\ \textit{fresh}$
$O'(\overline{u}(v).s, z) \triangleq z(w, v).\texttt{case}\ w\ \texttt{of}\ (u : O'(s, z))\quad w\ \textit{fresh}$

$Proxy(\epsilon, y, z) \triangleq 0$
$Proxy((\hat{v})uv.s, y, z) \triangleq Proxy(s, y, z)$
$Proxy((\hat{v})\overline{u}v.s, y, z) \triangleq \begin{cases} y(w).(\overline{z}\langle y, w\rangle \mid Proxy(s, y, z)) & w\ \textit{fresh}\quad \textit{if } u = y \\ Proxy(s, y, z) & \textit{otherwise} \end{cases}$

In the above, $\triangleq$ *is used for macro definitions. The reader may verify that* $\chi - \{\tilde{x}\}; f \vdash O(s)$ *where* f *maps every name in its domain to* $\perp$*. Further, if* s *is* ρ*-well-formed we have* $rcp(O(s)) \cap \rho = \emptyset$*, because the set of names occurring as subject of output actions in a* ρ*-well-formed trace is disjoint from* ρ*.* □

The observer $O(s)$ consists of a collection of proxies and a central matcher. There is one forwarding proxy for each external name that a configuration sends a message to while exhibiting s. The proxies forward messages to the matcher which analyzes the contents. This forwarding mechanism (which is not necessary for the construction of canonical observers in the corresponding proof for asynchronous π-calculus), is essential for Aπ because of uniqueness of actor names. Further, note that the forwarding mechanism uses polyadic communication, whose encoding was shown in Section 4.3. The following lemma formalizes our intention behind the construction of $O(s)$.

Lemma 4. *For a well-formed trace* s*,* $O(s) \stackrel{\overline{r}.\overline{\mu}\mu}{\Longrightarrow}$ *implies* $r \preceq s$*.* □

The following theorem, which establishes the alternate characterization of may preorder in Aπ, can be proved easily using Lemmas 2, 3, and 4.

Theorem 4. $P \stackrel{\sqsubseteq}{\sim}_\rho Q$ *if and only if* $P \ll_\rho Q$*.* □

5.3 Some Axioms for May Testing

Table 4 lists some inference rules besides the reflexivity and transitivity rules, and some axioms for $\stackrel{\sqsubseteq}{\sim}_\rho$. For an index set $I = \{1, \ldots, n\}$, we use the macro $\sum_{i\in I} P_i$ to denote, $(\nu u)(\texttt{case}\ u\ \texttt{of}\ (u : P_1, \ldots, u : P_n))$ for u fresh if $I \neq \emptyset$, and 0 otherwise. For an index set that is a singleton, we omit I and simply write $\sum P$ instead of $\sum_{i\in I} P$. We let the variable G range over processes of form $\sum_{i\in I} P_i$. We write $\sum_{i\in I} P_i + \sum_{j\in J} P_j$ to denote $\sum_{k\in I\uplus J} P_k$. We write $\sqsubseteq$ as a shorthand for $\sqsubseteq_\emptyset$, and $=$ for $=_\emptyset$.

Axioms *A1* to *A17* are self explanatory. We note that they also hold in asynchronous π-calculus [7]. But axiom *A18* is unique to Aπ. It captures the fact that a message targeted to an internal actor in a configuration, cannot escape to the environment. The axiom states that there are only two ways such a message can be handled in the next transition step: it can be consumed internally or delayed for later. The axiom also allows for dropping of the message permanently, which is useful when the message target no longer exists (it may have disappeared

$I1$ if $P \sqsubseteq_\rho Q$ and $rcp(R) \cap \rho = \emptyset$, then $(\nu x)P \sqsubseteq_{\rho-\{x\}} (\nu x)Q$, $P|R \sqsubseteq_\rho Q|R$.
$I2$ if for each $z \in fn(P,Q)$ $P\{z/y\} \sqsubseteq_\rho Q\{z/y\}$ then $x(y).P \sqsubseteq_\rho x(y).Q$
$I3$ if for each $i \in I$ $P_i \sqsubseteq_\rho \sum_{j\in J} Q_{ij}$ then $\sum_{i\in I} P_i \sqsubseteq_\rho \sum_{i\in I, j\in J} Q_{ij}$
$I4$ if $\rho_1 \subset \rho_2$ and $P \sqsubseteq_{\rho_1} Q$ then $P \sqsubseteq_{\rho_2} Q$.

$A1$ $G + G = G$ $A3$ $P|0 = P$ $A5$ $(P|Q)|R = P|(Q|R)$
$A2$ $G \sqsubseteq G + G'$ $A4$ $P|Q = Q|P$

$A6$ $(\nu x)(\sum_{i\in I} P_i) = \sum_{i\in I}(\nu x)P_i$
$A7$ $(\nu x)(P|Q) = P|(\nu x)Q$ $x \notin n(P)$
$A8$ $(\nu x)(\overline{x}y|\alpha.P) = \alpha.(\nu x)(\overline{x}y|P)$ $x \notin n(\alpha)$
$A9$ $(\nu x)(\overline{x}y|x(z).P) = (\nu x)(P\{y/z\})$
$A10$ $(\nu x)(y(z).P) = \begin{cases} y(z).(\nu x)P & \text{if } x \neq y, x \neq z \\ 0 & \text{if } x = y \end{cases}$

$A11$ $\overline{x}y|\sum_{i\in I} P_i = \sum_{i\in I}(\overline{x}y|P_i)$ $I \neq \emptyset$
$A12$ $\alpha.\sum_{i\in I} P_i = \sum_{i\in I} \alpha.P_i$ $I \neq \emptyset$
$A13$ $P = \sum P$

$A14$ $x(y).(\overline{u}v|P) \sqsubseteq \overline{u}v|x(y).P$ $y \neq u, y \neq v$
$A15$ $P\{y/z\} \sqsubseteq \overline{x}y|x(z).P$
$A16$ $x(y).(\overline{x}y|P) \sqsubseteq P$ $y \notin n(P)$

$A17$ $(\nu x)P \sqsubseteq P\{y/x\}$

$A18$ If $x \in \rho$, $w \neq x$ and $w \neq y$, then

$$\overline{x}y|z(w).P \sqsubseteq_\rho \sum z(w).(\overline{x}y|P) + \sum z(w).P + \sum Q$$

$$\text{where } Q = \begin{cases} P\{y/w\} & \text{if } x = z \\ 0 & \text{otherwise} \end{cases}$$

Table 4. Inference rules and axioms for $\overset{\sqsubseteq}{\sim}_\rho$ in Aπ.

during the computation). As an application of this axiom, if $x \in \rho$, we can prove $\overline{x}y \sqsubseteq_\rho 0$ as follows. For w fresh,

$$\begin{array}{lll}
\overline{x}y \sqsubseteq_\rho & \overline{x}y|(\nu w)(w(w).0) & (A3, A10, I1) \\
\sqsubseteq_\rho & (\nu w)(\overline{x}y|w(w).0) & (A7) \\
\sqsubseteq_\rho & (\nu w)(\sum w(w).0 + \sum w(w).\overline{x}y + \sum 0) & (A18, I1) \\
\sqsubseteq_\rho & \sum(\nu w)(w(w).0) + \sum(\nu w)w(w).\overline{x}y + \sum(\nu w)0 & (A6) \\
\sqsubseteq_\rho & 0 & (A1, A10, A13, I3)
\end{array}$$

Inference rules $I1$ and $I3$ are self explanatory, while $I4$ is motivated by Theorem 3. We illustrate $I1$ through some examples. First, using $\overline{x}y \sqsubseteq_{\{x\}} 0$ (proved above) and $I1$, we get $(\nu x)\overline{x}y \sqsubseteq (\nu x)0$, and by axiom $A17$ we have $(\nu x)0 \sqsubseteq 0$.

Therefore, $(\nu x)\overline{x}y \sqsubseteq 0$. Note the use of the ability to contract the parameter ρ of the may preorder after applying a restriction. Second, the following example illustrates the necessity of the side condition $rcp(R) \cap \rho = \emptyset$ for composition: $\overline{x}y \overset{\sqsubseteq}{\sim}_{\{x\}} 0$ but not $\overline{x}y|x(y).\overline{y}y \overset{\sqsubseteq}{\sim}_{\{x\}} x(y).\overline{y}y$, for the LHS can satisfy the observer $y(u).\overline{\mu}\mu$ and the RHS can not.

Note that the inference rules are generalizations of rules for asynchronous π-calculus presented in [7], in order to handle parameterization of the may preorder. In fact, the rules for asynchronous π-calculus can be obtained by setting $\rho = \emptyset$ in *I1, I2* and *I3*. Rule *I4* is unique to the parameterized may preorder.

The soundness of rules *I1-I4* can be easily proved directly from Definition 4. Soundness of the axioms is easy to check. For *A1-A17*, whenever $P \sqsubseteq Q$, we have $P \overset{s}{\Longrightarrow}$, implies $Q \overset{r}{\Longrightarrow}$ such that $r \preceq s$. For *A18* , both LHS and RHS exhibit the same ρ-well-formed traces. The reader can verify that *A18* would also be sound as an equality.

5.4 Discussion

The alternate characterization of may testing for Aπ turns out to be the same as that for L$\pi_=$ which we presented in [54]. L$\pi_=$ is a version of asynchronous π-calculus with match operator and the locality constraint (see Section 4.2 for a discussion on locality). This shows that of all the constraints enforced by the type system of Aπ, only locality and uniqueness (which is taken care of by parameterization of the may preorder) has an effect on may testing.

In Section, 4.3, we claimed that the fairness property of the Actor Model does not affect the theory we have presented. The justification is simple. May testing is concerned only with the occurrence of an event after a finite computation, while fairness affects only infinite computations. An interesting consequence of fairness, however, is that must equivalence [17] implies may equivalence, which was shown for a specific actor-based language in [3]. It can be shown by a similar argument that this result holds in Aπ also.

There has been a significant amount of research on notions of equivalence and semantic models for actors, including asynchronous bisimulation [15], testing equivalences [3], event diagrams [10], and interaction paths [50]. We have not only related may testing [3] to the interaction paths model [50], but also related our characterizations to that of asynchronous π-calculus and its variants.

6 Formal Semantics of SAL

Aπ can serve as the basis for actor based concurrent programming languages. As an illustration, we give a formal semantics for SAL by translating its programs into Aπ. The translation can be exploited to apply the characterizations established in Section 5 to reason about programs in SAL.

In Sections 6.1 and 6.2, we show how booleans, natural numbers and operations on them can be represented as processes in Aπ. These data types, along with names, are assumed as primitive in SAL. Of course, this exercise is not

entirely necessary, and in fact, a better strategy may be to directly consider an extended version of Aπ with basic data types. The characterizations for Aπ can be adapted in a straightforward manner to the extended calculus. We have chosen the other approach here, mainly to illustrate that the type system of Aπ does not reduce the expressive power of the calculus. In Sections 6.3-6.5, we present the translation of SAL expressions, commands and behavior definitions. SAL expressions and commands are translated into Aπ terms, and their evaluation is modeled as computation in these terms. SAL behavior definitions are translated into recursive definitions in Aπ.

6.1 Booleans

Booleans are encoded as configurations with a single actor that is also a receptionist. In the following, $\underline{T}$ defines the receptionist behavior for *true*, and $\underline{F}$ for *false*.

$$\underline{T} \stackrel{def}{=} (x)x(u,v,c).\overline{c}u$$

$$\underline{F} \stackrel{def}{=} (x)x(u,v,c).\overline{c}v$$

The behaviors accept messages containing three names, of which the last name is assumed to be the customer name (see Section 4.3 for an encoding of polyadic communication). The behavior $\underline{T}$ replies back to the customer with the first name, while $\underline{F}$ replies back with the second name.

The negation function can be encoded as follows

$$Not \stackrel{def}{=} (x)x(u,c).(\nu v,y,z)(\overline{u}\langle y,z,v\rangle \mid v(w).\texttt{case } w \texttt{ of}(y:\underline{F}\langle v\rangle, z:\underline{T}\langle v\rangle))$$

$Not(x)$ can be thought of as the function *not* available at name x. Evaluation of the function is initiated by sending a message containing a value and a customer name, to x. The customer eventually receives the negation of the value sent. The reader may verify that

$$Not\langle x\rangle \mid \underline{F}\langle u\rangle \mid \overline{x}\langle u,c\rangle \stackrel{\overline{c}(v)}{\Longrightarrow} \underline{T}\langle v\rangle$$

Following is the encoding of boolean *and*

$$\begin{aligned} And \stackrel{def}{=} (x)x(u,v,c).(\nu y,z_1,z_2)(&\overline{u}\langle z_1,z_2,y\rangle \mid \overline{v}\langle z_1,z_2,y\rangle \mid \\ &y(w_1).y(w_2).(\overline{c}y \mid \\ &\quad \texttt{case } w_1 \texttt{ of } (\\ &\quad\quad z_1 : \texttt{case } w_2 \texttt{ of } (z_1:\underline{T}\langle y\rangle, z_2:\underline{F}\langle y\rangle), \\ &\quad\quad z_2 : \underline{F}\langle y\rangle\ ))) \end{aligned}$$

The reader may verify the following

$$And\langle x\rangle \mid \underline{T}\langle u\rangle \mid \underline{F}\langle v\rangle \mid \overline{x}\langle u,v,c\rangle \stackrel{\overline{c}(y)}{\Longrightarrow} \underline{F}\langle y\rangle$$

The reader may also verify that for each behavior B defined above $\{x\};\{x \mapsto \perp\} \vdash B\langle x\rangle$.

6.2 Natural Numbers

Natural numbers are built from the constructors 0 and S. Accordingly, we define the following two behaviors.

$$Zero \stackrel{def}{=} (x)x(u,v,c).\overline{c}\langle u,x\rangle$$
$$Succ \stackrel{def}{=} (x,y)x(u,v,c).\overline{c}\langle v,y\rangle$$

With these, natural numbers can be encoded as follows.

$$\underline{0}(x) \stackrel{\triangle}{=} Zero\langle x\rangle$$
$$\underline{S^{n+1}0}(x) \stackrel{\triangle}{=} (\nu y)(Succ\langle x,y\rangle \mid \underline{S^n 0}(y))$$

The number $S^n 0$ is encoded as a sequence of $n+1$ actors each pointing to the next, and the last one pointing to itself. The first n actors have the behavior *Succ* and the last has behavior *Zero*. Only the first actor is the receptionist to the entire configuration. As in our encoding for booleans, both the behaviors accept messages with three names, the last of which is assumed to denote the customer. The behavior *Succ* replies back to the customer with the second name and the name of next actor in the sequence, while *Zero* replies back with the first name and its own name.

We only show the encoding of the addition operation, and hope the reader is convinced that it is possible to encode the others. Our aim is to define a behavior *Add* such that

$$Add\langle x\rangle \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \overline{x}\langle u,v,c\rangle \stackrel{\overline{c}(w)}{\Longrightarrow} \underline{S^{n+m}0}(w)$$

We first define a behavior *AddTo* such that

$$AddTo\langle x\rangle \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u,v,c\rangle) \mid \underline{S^m 0}(v) \Longrightarrow \quad (\nu u)(\underline{S^{n+m}0}(u) \mid \overline{c}u)$$

We will then use *AddTo* to define *Add*.

$$\begin{aligned} AddTo \stackrel{def}{=} (x)x(u_1,u_2,c).(\nu y_1,y_2,w)(\overline{u_2}\langle y_1,y_2,w\rangle \mid \\ w(z_1,z_2).\texttt{case } z_1 \texttt{ of } (\\ y_1 : \overline{c}u_1, \\ y_2 : (\nu v)(Succ\langle v,u_1\rangle \mid \overline{x}\langle v,z_2,c\rangle \mid AddTo\langle x\rangle))) \end{aligned}$$

We are now ready to define *Add*.

$$\begin{aligned} Add \stackrel{def}{=} (x)x(u,v,c).(\nu y,z,w)(AddTo\langle y\rangle \mid \underline{0}(w) \mid \overline{y}\langle w,u,z\rangle \mid \\ z(w).(\nu y)(AddTo\langle y\rangle \mid \overline{y}\langle w,v,c\rangle)) \end{aligned}$$

Lemma 5. $Add\langle x\rangle \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \overline{x}\langle u,v,c\rangle \stackrel{\overline{c}(w)}{\Longrightarrow} \underline{S^{n+m}0}(w)$

The reader may verify that for a natural number N, and each behavior B defined above, $\{x\};\{x \mapsto \bot\} \vdash \underline{N}(x)$, and $\{x\};\{x \mapsto \bot\} \vdash B\langle x\rangle$. This encoding of natural numbers can be extended to integers in a fairly straightforward manner (for example, by using tags to indicate the sign).

6.3 Expressions

Now that we have a representation of the basic constituents of expressions - namely, booleans, integers, and names - what remains is the representation of dependencies between the evaluation of subexpressions of an expression.

The translation of an expression takes as an argument, the name of a customer to which the result of the expression's evaluation is to be sent. An identifier expression x is translated as

$$[\![x]\!]c = \overline{c}x$$

A constant (boolean or integer) expression e is translated as

$$[\![e]\!]c = (\nu y)(\underline{e}\langle y\rangle \mid \overline{c}y)$$

where $\underline{e}$ is the encoding of the constant e. For an $n-$ary operator Op, the expression $Op(e_1, ..., e_n)$ is encoded as

$$[\![Op(e_1,\ldots,e_n)]\!]c = (\nu y_1,\ldots,y_{n+1},z)(Marshal(y_1,\ldots,y_{n+1},z) \mid [\![e_1]\!]y_1 \mid \ldots \mid [\![e_n]\!]y_n \mid \overline{y_{n+1}}c \mid \underline{Op}\langle z\rangle)$$

where $\underline{Op}$ is the encoding of operator Op, and z, y_i are fresh. The expressions e_1 to e_n are concurrently evaluated. The configuration $Marshal(y_1, \ldots, y_{n+1}, z)$ marshals their results and the customer name into a single tuple, and forwards it to an internal actor that implements Op. The marshaling configuration is defined as

$$Marshal(y_1,\ldots,y_n,c) = (\nu u)(R\langle y_1,u\rangle \mid \ldots \mid R\langle y_n,u\rangle \mid S_1\langle u,y_1,\ldots,y_n,c\rangle)$$

where

$$R \stackrel{def}{=} (x,y)\ x(u).\overline{y}\langle u,x\rangle$$

$$S_i \stackrel{def}{=} (x,y_i,\ldots,y_n,v_1,\ldots,v_{i-1},c)\ x(v_i,w).\texttt{case}\ w\ \texttt{of}\ (y_i : S_{i+1}\langle x,y_{i+1},\ldots,y_n,v_1,\ldots,v_i,c\rangle \quad y_{i+1}\ldots y_n : S_i\langle x,y_i,\ldots,y_n,v_1,\ldots,v_{i-1},c\rangle \mid \overline{x}\langle v_i,w\rangle) \quad \text{for } 1 \le i < n$$

$$S_n \stackrel{def}{=} (x,v_1,\ldots,v_{n-1},c)\ x(v_n,w).\overline{c}\langle v_1,\ldots,v_n\rangle$$

By structural induction on an expression e and name x, it is easy to show that $\emptyset; \{\} \vdash [\![e]\!]x$.

6.4 Commands

Although the Actor Model stipulates that the actions an actor performs on receiving a message are all concurrent, execution of SAL commands may involve sequentiality. For example, expressions need to be evaluated before the results are used to send messages or instantiate a behavior. This sequentiality is represented as communication patterns in the Aπ configurations that encode these commands. The translation of a command takes as an argument, the name of the SAL actor which executes the command. In the following, we assume that the names introduced during the translation are all fresh.

Message send: We use the *Marshal* configuration to marshal the results of expression evaluations into a polyadic message to the target.

$$[\![\textbf{send } [e_1,\ldots,e_n] \textbf{ to } z]\!]x = (\nu y_1,\ldots,y_n)(Marshal(y_1,\ldots,y_n,z) \mid [\![e_1]\!]y_1 \mid \ldots \mid [\![e_n]\!]y_n)$$

New behavior: We use an actor's ability to temporarily assume a new name to wait for the results of expression evaluations before assuming the new behavior.

$$[\![\textbf{become } B(e_1,\ldots,e_n)]\!]x = (\nu y_1,\ldots,y_n,z)([\![e_1]\!]y_1 \mid \ldots \mid [\![e_n]\!]y_n \mid Marshal\langle y_1,\ldots,y_n,z\rangle \mid z(u_1,\ldots,u_n).\underline{B}\langle x,u_1,\ldots,u_n\rangle)$$

where $\underline{B}$ is the Aπ behavior definition that is the translation of the SAL behavior definition B (see Section 6.5).

Actor creation: The identifiers in the `let` command are used as names for the new actors. If not tagged by the `recep` qualifier, these names are bound by a restriction. The actors are created at the beginning of command execution, but they assume a temporary name until their behavior is determined.

$$[\![\textbf{let } y_1 = \textbf{[recep] new } B_1(e_1,\ldots,e_{i_1}), \ldots y_k = \textbf{[recep] new } B_k(e_1,\ldots,e_{i_k}) \textbf{ in } Com]\!]x = (\nu\tilde{y})([\![\textbf{become } B_1(e_1,\ldots,e_{i_1})]\!]y_1 \mid \ldots \mid [\![\textbf{become } B_k(e_1,\ldots,e_{i_k})]\!]y_k \mid [\![Com]\!]x)$$

where $\tilde{y}$ consists of all y_i which have not been qualified with `recep`.

Conditional: We use a temporary actor that waits for the outcome of the test before executing the appropriate command.

$$[\![\textbf{if } e \textbf{ then } Com_1 \textbf{ else } Com_2]\!]x = (\nu u)([\![e]\!]u \mid u(z).(\nu v_1,v_2)(\overline{z}\langle v_1,v_2,u\rangle \mid u(w).\texttt{case } w \texttt{ of } (v_1 : [\![Com]\!]x, v_2 : [\![Com]\!]x))$$

Name matching: The translation simply uses the `case` construct of Aπ.

$$[\![\textbf{case } z \textbf{ of}(y_1 : Com_1,\ldots,y_n : Com_n)]\!]x = \texttt{case } z \texttt{ of } (y_1 : [\![Com_1]\!]x,\ldots,y_n : [\![Com_n]\!]x)$$

Concurrent Composition: The translation of a concurrent composition is just the composition of individual translations.

$$[\![Com_1 \mid\mid Com_2]\!]x = [\![Com_1]\!]x \mid [\![Com_2]\!]x$$

This completes the translation of commands. Let *Com* be a command such that in any of its subcommands that is a concurrent composition, at most one of the composed commands contains a `become`. Further, assume that a name is declared as a receptionist at most once, and that `let` constructs with receptionist

declarations are not nested under other `let` or conditional constructs. Let x be a fresh name. Then by structural induction on *Com*, we can show that $\{x, \tilde{y}\}; f \vdash [\![Com]\!]x$ if *Com* contains a `become`, and $\{\tilde{y}\}; f \vdash [\![Com]\!]x$ otherwise, where $\tilde{y}$ is the set of all names declared as receptionists in *Com*, and f is a function that maps all names in its domain to $\perp$.

6.5 Behavior Definitions

Behavior definitions in SAL are translated to behavior definitions in Aπ as follows

$$[\![\textbf{def } B(\tilde{u})[\tilde{v}] \quad Com \quad \textbf{end def}]\!] \;=\; \underline{B} \stackrel{def}{=} (self; \tilde{u})self(\tilde{v}).[\![Com]\!]self$$

Note that the implicitly available reference *self* in a SAL behavior definition becomes explicit in the acquaintance list after translation. Since the body of a behavior definition does not contain receptionist declarations, it follows that $\{self\}; \{self \mapsto \perp\} \vdash self(\tilde{v}).[\![Com]\!]self$. So the RHS is well-typed.

We have completed the translation of various syntactic domains in SAL, and are ready to present the overall translation of a SAL program. Recall that a SAL program consists of a sequence of behavior definitions and a single top level command. Following is the translation.

$$[\![BDef_1 \;\; ... \;\; BDef_n \;\; Com]\!] = [\![BDef_1]\!] \;\; \ldots \;\; [\![BDef_n]\!] \;\; [\![Com]\!]x$$

where x is fresh. Since the top level command cannot contain a `become`, its translation does not use the argument x supplied. Indeed, $\{\tilde{y}\}; f \vdash [\![Com]\!]x$, where $\{\tilde{y}\}$ is the set of all names declared as receptionists in *Com*, and f maps all names in $\{\tilde{y}\}$ to $\perp$.

6.6 Discussion

The translation we have given, can in principle be exploited to use the testing theory developed for Aπ, to reason about SAL programs. Note that the characterization of may-testing for Aπ applies unchanged to SAL. This is because the set of experiments possible in SAL have the same distinguishing power as the experiments in Aπ. Specifically, the canonical observers constructed for Aπ in Section 5, are also expressible in SAL. Further, it follows immediately from Lemma 4 and Theorem 4 that these observers have all the distinguishing power, i.e. are sufficient to decide $\stackrel{\sqsubset}{\sim}_\rho$ in Aπ.

Although, SAL is a very simple language, it can be enriched with higher level programming constructs without altering the characterization. This is corroborated by the work in [34], where a high level actor language is translated to a more basic kernel language (similar to SAL) in such a way that the source and its translation exhibit the same set of traces.

Translational semantics for actor languages similar to SAL has been previously attempted. In [11] a simple actor language is translated into linear logic

formulae, and computations are modeled as deductions in the logic. In [29] an actor-based object-oriented language is translated into HACL extended with records [28]. These translations provide a firm foundation for further semantic investigations. However, to reap the benefits of these translations, one still has to explicitly characterize actor properties such as locality and uniqueness in the underlying formal system, and identify the changes to the theory due to them. For instance, the asynchronous π-calculus can be seen as the underlying system of our translation, whereas only Aπ terms correspond to SAL programs, and the characterization of may testing for Aπ is very different from that for asynchronous π-calculus.

7 Research Directions

In this paper, we have focused only on the semantic aspects of the Actor Model, while much of the actor research over the last two decades has been on languages and systems.

Actor programming has been effective in combining benefits of object style encapsulation with the concurrency of real-world systems. The autonomy of actors frees a programmer from the burden of explicitly managing threads and synchronizing them. In fact, the autonomy of actors also facilitates mobility. Over the years, many implementations of the Actor Model have been done [9, 55, 59]. In fact, the numerous agent languages currently being developed typically follow the Actor Model [32, 23]. The Actor Model has also been used for efficient parallel computation [24, 26, 27, 51]. Actor languages and systems currently being developed include SALSA for web computing [56], Ptolemy II for embedded systems [30], and ActorFoundry for distributed computing [43].

An important area of active research in the Actor Model is the use of *computational reflection* [31]. The execution environment of an actor application can be represented as a collection of actors called *meta-actors*. These meta-actors constitute the middleware which mediates the interaction of the actor application and the underlying operating systems and networks. In order to customize fault-tolerance, security, synchronization and other types of interaction properties, meta-actors may be customized (see, for example, [5, 47]). Moreover, the meta-actor model supports the ability to express dynamic coordination policies as executable synchronization constraints between actors: the constraints may be enforced by customizing the meta-actors during execution [14]. An operational semantics of such reflective systems is developed in [57] and a rewriting model has been proposed in [37].

The use of meta-actors supports a separation of design concerns which is now popularly known as *aspect-oriented programming* [25]. The development of aspect-oriented programming will enable the reuse of interaction protocols and functional behavior of an object. The separation of the concurrent interaction protocols from the sequential behavior of actors is another step in the revolution in programming that was instigated by Dahl and Nygaard when they developed the idea of separating the interface of an object from its representation.

8 Acknowledgments

The research described in here has been supported in part by DARPA under contracts F33615-01-C-1907 and F30602-00-2-0586, and by ONR under contract N00014-02-1-0715. Part of the work described in here is a sequel to the research done in collaboration with Reza Ziaei, whom we would like to thank. We also thank Carolyn Talcott for useful discussions and comments.

References

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

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

[3] G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 1996.

[4] G. Agha, P. Wegner, and A. Yonezawa (editors). Proceedings of the ACM SIGPLAN workshop on object-based concurrent programming. Special issue of SIGPLAN Notices.

[5] M. Astley, D. Sturman, and G. Agha. Customizable middleware for modular distributed software. *CACM*, 44(5):99–107, 2001.

[6] G.M. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Van Nostrand Reinhold, New York, 1973.

[7] M. Boreale, R. de Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Information and Computation*, 172(2):139–164, 2002.

[8] M. Boreale and D. Sangiorgi. Some congruence properties for π-calculus bisimilarities. In *Theoretical Computer Science 198*, 1998.

[9] J. P. Briot. Acttalk: A framework for object-oriented concurrent programming - design and experience. In *Object-based parallel and distributed computing II - Proceedings of the* 2^{nd} *France-Japan workshop*, 1999.

[10] W.D. Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institute of Technology, AI Laboratory, 1981.

[11] J. Darlington and Y. K. Guo. Formalizing actors in linear logic. In *International Conference on Object-Oriented Information Systems*, pages 37–53. Springer-Verlag, 1994.

[12] R. de Nicola and M. Hennesy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[13] F.Dagnat, M.Pantel, M.Colin, and P.Sall. Typing concurrent objects and actors. In *L'Objet – Mthodes formelles pour les objets (L'OBJET)*, volume 6, pages 83–106, 2000.

[14] S. Frolund. *Coordinating Distributed Objects: An Actor-Based Approach for Synchronization*. MIT Press, November 1996.

[15] M. Gaspari and G. Zavattaro. An Algebra of Actors. In *Formal Methods for Open Object Based Systems*, 1999.

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

[17] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[18] C. Hewitt. Viewing Control Structures as Patterns of Message Passing. *Journal of Artificial Intelligence*, 8(3):323–364, September 1977.

[19] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In *International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.
[20] C.A.R. Hoare. *Communication Sequential Processes*. Prentice Hall, 1985.
[21] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Fifth European Conference on Object-Oriented Programming*, July 1991. LNCS 512, 1991.
[22] J-L.Colao, M.Pantel, and P.Sall. Analyse de linarit par typage dans un calcul d'acteurs primitifs. In *Actes des Journes Francophones des Langages Applicatifs (JFLA)*, 1997.
[23] N. Jamali, P. Thati, and G. Agha. An actor based architecture for customizing and controlling agent ensembles. *IEEE Intelligent Systems*, 14(2), 1999.
[24] L.V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1993.
[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer Verlag, 1997. LNCS 1241.
[26] W. Kim. *ThAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana Champaign, 1997.
[27] W. Kim and G. Agha. Efficient support of location transparency in concurrent object-oriented programming languages. In *Proceedings of SuperComputing*, 1995.
[28] N. Kobayashi and A. Yonezawa. Higher-order concurrent linear logic programming. In *Theory and Practice of Parallel Programming*, pages 137–166, 1994.
[29] N. Kobayashi and A. Yonezawa. Towards foundations of concurrent object-oriented programming – types and language design. *Theory and Practice of Object Systems*, 1(4), 1995.
[30] E. Lee, S. Neuendorffer, and M. Wirthlin. Actor-oriented design of embedded hardware and software systems. In *Journal of circuits, systems, and computers*, 2002.
[31] P. Maes. *Computational Reflection*. PhD thesis, Vrije University, Brussels, Belgium, 1987. Technical Report 87-2.
[32] P. Maes. Intelligent software: Easing the burdens that computers put on people. In *IEEE Expert, special issue on intelligent agents*, 1996.
[33] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework, 1993.
[34] I.A. Mason and C.Talcott. A semantically sound actor translation. In *ICALP 97*, pages 369–378, 1997. LNCS 1256.
[35] M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In *Proceeding of ICALP '98*. Springer-Verlag, 1998. LNCS 1443.
[36] J. Meseguer. Rewriting Logic as a Unified Model of Concurrency. Technical Report SRI-CSI-90-02, SRI International, Computer Science Laboratory, February 1990.
[37] J. Meseguer and C. Talcott. Semantic models for distributed object reflection. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 1–36, 2002.
[38] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
[39] R. Milner. Interactions, turing award lecture. *Communications of the ACM*, 36(1):79–97, January 1993.

[40] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.
[41] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.
[42] S. Miriyala, G. Agha, and Y.Sami. Visulatizing actor programs using predicate transition nets. *Journal of Visual Programming*, 1992.
[43] Open Systems Laboratory. The Actor Foundry: A Java based actor programming language. Available for download at http://www-osl.cs.uiuc.edu/foundry.
[44] J.L. Peterson. Petri nets. *Comput. Survey*, Sept. 1977.
[45] A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR*, pages 474–488, 2000. LNCS 1877.
[46] R.Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings of* 19^{th} *International Colloquium on Automata, Languages and Programming (ICALP '92)*. Springer Verlag, 1992. LNCS 623.
[47] D. Sturman and G. Agha. A protocol description language for cutomizing semantics. In *Proceedings of symposium on reliable distributed systems*, pages 148–157, 1994.
[48] C. Talcott. An Actor Rewriting Theory. In *Electronic Notes in Theoretical Computer Science 5*, 1996.
[49] C. Talcott. Interaction Semantics for Components of Distributed Systems. In E.Najm and J.B. Stefani, editors, *Formal Methods for Open Object Based Distributed Systems*. Chapman & Hall, 1996.
[50] C. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3), 1998.
[51] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Symposium on principles and practice of parallel programming (PPOPP)*, pages 218–228, 1993.
[52] P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in maude 2.0. In 4^{th} *International Workshop on Rewriting Logic and its Applications*, September 2002.
[53] P. Thati, R. Ziaei, and G. Agha. A theory of may testing for actors. In *Formal Methods for Open Object-based Distributed Systems*, March 2002.
[54] P. Thati, R. Ziaei, and G. Agha. A theory of may testing for asynchronous calculi with locality and no name matching. In *Proceedings of the* 9^{th} *International Conference on Algebraic Methodology and Software Technology*. Springer Verlag, September 2002. LNCS 2422.
[55] C. Tomlinson, W. Kim, M. Schevel, V. Singh, B. Will, and G. Agha. Rosette: An object-oriented concurrent system architecture. *Sigplan Notices*, 24(4):91–93, 1989.
[56] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Notices*, 36(12):20–34, 2001.
[57] N. Venkatasubramanian, C. Talcott, and G. Agha. A formal model for reasoning about adaptive QoS-enabled middleware. In *Formal Methods Europe (FME)*, 2001.
[58] P. Wegner. Dimensions of object-based language design. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 168–182, 1987.
[59] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.