Mobility and Security in Worldwide Computing

Robin D. Toll
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180-3590, U.S.A.
tollr@cs.rpi.edu

Carlos Varela
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180-3590, U.S.A.
cvarela@cs.rpi.edu

ABSTRACT
Modern distributed computing requires a secure framework capable of free code mobility. In this paper, we present a simple lambda-based actor language with extensions for mobility and security, as well as the operational semantics to reason about these topics in distributed systems. Finally, we describe our preliminary implementation results.

1. INTRODUCTION
Internet based distributed computing systems benefit from open dynamically reconfigurable designs as these designs allow the systems to be used in constantly changing heterogeneous environments. Consider a data mining application running on an open distributed system using hundreds of thousands of computing devices to discover useful patterns in scientific data sets (e.g., protein folding, SETI, weather forecasting, etc.). If a system can make use of new computing resources as they become available and is both secure and resilient to failures in a subset of its computing nodes, it has the potential to leverage the power of idle computing resources around the world.

In this paper, we study program component mobility and security as fundamental stepping stones towards robust distributed computing systems. Mobility and security introduce new requirements on software, e.g., it is critical to devise strategies for secure and controlled distributed resource management. We specify an actor-based model and formalism to reason about program component mobility and secure resource access in worldwide computing systems.

Paper Outline
In Section 2, we further motivate and informally introduce abstractions for programming worldwide computing applications. Section 3 specifies our model by providing an operational semantics for a simple actor language and extensions for mobility and security. Lastly, section 4 talks about other systems and future work.

2. PROGRAMMING ABSTRACTIONS FOR WORLDWIDE COMPUTING

2.1 Actors
The Actor model of computation is based around the concept of encapsulating state and process into a single entity. Actors are therefore inherently independent, concurrent and autonomous which enables efficiency in parallel execution [KA95] and facilitates mobility [AJ99]. Each actor is a unit of computation encapsulating data and behavior. The behavior defines how the actor reacts on receipt of a message. Each actor has a unique name, which can be used as a reference by other actors.

Actors only process information in reaction to messages. While processing a message, an actor can carry out any of three basic operations: alter its state, create new actors, or send messages to peer actors (see Figure 1).

Communication between actors is purely asynchronous and guaranteed. That is, when a message is sent, the model guarantees that the destination actor will receive the message; however, it does *not* guarantee the order of message arrival or, therefore, the order of processing. A side effect of this is that since actors can change their own behavior based on incoming messages, unless the actor's name is known only to the sender, its behavior could change significantly before a message arrives and is processed.

The actor model and languages provide a very useful framework for understanding and developing open distributed systems. For example, among other applications, actor systems have been used for enterprise integration [TCMW93], real-time programming [RAS96], fault-tolerance [AFPS93], and distributed artificial intelligence [FB88].

2.2 Universal Actors
In considering mobile computation, it becomes useful to not only model the interactions of actors with each other, but also to model the interactions of actors with their environments. In the actor model, locations are not represented, therefore, it does not matter if two actors are in the same memory space, or on two computers on opposite ends of the earth. However, when considering the problems associated with worldwide computing, it becomes important to represent the actor's environment. Otherwise it is not possible to model the behavior of an actor, e.g., when its computation environment is unreliable, or when different resources are available in different locations.

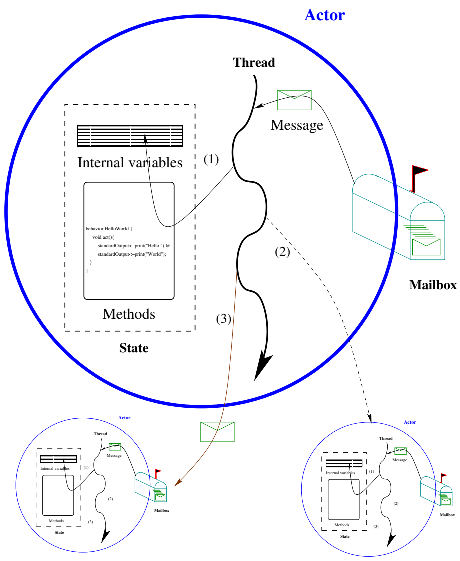

Figure 1: Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) create new actors, and/or (3) send messages to peer actors

Universal actors extend actors with locations, mobility, and the concept of universal names and universal locators. Names represent actor references that do not change with actor migration. Locators represent references that enable communication with universal actors at a specific location.

An actor's location abstracts over its position relative to other actors. Each location represents an actor's run-time environment and serves as an encapsulation unit for local resources. Ubiquitous resources have a generic representation –actors keep references which get updated upon migration to resources at new locations. Actors can also keep references to non-ubiquitous resources –scarce or not generally available– by using resource attachment and detachment operations. For example, a standard output stream is ubiquitous and can always refer to the current actor's execution environment. Conversely, an actor needs to attach to a robot resource in Mars, so that the reference remains the same upon migration.

2.3 Secure Actors

Mobile code can pose a serious danger to any environment it executes in and, conversely, any environment can prove dangerous to actors executing inside. Therefore, it is important to consider the security of host resources and actors. Secure actors restrict communication and migration behaviors to actors within specific access control lists.

The access control list for an actor or resource contains every actor allowed to send messages to it. The access control list for a location contains every actor allowed to migrate into the location. These lists can only be altered by the resource or actor in consideration, or by a resident actor in case of passive locations. Using this method, no unprivileged actor can gain access to a resource, since any unauthorized communication or migration request is rejected.

3. PROGRAMMING LANGUAGES AND SEMANTICS

3.1 A Simple Actor Language and Its Operational Semantics

Agha, Mason, Smith, and Talcott introduce a simple actor language as an extension to the call-by-value lambda calculus, with primitives for actor communication [AMST97].

The actor language, named here *AL*, formally defines three primitives:

- *new(b)*, which creates an actor which has behavior b and returns the new actor's name.
- $send(v_0, v_1)$, which sends a message with contents v_1 to actor v_0.
- *ready(b')*, which signals the end of the current execution and makes the actor ready to receive a new message using behavior b'.

3.2 Actor Configurations

They assume as given two sets At(Atoms) and X(Variables), and then define the set of *values*, V, *expressions*, E, and *messages*, M, as:

$V = At \cup X \cup \lambda X.E \cup pr(V, V)$

$E = V \cup app(E, E) \cup F_n(E^n)$

$M = <V \Leftarrow V>$

where $F_n(E^n)$ is all arity-n primitives.

Variables are used for actor names. At any given point, an actor can either be ready to receive a message (denoted ready(e), where e is a lambda abstraction); or currently executing some expression e. A message sent to actor v_0 with contents v_1 is written as $<v_0 \Leftarrow v_1>$.

An actor configuration is a global snapshot of a group of actors. It includes the concept of an *actor mapping*, where each actor name is mapped to a behavior; a *message set* of messages in transit; a set of *receptionists* (internal actors known to the outside world); and a set of *external actors* (known actors not in the configuration).

An *actor configuration* with actor map, α, multi-set of messages, μ, receptionists, ρ, and external actors, χ, is written[1]

$$\langle \alpha \mid \mu \rangle^{\rho}_{\chi}$$

where $\rho, \chi \in \mathbf{P}_\omega[X]$, $\alpha \in X \xrightarrow{f} E$, $\mu \in \mathbf{M}_\omega[M]$, and let $A = \text{Dom}(\alpha)$, then:

[1]Let $\mathbf{P}_\omega[X]$ be the set of finite subsets (Power Set) of X, $\mathbf{M}_\omega[M]$ be the set of (finite) multi-sets with elements in M, $X_0 \xrightarrow{f} X_1$ be the set of finite maps from $X_0 \xrightarrow{f} X_1$, Dom(f) be the domain of f and FV(e) be the set of free variables in e.

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

(1) if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$, and if $\texttt{<}v_0 \Leftarrow v_1\texttt{>} \in \mu$ then $\mathrm{FV}(v_i) \subseteq A \cup \chi$ for $i < 2$.

3.3 Operational Semantics

We define a transition relation between actor configurations as the least relation satisying the rules in Figure 2[2]. To describe the internal transitions between configurations other than message receipt, an expression is decomposed into a reduction context filled with a redex. The notation $R[e]$ represents a redex e in a reduction context R, as described by Honsell et al.[HMST95] and used by Agha et al[AMST97]. For a formal definition of reduction contexts, expressions with a unique hole; and for the definition of functional progress within an actor ($\overset{\lambda}{\mapsto}_A$), we refer the reader to [AMST97]. The actor redexes are: $\texttt{newactor}(e)$, $\texttt{send}(v_0, v_1)$, and $\texttt{ready}(v)$.

$\texttt{<fun} : a\texttt{>}$

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \langle\, \alpha\{\texttt{[}e\texttt{]}_a\} \mid \mu \,\rangle^\rho_\chi \mapsto \langle\, \alpha\{\texttt{[}e'\texttt{]}_a\} \mid \mu \,\rangle^\rho_\chi$$

$\texttt{<new} : a, a'\texttt{>}$

$$\langle\, \alpha\{\texttt{[}R[new(e)]\texttt{]}_a\} \mid \mu \,\rangle^\rho_\chi \mapsto \langle\, \alpha\{\texttt{[}R[a']\texttt{]}_a, \texttt{[}e\texttt{]}_{a'}\} \mid \mu \,\rangle^\rho_\chi \qquad a' \text{ fresh}$$

$\texttt{<send} : a, v_0, v_1\texttt{>}$

$$\langle\, \alpha\{\texttt{[}R[send(v_0, v_1)]\texttt{]}_a\} \mid \mu \,\rangle^\rho_\chi \mapsto \langle\, \alpha\{\texttt{[}R[nil]\texttt{]}_a\} \mid \mu \uplus \texttt{<}v_0 \Leftarrow v_1\texttt{>} \,\rangle^\rho_\chi$$

$\texttt{<receive} : v_0, v_1\texttt{>}$

$$\langle\, \alpha\{\texttt{[}ready(v)\texttt{]}_a\} \mid \texttt{<}a \Leftarrow v_0\texttt{>} \uplus \mu \,\rangle^\rho_\chi \mapsto \langle\, \alpha\{\texttt{[}app(v, v_0)\texttt{]}_a\} \mid \mu \,\rangle^\rho_\chi$$

$\texttt{<out} : v_0, v_1\texttt{>}$

$$\langle\, \alpha \mid \mu \uplus \texttt{<}a \Leftarrow v_0\texttt{>} \,\rangle^\rho_\chi \mapsto \langle\, \alpha \mid \mu \,\rangle^{\rho'}_\chi$$
$$\text{if } a \in \chi \text{ and } \rho' = \rho \cup (\mathrm{FV}(v_0) \cap \mathrm{Dom}(\alpha))$$

$\texttt{<in} : v_0, v_1\texttt{>}$

$$\langle\, \alpha \mid \mu \,\rangle^\rho_\chi \mapsto \langle\, \alpha \mid \mu \uplus \texttt{<}a \Leftarrow v_0\texttt{>} \,\rangle^\rho_{\chi\cup(\mathrm{FV}(v_0)-\mathrm{Dom}(\alpha))}$$
$$\text{if } a \in \rho \text{ then } \mathrm{FV}(v_0) \cap \mathrm{Dom}(\alpha) \subseteq \rho$$

Figure 2: Actor Language (AL) Semantics

3.4 Mobile Actor Language and Its Operational Semantics

3.4.1 Resources

When we introduce the concept of locations, we also introduce the need to model resources in those locations. One example is the standard output stream in a run-time environment. While the concept remains the same, the actual implementation may change when the actor migrates across different locations.

Each resource is referred to by a universally understood resource name. This resource name is used to contact the implementing resource actor without necessarily ever knowing the specific name of the service providing that resource, or the implementation of that resource. These resources may be ubiquitous, such as a standard output stream. The name 'standard output' may apply to output on a console, or a text field on a graphical user interface, a log file, or even a printer interface, but almost any executing program has access to a primary output stream. If an actor migrates, the primary output stream changes but the transition is transparent to the actor. Referring to some standard resource name allows the environment to handle requests properly.

3.4.2 Resource Maps

Resource-to-actor translations are stored in resource maps. These functions are maps between global names and the names of local actors who fill a resource's roll. When an actor at a location sends a message, resource maps are seached for the target's name; first the actor's resource map followed by the location's resource map. If the target is found, the message is diverted to the resolved actor- otherwise conventional message sending proceeds.

Remote access to resources is permitted because both actors and locations have independent resource maps. Consider the case when a actor migrates but needs to retain a reference to a resource at the original location, such as a output device; the actor's map will direct messages to the original location, where the resource's name is resolved to the implementing actor.

3.5 Mobile Actor Language

To reflect mobility in the universal actor model, we add four primitives to *AL*, forming the Mobile Actor Language (*MAL*). *MAL* thus formally defines seven primitives:

- *new(b)*, which creates an actor which has behavior b and returns the new actor's name.
- *send*(v_0, v_1), which sends a message with contents v_1 to actor v_0.
- *ready(b')*, which signals the end of the current execution and makes the actor ready to receive a new message using behavior b'.
- *newloc*(Y), which indicates the appearance or creation of a new location, with an initial resource map denoted by Y.
- *migrate*(*l'*), which moves an actor from its current location to one denoted by *l'*.
- *attach*(v), which saves a resource denoted by v into the actor's resource map.
- *detach*(v), which removes a resource denoted by v from the actor's resource map.
- *register*(v_0, v_1, l), which adds the mapping of a resource name to an actor in a location.

[2]We define $\alpha\{\texttt{[}e\texttt{]}_a\}$ as an extended mapping which maps a into e, and all other actor names a' into $\alpha(a')$, and $\alpha\{\texttt{[}e\texttt{]}_a, \texttt{[}e'\texttt{]}_{a'}\}$ as $\alpha\{\{\texttt{[}e\texttt{]}_a\}\{\texttt{[}e'\texttt{]}_{a'}\}\}$ for $a \neq a'$.

- *deregister*(v_0, v_1, l), which removes the mapping of a resource name to an actor in a location.

We assume as given two sets At(Atoms) and X(Variables), and we then define the set of *values*, V, *expressions*, E, and *messages*, M, as:

$$V = At \cup X \cup \lambda X.E \cup pr(V, V)$$

$$E = V \cup app(E, E) \cup F_n(E^n)$$

$$M = \texttt{<}V \Leftarrow V\texttt{>}$$

where $F_n(E^n)$ is all arity-n primitives.

The set X of variables represents both actors –which includes all resources– and locations. We introduce a set L, of locations, with $L \subseteq X$. We also introduce a set R, of resource identifiers, with $R \subseteq X$. We modify the definition of α so that $\alpha \in X \xrightarrow{f} (E \times L \times (R \xrightarrow{f} X))$.[3] We extend the definition of an actor configuration to include a mapping, π, from locations to resource maps, with $\pi \in L \xrightarrow{f} (R \xrightarrow{f} X)$. A universal actor configuration is written

$$\langle\, \alpha \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

We add another rule to the definition of actor configurations to denote that actor names and locations are disjoint[4]:

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

(1) if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$, and if $\texttt{<}v_0 \Leftarrow v_1\texttt{>} \in \mu$ then $\mathrm{FV}(v_i) \subseteq A \cup \chi$ for $i < 2$.

(2) $Range(\alpha) \downarrow_2 \cap A = \emptyset$

3.6 Operational Semantics

We define a transition relation between universal actor configurations as the least relation satisying the rules in Figures 3 and 4.

3.7 Secure Mobile Actor Language

We extend the mobile actor language to form the Secure Mobile Actor Language (*SMAL*), which defines eleven primitives:

- *new(b)*, which creates an actor which has behavior b and returns the new actor's name.
- *send*(v_0, v_1), which sends a message with contents v_1 to actor v_0.
- *ready(b')*, which signals the end of the current execution and makes the actor ready to receive a new message using behavior b'.

[3] $\alpha(a) = e \times l \times r$ implies that actor a has behavior e and is currently executing at location l with resource map r.

[4] $\downarrow_i$ indicates the projection of a cross product onto its ith coordinate.

$\texttt{<fun} : a\texttt{>}$

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \langle\, \alpha\{\texttt{[}e, l, r\texttt{]}_a\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\{\texttt{[}e', l, r\texttt{]}_a\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

$\texttt{<new} : a, a'\texttt{>}$

$$\langle\, \alpha\{\texttt{[}R[new(e)], l, r\texttt{]}_a\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\{\texttt{[}R[a'], l, r\texttt{]}_a, \texttt{[}e, l, \emptyset\texttt{]}_{a'}\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

a' fresh

$\texttt{<send} : a, v_0, v_1\texttt{>}$

$$\langle\, \alpha\{\texttt{[}R[send(v_0, v_1)], l, r\texttt{]}_a\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\, \alpha\{\texttt{[}R[nil], l, r\texttt{]}_a\} \mid \mu \uplus \texttt{<}v_2 \Leftarrow v_1\texttt{>} \mid \pi \,\rangle^{\rho}_{\chi}$$

if $r(v_0)=v_2$

$$\langle\, \alpha\{\texttt{[}R[nil], l, r\texttt{]}_a\} \mid \mu \uplus \texttt{<}v_3 \Leftarrow v_1\texttt{>} \mid \pi \,\rangle^{\rho}_{\chi}$$

if $\pi(l)(v_0)=v_3$

$$\langle\, \alpha\{\texttt{[}R[nil], l, r\texttt{]}_a\} \mid \mu \uplus \texttt{<}v_0 \Leftarrow v_1\texttt{>} \mid \pi \,\rangle^{\rho}_{\chi}$$

if $v_0 \notin \mathrm{Dom}(r)$ & $v_0 \notin \mathrm{Dom}(\pi)$

$\texttt{<receive} : v_0, v_1\texttt{>}$

$$\langle\, \alpha\{\texttt{[}R[ready(v)], l, r\texttt{]}_{v_0}\} \mid \texttt{<}v_0 \Leftarrow v_1\texttt{>} \uplus \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\{\texttt{[}R[app(v, v_1)], l, r\texttt{]}_{v_0}\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

Figure 3: MAL Semantics, Part I

- *newloc*(Y), which indicates the appearance or creation of a new location, with an initial resource map denoted by Y.
- *migrate*(l), which moves an actor from its current location to one denoted by l.
- *attach*(v), which saves a resource denoted by v into the actor's resource map.
- *detach*(v), which removes a resource denoted by v from the actor's resource map.
- *register*(v_0, v_1, l), which adds the mapping of a resource name to an actor in a location.
- *deregister*(v_0, v_1, l), which removes the mapping of a resource name to an actor in a location.
- *allow(v)* changes the actor's access list to include actor v.
- *allowloc(v)* changes the actor location's access list to include actor v.
- *disallow(v)* changes the actor's access list to exclude actor v.
- *disallowloc(v)* changes the actor location's access list to exclude actor v.

<out : v_0, v_1>

$$\langle\, \alpha \mid \mu \uplus \texttt{<}a \Leftarrow v_0\texttt{>} \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha \mid \mu \mid \pi \,\rangle^{\rho'}_{\chi}$$

if $a \in \chi$, and $\rho' = \rho \cup (\mathrm{FV}(v_0) \cap \mathrm{Dom}(\alpha))$

<in : v_0, v_1>

$$\langle\, \alpha \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha \mid \mu \uplus \texttt{<}a \Leftarrow v_0\texttt{>} \mid \pi \,\rangle^{\rho}_{\chi \cup (\mathrm{FV}(v_0) - \mathrm{Dom}(\alpha))}$$

if $a \in \rho$ then $\mathrm{FV}(v_0) \cap \mathrm{Dom}(\alpha) \subseteq \rho$

<migrate : l'>

$$\langle\, \alpha\{ [\![R[migrate(l')], l, r]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\{ [\![R[nil], l', r]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

$l' \in \mathrm{Dom}(()\pi)$

<newloc : Y>

$$\langle\, \alpha\{ [\![R[newloc(Y)], l, r]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\{ [\![R[l'], l, r]\!]_a \} \mid \mu \mid \pi \cup (l' \rightarrow Y) \,\rangle^{\rho}_{\chi}$$

l' fresh, $Y \in (X \rightarrow X)$

<attach : a'>

$$\langle\, \alpha\{ [\![R[attach(a')], l, r]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\{ [\![R[nil], l, r \cup (a' \rightarrow a'')]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

if $(a' \rightarrow a'') \in \pi(l)$

<detach : a'>

$$\langle\, \alpha\{ [\![R[detach(a')], l, r \cup (a' \rightarrow a'')]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\{ [\![R[nil], l, r]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

<register : a', a'', l'>

$$\langle\, \alpha\{ [\![R[register(a', a'', l')], l, r]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [\![R[nil], l, r \cup (a' \rightarrow a'')]\!]_a \} \mid \mu \mid \pi \cup (l' \rightarrow (a' \rightarrow a'')) \,\rangle^{\rho}_{\chi}$$

<unregister : a', a'', l'>

$$\langle\, \alpha\{ [\![R[unregister(a', a'', l')], l, r]\!]_a \} \mid \mu \mid \pi \cup (l' \rightarrow (a' \rightarrow a'')) \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [\![R[nil], l, r]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

Figure 4: MAL Semantics, Part II

The *allow* and *allowloc* primitives can receive as an argument a null access list, represented by $\bot$. This indicates the absence of restrictions on messaging and migration.

We assume as given two sets At(Atoms) and X(Variables), and we then define the set of *values*, V, *expressions*, E, and *messages*, M, as:

$$V = At \cup X \cup \lambda X.E \cup pr(V, V)$$

$$E = V \cup app(E, E) \cup F_n(E^n)$$

$$M = \texttt{<}V \Leftarrow V\texttt{>}_X$$

where $F_n(E^n)$ is all arity-n primitives.

The set of variables, X, including resource names, R, and locations, L, remains the same. The structure of M allows selection of valid senders via access control lists. We define a set of access control lists as $ACL \in \mathbf{P}_{\omega}[Dom(\alpha) \cup \{\bot\}]$. We alter the actor configuration definition to change the actor and location maps, α and π respectively. The actor configuration is written

$$\langle\, \alpha \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

where ρ, μ and χ are as in MAL, $\alpha \in X \xrightarrow{\mathrm{f}} (E \times L \times (R \xrightarrow{\mathrm{f}} X) \times ACL)$, and $\pi \in L \xrightarrow{\mathrm{f}} ((R \xrightarrow{\mathrm{f}} X) \times ACL)$. Changes in α and π reflect the inclusion of access control lists for actors and locations.[5]

3.8 Operational Semantics

We define a transition relation between secure universal actor configurations as the least relation satisying the rules in Figures 5 , 6 and 7.

4. DISCUSSION AND FUTURE WORK

[5] $\alpha(a) = e \times l \times r \times c$ implies that actor a has behavior e and is currently executing at location l with resource map r and access control list c.

$\texttt{<migrate}: l'\texttt{>}$

$$\langle\, \alpha\{\, \texttt{[}R[migrate(l')], l, r, c\texttt{]}_a\} \mid \mu \mid \pi \uplus \texttt{[}c', r'\texttt{]}_l \,\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\, \alpha\{\, \texttt{[}R[nil], l', r, c\texttt{]}_a\} \mid \mu \mid \pi \uplus \texttt{[}c', r'\texttt{]}_{l'} \,\rangle^{\rho}_{\chi}$$

if $a \in c'$ or $c' = \bot$

$\texttt{<newloc}: Y\texttt{>}$

$$\langle\, \alpha\{\, \texttt{[}R[newloc(Y)], l, r, c\texttt{]}_a\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\, \alpha\{\, \texttt{[}R[l'], l, r, c\texttt{]}_a\} \mid \mu \mid \pi \uplus \texttt{[}\{a\}, Y\texttt{]}_{l'} \,\rangle^{\rho}_{\chi}$$

l' fresh

$\texttt{<attach}: a'\texttt{>}$

$$\langle\, \alpha\{\, \texttt{[}R[attach(a')], l, r, c\texttt{]}_a\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\, \alpha\{\, \texttt{[}R[nil], l, r \cup (a' \to a''), c\texttt{]}_a\} \mid \mu \mid \pi \uplus \texttt{[}c', r' \cup (a' \to a'')\texttt{]}_{l'} \,\rangle^{\rho}_{\chi}$$

$\texttt{<detach}: a'\texttt{>}$

$$\langle\, \alpha\{\, \texttt{[}R[detach(a')], l, r \cup (a' \to a''), c\texttt{]}_a\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\, \alpha\{\, \texttt{[}R[nil], l, r, c\texttt{]}_a\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

$\texttt{<register}: a', a'', l'\texttt{>}$

$$\langle\, \alpha\{\, \texttt{[}R[register(a', a'', l')], l, r\texttt{]}_a\} \mid \mu \mid \pi \uplus \texttt{[}c', r'\texttt{]}_{l'} \,\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\, \alpha\{\, \texttt{[}R[nil], l, r \cup (a' \to a'')\texttt{]}_a\} \mid \mu \mid \pi \uplus \texttt{[}c', r' \cup (a' \to a'')\texttt{]}_{l'} \,\rangle^{\rho}_{\chi}$$

if $(a' \to a'') \in r$

$\texttt{<unregister}: a', a'', l'\texttt{>}$

$$\langle\, \alpha\{\, \texttt{[}R[unregister(a', a'', l')], l, r\texttt{]}_a\} \mid \mu \mid \pi\texttt{[}c', r' \cup (a' \to a'')\texttt{]}_{l'} \,\rangle^{\rho}_{\chi} \mapsto$$

$$\langle\, \alpha\{\, \texttt{[}R[nil], l, r\texttt{]}_a\} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

if $(a' \to a'') \in r$

Figure 6: SMAL Semantics, Part II: Mobility

The actor model was first created by Hewitt and his group at MIT [Hew77] in the late 1970s. The model has been further developed by Agha [Agh86] and his group at UIUC. Agha, Mason, Smith and Talcott [AMST97] have developed a simple actor language as an extension to the lambda calculus, its operational semantics and they have studied a family of equivalence relations on actor expressions. Talcott has developed an interaction semantics for actor systems [Tal96, Tal98]. These forms of actor semantics have been the basis of many studies on extensions to the actor model (e.g., for coordination [FA93, VA99, AJV01, FV02], real-time [RAS96], software architectures [AA98], fault-tolerance [SA94], adaptive and meta-level architectures [VTA01], and artificial intelligence [AJ99]).

Several research groups have been trying to achieve distributed computing on a large scale. Berkeley's NOW project has been effectively distributing computation in a "building-wide" scale [ACP95], and Berkeley's Millennium project is exploiting a hierarchical cluster structure to provide distributed computing on a "campus-wide" scale [BGC98]. The Globus project seeks to enable the construction of larger *computational grids* [FK98]. Caltech's Infospheres project has a vision of a worldwide pool of millions of objects (or agents) much like the pool of documents on the World-Wide Web today [CRS+96]. WebOS seeks to provide operating system services, such as client authentication, naming, and persistent storage, to wide area applications [VAD+98]. UIUC's 2K is an integrated operating system architecture addressing the problems of resource management in heterogeneous networks, dynamic adaptability, and configuration of component-based distributed applications [KCMN99].

Security for distributed systems has been looked at for a number of other agent systems. Safe Mobile Ambients [LS00] restrict mobile ambients [CG98] so that sensitive operations such as entering, exiting, and opening an ambient are performed with common agreement. While safe ambients preserve the expressibility of mobile ambients, they prevent programming mistakes by controlling undesirable *grave* interferences. The Seal calculus [VC98], resembles ambients, with two important exceptions. First, *seals* can only move with the environment's control, and the π-calculus is used as a basis for computation, rather than mobility itself. Other methods have been used for securing code on different levels. Ideas such as proof-carrying code [Nec97] and stack inspec-

<allow : a'>

$$\langle\, \alpha\{ [\![R[allow(a')], l, r, c]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\{ [\![R[nil], l, r, c \cup \{a'\}]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

<allow : $\bot$>

$$\langle\, \alpha\{ [\![R[allow(nil)], l, r, c]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\{ [\![R[nil], l, r, \bot]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

<allowloc : a'>

$$\langle\, \alpha\{ [\![R[allowloc(a')], l, r, c]\!]_a \} \mid \mu \mid \pi \uplus [\![c', r']\!]_l \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [\![R[nil], l, r, c]\!]_a \} \mid \mu \mid \pi \uplus [\![c' \cup \{a'\}, r']\!]_l \,\rangle^{\rho}_{\chi}$$

<allowloc : $\bot$>

$$\langle\, \alpha\{ [\![R[allowloc(\texttt{nil})], l, r, c]\!]_a \} \mid \mu \mid \pi \uplus [\![c', r']\!]_l \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [\![R[nil], l, r, c]\!]_a \} \mid \mu \mid \pi \uplus [\![\bot, r']\!]_l \,\rangle^{\rho}_{\chi}$$

<disallow : a'>

$$\langle\, \alpha\{ [\![R[disallow(a')], l, r, c \cup \{a'\}]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [\![R[nil], l, r, c]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

<disallow : $\bot$>

$$\langle\, \alpha\{ [\![R[disallow(\texttt{nil})], l, r, c]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [\![R[nil], l, r, \emptyset]\!]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

<disallowloc : a'>

$$\langle\, \alpha\{ [\![R[disallowloc(a')], l, r, c]\!]_a \} \mid \mu \mid \pi \uplus [\![c' \cup \{a'\}, r']\!]_l \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [\![R[nil], l, r, c]\!]_a \} \mid \mu \mid \pi \uplus [\![c', r']\!]_l \,\rangle^{\rho}_{\chi}$$

<disallowloc : $\bot$>

$$\langle\, \alpha\{ [\![R[disallowloc(\texttt{nil})], l, r, c]\!]_a \} \mid \mu \mid \pi \uplus [\![c', r']\!]_l \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [\![R[nil], l, r, c]\!]_a \} \mid \mu \mid \pi \uplus [\![\emptyset, r']\!]_l \,\rangle^{\rho}_{\chi}$$

Figure 7: SMAL Semantics, Part III: Access Control

tion [WF98] are methods of protecting hosts, also discussed in [ST98].

While there are excellent algorithms for load balancing in clusters and other more static environments, e.g., random stealing and cluser-aware random stealing [vNKB01], the dynamic and heterogeneous nature of the nodes on the WWC make such algorithms much less efficient, especially when IO's peer-to-peer nature is taken into account. Currently, IO's peer-to-peer network is a variant of Gnutella [Cli00]; however, in the future, implementing IO on top of an already existing peer-to-peer network such as JXTA[TAD+02] may prove to be a more interesting option.

5. ACKNOWLEDGEMENTS

We would like to thank the Worldwide Computing group at RPI for input on these semantics. We would especially like to thank Carolyn Talcott for her help in finalizing the semantics.

6. REFERENCES

M. Astley and G. Agha. Modular construction and composition of distributed software architectures. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE '98)*, 1998.

Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for Networks of Workstations: NOW. *IEEE Micro*, February 1995.

G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III*, pages 345–363. International Federation of Information Processing Societies (IFIP), Elsevier Science Publisher, 1993.

G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT

Press, 1986.

G. Agha and N. Jamali. Concurrent programming for distributed artificial intelligence. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to DAI.*, chapter 12. MIT Press, 1999.

G. Agha, N. Jamali, and C. Varela. Agent Naming and Coordination: Actor Based Models and Infrastructures. In A. Ominici, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents*, chapter 9, pages 225–248. Springer-Verlag, 2001.

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

Philip Buonadonna, Andrew Geweke, and David E. Culler. An implementation and analysis of the virtual interface architecture. In *Proceedings of Supercomputing '98*, Orlando, FL, November 1998.

L. Cardelli and A.D. Gordon. Mobile ambients. In *Foundations of System Specification and Computational Structures*, LNCS 1378, pages 140–155. Springer Verlag, 1998.

Clip2.com. The gnutella protocol specification v0.4, 2000.

K. M. Chandy, A. Rifkin, P. A. G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A World-Wide Distributed System Using Java and the Internet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, New York, U.S.A., Aug 1996.

S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP 1993*. Springer Verlag, 1993. LNCS 707.

J. Ferber and J. Briot. Design of a concurrent language for distributed artificial intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 755–762. Institute for New Generation Computer Technology, 1988.

I. Foster and C. Kesselman. The Globus Project: A Status Report. In J. Antonio, editor, *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, pages 4–18. IEEE Computer Society, March 1998.

John Field and Carlos Varela. Toward a programming model for building reliable systems with distributed state. In *Proceedings of the First International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA).*, Brno, Czech Republic, August 2002.

C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

Furio Honsell, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 1995.

W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, 1995.

F. Kon, R. Campbell, M. Dennis Mickunas, and K. Nahrstedt. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1999.

Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *Symposium on Principles of Programming Languages*, pages 352–364, 2000.

George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

S. Ren, G. A. Agha, and M. Saito. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing*, 36:4–12, 1996.

D. C. Sturman and G. Agha. A protocol description language for customizing failure semantics. In *Proceedings of the 13th Symposium on Reliable Distributed Systems.* IEEE Computer Society Press, October 1994.

Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. *Lecture Notes in Computer Science*, 1419:44–??, 1998.

Bernard Traversat, Mohamed Abdelaziz, Mike Duigou, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project jxta virtual network, February 2002.

C. Talcott. Interaction semantics for components of distributed systems. In *First IFIP workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS '96)*, Paris, France, March 1996.

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

C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The extensible services switch in Carnot. *IEEE Parallel and Distributed Technology*, 1(2):16–20, May 1993.

C. Varela and G. Agha. A Hierarchical Model for Coordination of Concurrent Activities. In P. Ciancarini and A. Wolf, editors, *Third International Conference on Coordination Languages and Models (COORDINATION '99)*, LNCS 1594, pages 166–182, Berlin, April 1999. Springer-Verlag. http://osl.cs.uiuc.edu/Papers/Coordination99.ps.

Amin Vahdat, Thomas Anderson, Michael Dahlin, David Culler, Eshwar Belani, Paul Eastham, and Chad Yoshikawa. WebOS: Operating System Services For Wide Area Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Computing*, July 1998.

Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *ICCL Workshop: Internet Programming Languages*, pages 47–77, 1998.

Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. *ACM*, 36:34–43, 2001.

Nalini Venkatasubramanian, Carolyn Talcott, and Gul A. Agha. A formal model for reasoning about adaptive qos-enabled middleware. In *Formal Methods Europe (FME 2001)*. Humboldt-Universitt, Berlin, Germany, March 2001.

Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *19th IEEE Symposium on Security and Privacy*, pages 52–63, 1998.

<**fun** : a>

$$e \xrightarrow{\lambda}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \langle\, \alpha\{ [e, l, r, c]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [e', l, r, c]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

<**new** : a, a'>

$$\langle\, \alpha\{ [R[new(e)], l, r, c]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [R[a'], l, r, c]_a, [e, l, r, \{a, a'\}]_{a'} \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi}$$

a' fresh

<**send** : a, v_0, v_1>

$$\langle\, \alpha\{ [R[send(v_0, v_1)], l, r, c]_a \} \mid \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [R[nil], l, r, c]_a \} \mid \mu \uplus \texttt{<} v_2 \Leftarrow v_1 \texttt{>}_a \mid \pi \,\rangle^{\rho}_{\chi}$$

if $r(v_0)=v_2$

$$\langle\, \alpha\{ [R[nil], l, r, c]_a \} \mid \mu \uplus \texttt{<} v_3 \Leftarrow v_1 \texttt{>}_a \mid \pi \,\rangle^{\rho}_{\chi}$$

if $\pi(l)(v_0)=v_3$

$$\langle\, \alpha\{ [R[nil], l, r, c]_a \} \mid \mu \uplus \texttt{<} v_0 \Leftarrow v_1 \texttt{>}_a \mid \pi \,\rangle^{\rho}_{\chi}$$

if $v_0 \notin \mathrm{Dom}(()r)$ & $v_0 \notin \mathrm{Dom}(()\pi)$

<**receive** : v_0, v_1>

$$\langle\, \alpha\{ [R[ready(v)], l, r, c]_{v_0} \} \mid \texttt{<} v_0 \Leftarrow v_1 \texttt{>}_a \uplus \mu \mid \pi \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha\{ [R[app(v, v_1)], l, r, c]_{v_0} \} \mid \mu \mid \ell \,\rangle^{\rho}_{\chi}$$

if $a \in c$ or $c = \bot$

<**out** : v_0, v_1>

$$\langle\, \alpha \mid \mu \uplus \texttt{<} a \Leftarrow v_0 \texttt{>}_a \mid \pi \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha \mid \mu \mid \pi \,\rangle^{\rho'}_{\chi}$$

if $a \in \chi$, and $\rho' = \rho \cup (\mathrm{FV}(v_0) \cap \mathrm{Dom}(\alpha))$

<**in** : v_0, v_1>

$$\langle\, \alpha \mid \mu \mid \ell \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \alpha \mid \mu \uplus \texttt{<} a \Leftarrow v_0 \texttt{>}_{a'} \mid \pi \,\rangle^{\rho}_{\chi \cup (\mathrm{FV}(v_0) - \mathrm{Dom}(\alpha))}$$

if $a \in \rho$ then $\mathrm{FV}(v_0) \cap \mathrm{Dom}(\alpha) \subseteq \rho$

Figure 5: SMAL Semantics, Part I: Actor Semantics