Semantics-preserving Sharing Actors

Mohsen Lesani[{1,2}] Antonio Lain[2]
University of California, Los Angeles[1] HP Labs, Palo Alto[2]

Abstract

Actors interact by asynchronous message passing. A key semantic property of actors is that they do not share state. This facilitates data-race freedom, fault isolation and location transparency. On the other hand, strict avoidance of sharing can lead to inefficiency. We propose the sharing actor programming model that extends the actor programming model with single-writer multiple-reader sharing of data. We define the sharing actor theory and prove its semantic equivalence to the pure actor theory. We realize the sharing actor theory with an efficient implementation. The implementation benefits from sharing data but keeps it transparent to actors. To increase the confidence that the implementation complies with the semantics, we have built a checking tool based on deterministic replay of actor programs.

1. Introduction

1.1. Background

Actors is a well-known model of concurrent programming for parallel and distributed systems. The term *actor* was first used by Hewitt [10] to refer to reflexive agents and later to a model of concurrent computing. The commonly used semantics of actors is formalized by Agha [1] in 1986. Since then, many actor languages and frameworks have been developed. With the growth of parallel and distributed computing platforms such as multi-core architectures and cluster computers in recent years, the actor model has gained popularity. Contemporary actor languages and frameworks include: Ericsson's Erlang programming language [24][19] that supports massively concurrent telecom systems [2], Akka Actors and the Scala Actors library [8], Ptolemy project [14], JCoBox [20], SALSA [23], Microsoft Asynchronous Agents Library, Microsoft Axum, and Microsoft Research Orleans framework for cloud computing [4] to name a few.

A key semantic property of the pure actor model is encapsulation of state. Actors do not share state: an actor must explicitly send a message to another actor in order to affect its behavior. Most shared memory implementations of actors can bypass this sharing restriction by including references to mutable state within messages. As deviation from the non-sharing philosophy can be troublesome, researches have proposed static analysis methods and type systems to rule out sharing [22][9][17][7]. In fact, the growing popularity of the actor model is largely due to this simple model of data sharing. This restricted model of sharing provides a uniform framework for exploiting parallelism both within a server and across a data center. In addition, it facilitates data-race freedom, fault isolation and location transparency. Unfortunately, this comes at a price in terms of the convenience and efficiency of sharing. In particular, in scenarios where large-scale mutable but read-dominated data is needed by a large number of actors, a pure message-based approach can become extremely inefficient.

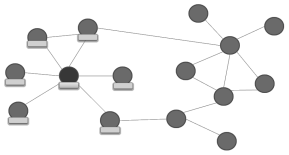

Figure 1. Distributed Social Graph

1.2. Motivating Example

A tangible example is a distributed social graph. Consider Figure 1. A node in the graph is the session of a user and an edge in the graph is a friendship relation. The dark circle represents the session of a definite user and the rectangle below it is the session state. When a user changes her session state, her friends should be notified. Each session state is a single-writer multiple-readers data that is written by one user and read by her friends. Typically, the social graph is too large to fit in the memory of a single computer and needs to be partitioned across the cluster. Ideally, the partitioning algorithm minimizes the number of edges that span two machines while keeping the load balanced. The sessions that are mapped to the same machine can potentially leverage the shared memory for efficiency; for example, by avoiding duplication of session state.

Let's look at the previous approaches to programming the single-writer multiple-readers abstraction. Ad-hoc multi-threaded implementations can share the data of a user session between the sessions of her friends in each host and control accesses by the conventional synchronization mechanisms such as locks and conditional variables. These implementations can be efficient but are prone to traditional problems of programming shared memory such as deadlocks or races.

Implementations that use the pure actor model can employ either full replication or a delegate actor. We consider each one in turn. Consider Figure 2 where large rectangles represent hosts, circles represent actors (the darker circle represents the writer actor) and small rectangles represent state. In the first implementation, the state is replicated in each reader actor. When the writer actor finishes updating its state, it sends an update message to each reader actor. The replica at each reader is updated when it processes the update message. Unfortunately, this approach wastes space and more importantly time as the same update is repeated at each reader actor. The second implementation employs a delegate actor at each reader host. The state is replicated and updated once at the delegate actor. The reader actors communicate with the delegate actor to read the state. The problem with this approach is the loss of potential parallelism as the accesses of the readers are serialized in the delegate actor.

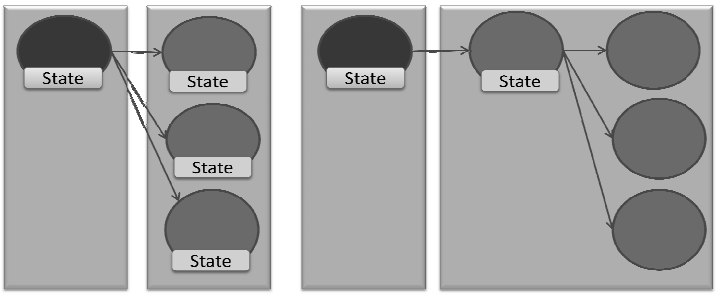

Figure 2. (Left) Full Replication (Right) Delegate Actor

1.3. Our Approach

We propose sharing actors. In this approach, state can be shared between a single writer actor and multiple reader actors. The writer and reader actors see the shared state as part of their internal state. The writer actor can read from and write to the shared state while the reader actors have a read-only access to it. This approach merges programmability and efficiency, namely the state-abstraction of actors and the efficiency of sharing.

Consider sharing actors depicted in Figure 3. Each friend session views the user session state as part of its private internal state (similar to the full replication case explained above). This is while the implementation shares a single replica between friends (similar to the ad-hoc sharing mentioned above). This approach provides the programmer with the high level programming model of actors and at the same time avoids duplicate state and repeated updates. As the number of readers increase this benefit increases.

Our Cloud Assistant Framework (CAF) provides an open source implementation of Sharing Actors[1]. CAF abstracts aspects of distributed computing, and this enables front end mobile application programmers to codesign back end components in JavaScript (using Node.js[2]). A Cloud Assistant (CA) is an actor that permanently represents a mobile application instance in the data center. CAs act as light-weight, stateful proxies that autonomously interact with other CAs, cloud services, or the external world. CAF examples can be tried online.[3]

Figure 4 shows code snippets from CAF Sharing Actors. It is from a sample application called moody friends. Every user is represented in the cloud by a CA and shares a map containing his current mood and place with her friends. Changes to mood and place are eventually propagated to friends. The shared map is updated atomically. Therefore, friends will always see consistent mood and location tuples. Figure 4.(A) shows an initialization phase where the owner of the session registers as the writer actor of the session map (with the name `mySession`) and the other actors register on the session map as reader actors (with the name `friendSession`). Figure 4.(B) shows that the owner actor writes to the session map and the reader actors read from it.

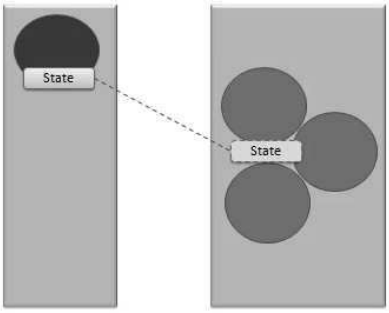

Figure 3. Sharing Actors

We formalize the semantics of sharing actor theory as an extension of the pure actor theory. The extension is the notion of single-writer multiple-reader state of actors. A program translation from the sharing actor theory to the pure subset (without sharing) of it is defined. We prove that for every sharing actor program, the translated pure actor program has equivalent interaction-semantics that is every interaction by the sharing actor program is a possible interaction with the translated pure actor program. This justifies that our extension is only semantic-sugar for the pure actor model.

Naive implementations of the sharing semantics can lead to inefficiencies in terms of both time and space, in particular, when there is a large number of reader actors. We describe an optimized implementation. It benefits from a versioned data structure shared by all the readers co-located in the same host but keeps this sharing transparent to the readers.

To check that the implementation complies with the semantics, we have built a checking tool. The checker uses the direct implementation of the semantics as the reference implementation. It plays and logs with the optimized implementation and performs a deterministic replay with the reference implementation. Any discrepancy between the final configurations of the two actor systems suggests a bug in the optimized implementation.

The structure of the paper is as follows. We formalize the sharing actor theory in the next section. It includes the definition of the syntax, configurations, transitions, equivalence, program

```
if (this.state.admin) {
   this.sharing.addMap(
      true, MAP_NAME, 'mySession');
} else {
   var mapName = getFriendMapName();
   this.$.sharing.addMap(
      false, mapName, 'friendSession');
}
```

(A)

```
var self = this;
if (this.state.admin) {
   this.$.sharing.mySession.status = 'Happy'
   this.$.sharing.mySession.place = 'Paris'
   // ...
} else {
  var herStatus = this.$.sharing.friendSession.status
  var herPlace = this.$.sharing.friendSession.place
  // ...
}
```

(B)

Figure 4. Sharing Actors in CAF

[1] http://www.cafjs.com/
[2] http://www.nodejs.org
[3] http://www.cafjs.com/examples.html

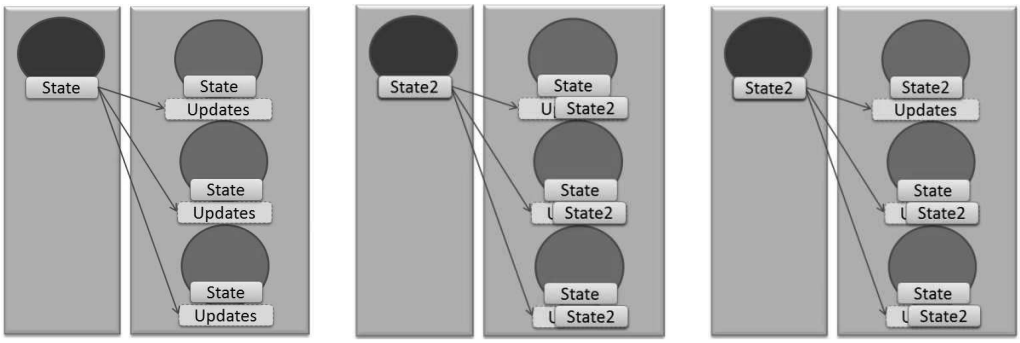

Figure 5. (A) Writer and reader actors (B) Writer changed state (C) The first reader actor installed the update

translation and the theorem of equivalence. Implementation is described afterwards. Next, we explain the checking tool. Finally, related works and conclusion sections conclude the paper.

2. Sharing Actor Theory

In this section, we define the sharing actor theory as an extension of the pure actor theory. In the pure actor theory, each actor has a message queue. An actor dequeues and processes messages sequentially. The sharing actor theory allows using single-writer multiple-reader data. Consider Figure 5.A. The sharing actor semantics adds an update queue to each reader actor. Consider Figure 5.B. When the writer actor finishes updating the state, the state is put at the end of the update queue of each reader actor. Consider Figure 5.C. A reader actor can take an element from the update queue before processing a new message.

We define a translation that transforms a sharing actor program to a pure actor program where the only way to change the internal state of an actor is to send a message to it. We prove that each sharing program is interaction-equivalent to the translated program. The key properties of the sharing actor semantics that enable the equivalence to the pure actor semantics are atomicity, isolation and fairness.

- Atomicity means that partial changes of the writer to the state should not be externalized. In the context of the above example, each friend either sees all or none of the state updates of a single user action.
- Isolation means that new updates should not become visible to an actor while it is processing a message. In the previous example, each friend always sees a consistent (but possibly outdated) view of a user session.
- Fairness means that the propagation of an update should not be delayed arbitrarily. In the context of the above example, each friend can eventually see the updated session of the user regardless of how busy the other friends are.

Our sharing actor theory does not allow multiple-writer data. The shared state is partitioned into owners with exclusive write access. When an actor wants to modify a part of the shared state that it does not own it has to send a message (containing the update or a closure that will perform the update) to the owner actor.

We formalize our model in the following subsections. The definitions and notations are based on the formulation of pure actors by Mason and Talcott [16].

2.1. Basic definitions

Let $\mathcal{P}_\omega(S)$ denote the power set of S and $\mathcal{M}_\omega(S)$ denote the multiset power set of S. Let $e :: s$ denote a sequence of elements where e is the first element and $s :: e$ denote a sequence of elements where e is the last element. We use $let\ \{\overline{x := e}\}\ e'$ as a syntactic sugar for $(\lambda \overline{x}.e')\ \overline{e}$. We define $true \triangleq \lambda tf.t$ and $false \triangleq \lambda tf.f$.

2.2. Syntax

The syntax is depicted in Figure 6. Id is the set of identifiers $\{id_1, id_2, \dots\}$. A is the set of actor names $\{a_1, a_2, \dots\}$. $M = Id[V^*]$ is the set of message contents. A message content is a method identifier Id and argument values V^*. The set of messages is $Msg = A \lhd M$ (that is set isomorphic to $A \times M$). $a \lhd m : Msg$ denotes a message for actor a with contents m.

A program for an actor subsystem declares its interaction interface with other actor subsystems as *Receptionists* and *Externals* sets of actors. *Receptionists* is a subset of actors of this actor subsystem that are visible to the outside world. *Externals* is the set of actors that are known from the external world. The

$$
\begin{aligned}
P ::= \text{program}(&receptionists: \mathcal{P}_\omega[A], externals: \mathcal{P}_\omega[A],\\
&library: \mathcal{P}_\omega[BehDef],\\
&actors: \mathcal{P}_\omega[A \mapsto E],\\
&messages: \mathcal{M}_\omega[A \lhd M],\\
&sharing: \langle a_w: A, A_r: \mathcal{P}_\omega[A]\rangle)
\end{aligned}
$$

where

$$\forall a \in A: |\{(a \mapsto e) \in actors \mid e \in E\}| \leq 1$$

$$\{a_w\} \cup A_r \subseteq dom(actors)$$

$$
\begin{aligned}
BehDef &::= \text{behavior } Id(X^*)\ MethDef^*\\
MethDef &::= \text{method } Id(X^*)\ [\text{enable } E]\ E\\
E &::= \lambda X.E\\
&\mid E\ E\\
&\mid E \lhd Id[E^*]\\
&\mid \text{ready}(Id(E^*))\\
&\mid \text{self}\\
&\mid X \mid V\\
V &::= \lambda X.E \mid A\\
X &::= Id\\
M &::= Id[V^*]\\
Id &= \{id_1, id_2, \dots\}
\end{aligned}
$$

Figure 6. Syntax

Definition of $\rightarrow$:

$internal$	$\dfrac{l: I_1 \Rightarrow I_2}{\ll I_1, I \gg_X^\rho \xrightarrow{l} \ll I_2, I \gg_X^\rho}$	Eq. 1
in	$\ll I \gg_X^\rho \xrightarrow{in(a,m)} \ll I, a \lhd m \gg_{X \cup acq(m) - \rho}^\rho \quad if\ (a \in \rho) \wedge (acq(m) \cap InAct(I) \subseteq \rho)$	Eq. 2
out	$\ll I, a \lhd m \gg_X^\rho \xrightarrow{out(a,m)} \ll I \gg_X^{\rho \cup (acq(m) - X)} \quad if\ a \notin InAct(I)$	Eq. 3
$idle$	$\ll I \gg_X^\rho \xrightarrow{idle} \ll I \gg_X^\rho$	Eq. 4

Definition of $\Rightarrow$:

$deliver(a,m)$	$[bid(\overline{v}), \langle [\,], q_r \rangle]_a, a \lhd m \Rightarrow [bid(\overline{v}), \langle [m], q_r \rangle]_a$	Eq. 5
$traverse(a)$	$[bid(\overline{v}), \langle m :: q_u, q_r \rangle]_a \Rightarrow [\langle e_c, e_b, bid(\overline{v}) \rangle, \langle m, q_u, q_r \rangle]_a$ $if\ methMatch(Lib, bid, \overline{v}, m) = \langle e_c, e_b \rangle$	Eq. 6
$check(a)$	$\dfrac{e \rightarrow_a e'}{[\langle e, e_b, bid(\overline{v}) \rangle, \langle m, q_u, q_r \rangle]_a \Rightarrow [\langle e', e_b, bid(\overline{v}) \rangle, \langle m, q_u, q_r \rangle]_a}$	Eq. 7
$disable(a)$	$[\langle v, e_b, bid(\overline{v}) \rangle, \langle m, q_u, q_r \rangle]_a \Rightarrow [bid(\overline{v}), \langle q_u, q_r :: m \rangle]_a \quad if\ v \neq \text{true}$	Eq. 8
$enable(a)$	$[\langle \text{true}, e_b, bid(\overline{v}) \rangle, \langle m, q_u, q_r \rangle]_a \Rightarrow [e_b, q_u . q_r]_a$	Eq. 9
$seq(a)$	$\dfrac{e \rightarrow_a e'}{[e, q_u]_a \Rightarrow [e, q_u]_a}$	Eq. 10
$send(a)$	$[\mathcal{R}[\![v \lhd mid[\overline{v}]]\!], q_u]_a \Rightarrow [\mathcal{R}[\![0]\!], q_u]_a, v \lhd mid[\overline{v}] \quad if\ v \in A$	Eq. 11
$ready(a)$	$\left[\mathcal{R}[\![\text{ready}(bid(\overline{v}))]\!], q_u\right]_a \Rightarrow [bid(\overline{v}), \langle q_u, [\,] \rangle]_a \quad if\ behMatch(Lib, bid, \overline{v})\ and\ a \neq a_w$	Eq. 12

The rules that we add to the pure semantics are:

$ready(a_w)$	$\left[\mathcal{R}[\![\text{ready}(bid(\overline{v}))]\!], q_u\right]_{a_w}, [b, q, u]_{a_{r_{i=1..n}}} \Rightarrow$ $[bid(\overline{v}), \langle q_u, [\,] \rangle]_{a_w}, [b, q, u :: bid(\overline{v})]_{a_{r_{i=1..n}}}$ $if\ behMatch(Lib, bid, \overline{v})$	Eq. 13
l	$\dfrac{l: [b, q]_a \Rightarrow [b', q']_a}{[b, q, u]_a \Rightarrow [b', q', u]_a}$	Eq. 14
$update(a_r, bid(\overline{v}))$	$\left[bid'\left(\overline{v'}\right), \langle [\,], q_r \rangle, bid(\overline{v}) :: u\right]_{a_r} \Rightarrow [bid(\overline{v}), \langle q_r, [\,] \rangle, u]_{a_r}$	Eq. 15

Definition of $\rightarrow_a$:

rdx	$\dfrac{e \rightarrow_a e'}{\mathcal{R}[\![e]\!] \rightarrow_a \mathcal{R}[\![e']\!]}$	Eq. 16
$beta$	$(\lambda x. e)\ v \rightarrow_a e[x := v]$	Eq. 17
$self$	$\text{self} \rightarrow_a a$	Eq. 18

Figure 7. Semantics

program defines a library of behaviors, a set of initial actors and messages. There is one sharing group in each actor program. A sharing group $G = \langle a_w, A_r \rangle$, is a writer actor a_w and a set of reader actors $A_r = \{a_{r_{i=1..R}}\}$. The behaviors of the reader actors are updated with the new behaviors of the writer actor.

Each behavior definition $BehDef$ defines a behavior with a behavior identifier Id, initialization parameters X^* and a set of method definitions $MethDef^*$. A method definition $MethDef$ specifies the method parameters X^*, the optional selective receive $[\text{enable}\, E]$ and the body expression E of the method. The selective receive expression (also called guard or synchronization constraint) of a method is required to be functional i.e. its evaluation involves no send or ready expression. If the selective receive is not specified, it is assumed to be $true$. For each candidate message, the selective receive expression is evaluated under the binding of method parameters to the arguments in the message. If the selective receive is evaluated to $true$, the body expression is evaluated; otherwise, the message is rejected at this time and is considered again later. An expression is a lambda abstraction $\lambda X. E$, application $E\ E$, a variable X, a value V or an actor specific expression. Actor specific expressions are sending a message $E \lhd Id[E^*]$, installing a new behavior $\text{ready}(Id(E^*))$ and the current actor expression self. In a send expression $E_a \lhd Id[\overline{E}]$, the target of the message is the value of E_a and the message content has Id as the method name and values of $\overline{E}$ as arguments. The ready expression $\text{ready}\left(Id(\overline{E})\right)$ installs the behavior Id with the values of $\overline{E}$ as arguments. $self$ is evaluated to the name of the executing actor. A value is either a lambda expression or an actor name. For the sake of simplicity, our syntax does not support dynamic creation of actors.

2.3. Configuration

An actor configuration is a pool of actors with definite states and messages. The set of actor states S is defined as follows:

$S ::= (B, Q) \mid (B, Q, U)$ Eq. 19

$B ::= Id(V^*) \mid \langle E, E, Id(V^*)\rangle \mid E$
$Q ::= \langle M^*, M^*\rangle \mid \langle M, M^*, M^*\rangle \mid M^*$
$U ::= Id(\overline{V})^*$

B is the behavior component of an actor state. If B is $Id(V^*)$, the behavior Id with arguments V^* is installed. If B is $\langle E, E, Id(V^*)\rangle$, a message is being checked for enabledness where the first E is the guard expression that is being evaluated, the second E is the body expression and $Id(V^*)$ is the latest installed behavior. If B is an E, the actor is evaluating the body of a method. Q is the queue component of an actor state. If Q is $\langle M^*, M^*\rangle$, the first and second components represent the unchecked and rejected queues of messages respectively. If Q is $\langle M, M^*, M^*\rangle$, the first component is the message that is being checked for enabledness and the second and third components are the unchecked and rejected queues. If Q is M^*, it represents the queue of (unchecked) messages to be checked by the next behavior. U is the sequence of updates for a reader actor. $s{:}S$ denotes an actor state. $b{:}B$ denotes a behavior component. $q{:}Q$ denotes a queue component. $u{:}U$ denotes an updates list component.

An actor entity is an actor name paired with an actor state. Actor entities $AE = [S]_A$ is the set of actor entities. (AE is set isomorphic to $A \times S$). $[s]_a{:}[S]_A$ denotes an actor named a with state s. The interior of an actor configuration is a set of actors and multiset of messages. The set of configuration interiors is defined as

$$IS = \{\sigma \cup \mu \mid \sigma \in \mathcal{M}_\omega[AE], \mu \in \mathcal{M}_\omega[Msg]\} \quad \text{EQ. 20}$$

where $\forall I \in IS{:}\, \forall a \in A{:}\, |\{[s]_a \in I \mid s \in S\}| \leq 1$. $I{:}IS$ denotes a configuration interior.

Internal and external actors of a configuration interior are defined as follows:

$$InAct, ExtAct{:}\, I \to \mathcal{P}_\omega[A] \quad \text{EQ. 21}$$
$$InAct(I) = \{a \in A \mid \exists s \in S{:}\, [s]_a \in I\} \quad \text{EQ. 22}$$
$$ExtAct(I) = acq(I) \,/\, InAct(I) \quad \text{EQ. 23}$$

The acquaintance function gives the finite set of actor names occurring in I. A configuration is defined as follows:

$$K = \{\ll I \gg_{\mathcal{X}}^{\rho} \mid \rho \subseteq InAct(I) \wedge ExtAct(I) \subseteq \mathcal{X} \wedge \mathcal{X} \cap InAct(I) = \emptyset\} \quad \text{EQ. 24}$$

The receptionists set ρ is a subset of the internal actors of the interior that are visible from the environment. The externals set $\mathcal{X}$ includes all actors mentioned in the interior that are not internal actors. Consider the following program p.

$$p \triangleq \text{program}(receptionists{:}\,\rho, externals{:}\,\mathcal{X}, library{:}\,Lib, actors{:}\,\{a_i \mapsto e_i\}_{1\leq i\leq m}, messages{:}\,\{a'_i \lhd m_i\}_{1\leq i\leq n}, sharing{:}\,\langle a_w, A_r\rangle) \text{ where} \quad \text{EQ. 25}$$
$$A_r \triangleq \{a_{r_{i=1..R}}\}$$
$$\{a_w\} \cup A_r \subseteq \{a_{i=1..m}\}$$

The initial configuration for p is denoted by $[\![p]\!]$ and is defined as follows:

$$[\![p]\!] \triangleq \ll I \gg_{\mathcal{X}}^{\rho} \text{ where}$$
$$I \triangleq \{(e_i, nil)_{1\leq i\leq m, a_i \notin A_r}, (e_i, nil, nil)_{1\leq i\leq m, a_i \in A_r}, (a'_i \lhd m_i)_{1\leq i\leq n}\} \quad \text{EQ. 26}$$

2.4. Semantics

The semantics of sharing actors is defined in Figure 7. The reduction context and the helper functions are defined in Figure 8. The internal transition relation $\Rightarrow$ defines steps of actor computation inside the current subsystem. The internal transition $(l{:}\, I \Rightarrow I')$ denotes the transition of I to I' by $\Rightarrow$ with label l. $InAct(I)$ is called the old actors of l and $InAct(I')/InAct(I)$ are called the new actors of l. We explain each internal transition in turn.

To maintain fairness for processing of messages that are rejected by previous behaviors of the actor, a new message is delivered only after all the messages in the unchecked queue are checked and rejected by the current behavior. If the unchecked queue is empty, the $deliver(a, m)$ rule receives message m and puts it in the unchecked queue.

The $traverse(a)$ rule takes a message m from the head of the unchecked queue and uses $methMatch$ to check whether the method name and the argument list of the message match a method name and parameter list of the current behavior. If there is a matching method, $methMatch$ returns the pair of the guard and body expressions that are instantiated with the passed arguments. The resulting behavior tuple contains the guard and body expressions and the current behavior. Later, the guard expression may be evaluated to a normal expression other than true by the $check(a)$ rule and then the message should be put to the rejected queue. Thus, the message m is kept as the first element of the queue tuple.

The $check(a)$ rule evaluates the guard expression. If the guard expression is evaluated to a value other than true, the $disable(a)$ rule drops the tentative body expression, restores the current

$$\mathcal{R} ::= [\![\,]\!] \mid \mathcal{R}\,E \mid V\,\mathcal{R} \mid \mathcal{R} \lhd Id[E^*] \mid V \lhd Id[V^*, \mathcal{R}, E^*] \mid ready(Id(V^*, \mathcal{R}, E^*))$$

$parCheck(\overline{x}, \overline{v})$ iff $\overline{x}$ and $\overline{v}$ are of the same length. EQ. 27

$behMatch(Lib, bid, \overline{v})$ iff $(\exists \overline{x}, \overline{methodDef}{:}\, (\text{behavior } bid(\overline{x})\ \overline{methodDef}) \in Lib) \wedge parCheck(\overline{x}, \overline{v})$. EQ. 28

$$methodMatch(Lib, bid, \overline{v}, mid[\overline{v'}]) = \begin{cases} \langle e[\overline{x} := \overline{v}][\overline{x'} := \overline{v'}], e'[\overline{x} := \overline{v}][\overline{x'} := \overline{v'}]\rangle & if\ (\exists \overline{x}, \overline{methodDef}{:}\, (\text{behavior } bid(\overline{x})\ \overline{methodDef}) \in Lib) \wedge \\ & (\text{method } mid(\overline{x'})\ [\text{enable } e]\ e' \in \overline{methodDef}) \wedge \\ & (parCheck(\overline{x'}, \overline{v'})) \\ false & otherwise \end{cases} \quad \text{EQ. 29}$$

Figure 8. Reduction Context and Helper Functions

behavior and puts the message at the tail of the rejected queue. If the guard expression is evaluated to true, the body expression is installed, the backed-up message is dropped and the concatenation of the unchecked and rejected queues is stored as the unchecked queue for the next behavior. The $seq(a)$ rule evaluates the body expression.

The $send(a)$ rule checks if the recipient expression of the message is an actor name and emits the message.

The $ready(a)$ rule (where a is not the writer actor) uses $behMatch$ to check whether the behavior name exists in the library and the argument and parameter lists match. If there is a match, the new behavior is installed and the tuple of the unchecked queue and an empty rejected queue is stored as the queue component. The $ready(a_w)$ rule (where a_w is the writer actor) does the same and in addition, adds the new behavior to the tail of the update list of each reader actor. To maintain atomicity, the updates are propagated only when the writer has finished processing a message.

A rule mirrors all transitions possible for non-reader actors (without an update list) for reader actors. So, reader actors can have any type of transition that other normal actors can. A reader actor a_r can specifically do $update(a_r)$ transition. The $update(a_r)$ rule takes the behavior at the head of the update list and installs it as the current behavior. To maintain isolation, an update is installed only when the actor is not processing a message. To maintain fairness, similar to the $deliver(a,m)$ rule, an update can be performed only if all the messages in the unchecked queue are checked and rejected by the current behavior.

2.5. Computation and Interaction Semantics

2.5.1. Computation Path Semantics

The set of computation paths $\mathcal{P}$ is the set of sequences of the form

$$\pi = \left[K_i \xrightarrow{l_i} K_{i+1} \mid i \in \mathrm{N}\right] \quad \text{EQ. 30}$$

$l_i \in L \cup in(A,M) \cup out(A,M)$

where L is the set of internal transition labels and the paths with the initial configuration K are defined as

$$\mathcal{P}(K) = \{\pi \in \mathcal{P} \mid K \text{ is the source of } \pi(0)\} \quad \text{EQ. 31}$$

A finite computation is a path in which all but a finite number of the transition labels are $idle$. A label l is enabled in K if $l \in L \cup out(A,M)$ and K has a transition with label l' where (1) l' is the same as l up to choice of names for new actors or (2) K has a transition with label $l' = update(a_r, bid(\overline{v})_0)$ and $[b, q, bid(\overline{v})_{i=0..n}]_{a_r} \in K$ and $l = update(a_r, bid(\overline{v})_i)$, $i = 1..n$. In other words, if the first update is enabled to be installed, all the other pending updates are enabled to be installed in sequence after that. $Enabled(\pi, i)$ is the set of labels that are enabled in the source K_i of $\pi(i)$. $Fired(\pi, i) = l$ if $\pi(i)$ has the form $K_i \xrightarrow{l_i} K_{i+1}$ where l_i differs from l only in the names of new actors.

Actor computations are required to be fair. A computation path is fair if whenever a transition is enabled, either it eventually fires or it becomes permanently disabled. $\mathcal{F}(K)$ is the set of fair paths with the initial configuration K.

$$Fair(\pi) \Leftrightarrow \forall i, l: \; l \in Enabled(\pi, i) \Rightarrow \left(\exists j \geq i: l = Fired(\pi, j)\right) \vee \left(\exists k > i: \forall j > k: l \notin Enabled(\pi, j)\right) \quad \text{EQ. 32}$$

$$\mathcal{F}(K) = \{\pi \in \mathcal{P}(K) \mid Fair(\pi)\} \quad \text{EQ. 33}$$

2.5.2. Interaction Semantics

An actor system is considered as a black box characterized by the set of possible interactions with its environment. Two actor systems are equivalent if they cannot be distinguished by interacting with other actor systems.

Interaction path $isem(\pi)$ of a computation path π is the sequence of its message input and output actions and is defined as follows:

$isem(\pi) = v_{X_0}^{\rho_0}$ iff

$$\pi(i) = \ll I_i \gg_{X_i}^{\rho_i} \xrightarrow{l_i} \ll I_{i+1} \gg_{X_{i+1}}^{\rho_{i+1}} \text{ and } v(i) = isem(l_i) \text{ for } i \in \mathrm{N} \quad \text{EQ. 34}$$

$$isem(l) = \begin{cases} \tau & if\ l \in L \cup \{idle\} \\ l & if\ l \in in(A,M) \cup out(A,M) \end{cases} \quad \text{EQ. 35}$$

Two interaction paths are equivalent if and only ig they differ only by insertion or deletion of τ^*. EQ. 36

τ stands for possible internal activity.

The interaction semantics $Isem(K)$ of a configuration K is defined as follows:

$$Isem(K) = \{isem(\pi) \mid \pi \in \mathcal{F}(K)\} \quad \text{EQ. 37}$$

Consider two adjacent steps $K_0 \xrightarrow{l_0} K_1 \xrightarrow{l_1} K_2$. The two steps can be legally commuted if

(1) The old and new actors of l_0 are disjoint from the old actors of l_1. EQ. 38

(2) The messages produced in the l_0 rule do not participate in the l_1 rule. EQ. 39

(3) l_0 and l_1 are not both interaction labels. EQ. 40

Computation paths that differ only by legal permutations result in the same interaction path [16]. Consider a computation path π. Consider a computation path π' that differs from π only by legal permutations. If an actor system produces π, it produces π' and $isem(\pi) = isem(\pi')$.

2.6. Interaction Equivalence

A configuration K is equivalent to the configuration K' if and only if $Isem(K) = Isem(K')$. In other words, a configuration K is equivalent to the configuration K' if and only if

$$\forall \pi \in \mathcal{F}(K): \exists \pi' \in \mathcal{F}(K'): isem(\pi) = isem(\pi') \wedge \forall \pi' \in \mathcal{F}(K'): \exists \pi \in \mathcal{F}(K): isem(\pi') = isem(\pi) \quad \text{EQ. 41}$$

In order to show the interaction equivalence of sharing actors to pure actors, we define a program translation $s2p$ that removes sharing and translates automatic updates to pure message passing. To reason about intermediate configurations, we lift the program translation to configuration translation. We present the formal definitions of the translation in the appendix [15] and briefly explain it in the following paragraphs.

The interesting part of translation is the translation of $ready$ for the writer and the addition of $uMethod$ to behaviors for the readers. The reduction of $ready$ for a_w (EQ. 13), installs the new behavior for a_w and stores it at the tail of the update store u of each a_r. Later, in $update(a_r)$ transitions (EQ. 15), the behavior at the head of u is taken and installed as the current behavior of a_r. To process updates in order, new behaviors are added at the tail and removed from the head of the update store.

The translation simulates the update mechanism with message passing. A $ready$ expression of a_w is translated to sending $Update$ messages to reader actors a_r and then installing the new behavior for a_w. $Update$ messages contain the new behavior for the reader actors. At the reader actors, $uMethod$ receives $Update$ messages. To preserve the order of updates, the update messages

should be processed in the same order as they are sent. The translation adds a counter parameter to the definition of the writer and readers behaviors. The writer actor maintains a counter c_w that holds the number of the next update to send and each reader actor a_r maintains the number of the next update message to receive c_r. The selective receive feature of the language is used to check the number of the messages. The $Update$ messages that a_w sends contain a pair of expressions. The first element of the pair is the number of the update message and the second element is the new behavior. The guard of the $uMethod$ checks the equality of the first element of the pair (that is the number of the message) to c_r (that is the number of the next update message to receive). If the check succeeds, the $ready$ expression that installs the new behavior is obtained from the second element of the message and is evaluated.

The following theorem states the interaction equivalence of sharing actors and pure actors. We show that every sharing actor system can be translated to a pure actor system such that any interaction of the sharing actor system could be resulted from the pure actor system.

THEOREM 1. $\forall p \in P: Isem([\![p]\!]) = Isem([\![s2p(p)]\!])$

Please see the appendix [15] for the proof.

3. Implementation

As we mentioned in the introduction, CAF provides an open source implementation of Sharing Actors using server-side JavaScript. JavaScript execution model is single-threaded, but asynchronous calls during the processing of a message introduce task interleaving. We have also experimented with multi-threaded garbage collected (Erlang and Scala) and non-garbage collected (C) implementations of the same concepts as HP Labs proprietary libraries.

The implementation provides a set of commonly used ADTs (Abstract Data Types) such as Map and Set that actors can share. There is one actor that owns the shared abstraction. The owner is the only actor that can mutate the abstraction. Each host containing a sharing actor holds one copy of the abstraction. All the sharing actors can transparently access the abstraction as part of their internal state. We use the term epoch to refer to the execution period of an actor that services a single message. After each epoch of the owner actor that mutates the shared abstraction, an update message is sent to sharing hosts. This pushes updates to hosts before they are needed and hence hides communication latency. An epoch by the owner may involve multiple updates on the abstraction. To preserve the atomicity property, one update message containing all the updates that were performed on the abstraction is sent. Serialization of updates is obtained by numbering the update messages at the owner and considering them in order at other hosts.

To provide the isolation property, an actor that is in the middle of processing a message should not see an update. On the other hand, to support the fairness property, other sharing actors on the same host should not wait indefinitely for the update. To meet both of these requirements, the implementation maintains multiple versions of the abstraction at each host. While the old versions are retained for the actively processing actors, new epochs can see new versions. The implementation maintains multiple versions efficiently as follows. We use a simple semi-persistent data structure [6]. See Figure 9.(A). Each copy of the shared abstraction is represented as a reference to the head of the list of

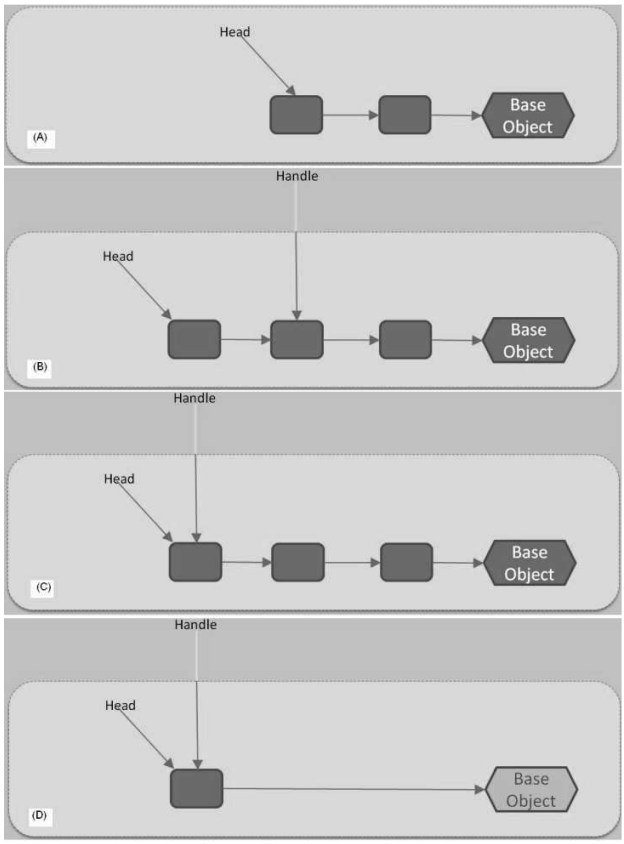

Figure 9. Implementation

update messages. The tail of the update list points to a base object. The base object is an instance of a conventional linearizable implementation of the abstraction. We can benefit from existing high performance data structure implementations. See Figure 9.(B). The update list is initially empty. Received update messages are prepended to the update list and the reference to the head of the list is updated. Obviously, in the multi-threaded implementation, the reference is updated atomically. References to different elements of the update list represent different versions of the abstraction. The head of the update list is the most recent version of the abstraction.

Upon a read request on the abstraction using a handle (that references possibly the middle of the list), the list is traversed starting from the update message that the reference points to. The result of the access may be determined according to an update message that is visited in the traversal. Otherwise the access is finally performed on the base object.

It is obvious that concurrent update and read operations do not block each other. It is notable that this representation avoids memory overhead of replication for each actor and at the same time the synchronization bottleneck of a single delegate actor. Visiting update messages in the traversal is dependent on and can be optimized according to the abstraction type. For example, consider accessing the value of a key in the map abstraction. Having a bloom filter [3] on the keys in each update message can accelerate skipping update messages that are irrelevant to the accessed key.

References of user code to the abstraction are objects of type handle. To have the fairness property, if a handle is accessed for

the first time in an epoch, we want the reference in the handle to be updated to the current head of the update list. To have the isolation property, we want the handle to stay unchanged if the handle is accessed again in an epoch. The strategy for keeping track of references influences the performance of the list compaction. We first describe the structure of handles and then the compaction mechanism.

In the single-threaded implementation, a handle has a validity state in addition to a reference to an element of the update list. Each element of the list maintains a reference count. During an epoch, if an invalidated handle is accessed, it is updated to reference the head of the update list and the reference count of the head element is incremented. At the end of the epoch, we decrement the count of the element that the handle references and set the handle state to invalid. Due to single-threaded execution, no synchronization is needed for updates to the reference counts.

On the other hand in the multi-threaded implementation, a handle contains an epoch number in addition to a reference to an element of the update list. A unique epoch number is assigned to each message when it is being processed. This epoch number helps discriminate two cases. Upon accessing a handle during processing of a message, the epoch number of the message that is being processed and the epoch number of the accessed handle are compared. If the epoch numbers are different, the handle is being accessed for the first time in the current epoch. Hence, the reference in the handle is updated to the current head of the update list and the epoch number in the handle is updated to the epoch number of the message. If the two epoch numbers are the same, the handle is being accessed again in the current epoch and hence the reference in the handle is not updated.

As an access to the shared abstraction includes a traversal of the update list, the access time is a linear function of the length of the update list plus the access time of the base object. To make the performance of accesses as close as possible to the performance of accesses on the base object, we need to keep the length of the update list small. Consider a reference to an element of the update list and that it divides the list to two sublists that we call the head list and the tail list. The tail list can be compacted as follows. We want to apply the updates of the tail list to the base object in the reverse order of the list and then update the last element of the head list to the base object. See Figure 9.(C) and Figure 9.(D). But consider a tail list and an epoch that is accessing the abstraction using a handle that references the middle of the tail list. To preserve the isolation property, the epoch should not observe the updates that are added to list after the handle of this epoch. If we compact this tail list, its updates are applied to the base object. The base object is accessible starting from the handle too. This can lead the epoch observe the updates that are after the handle and therefore, isolation is violated. To prevent violation of the isolation property, we need to compact a list only if no handle references the middle of it.

In the single-threaded implementation, the list can be compacted efficiently based on reference counting. If the reference count of all elements of a list is zero, it can be merged into the base object. On decrementing a counter at the end of an epoch, if the counter reaches zero, the sublist staring from the element is checked and if all the reference counts in it are zero, it is merged. If the reference count of an element reaches zero, it will not be referenced again. Therefore, checking the reference counts can be optimized by skipping the previously visited sublists. Again, note that sequential processing precludes synchronization.

Unfortunately, compaction based on reference counting introduces contention that limits the scalability of multi-threaded implementations. Instead, we employ a technique inspired by RCU (Read-Copy-Update) [13]. To compact the update list, we first read the current head. The goal is to compact the list that the first element points to which we call the target list. As handles used in processing of new messages are updated to reference the head of the update list, no new epoch will reference an element in the target list. We need to make sure that all the currently active epochs that have a handle to the target list are finished. We schedule a dummy message in each thread of the thread pool that executes the epochs. As threads process messages non-preemptively, the new messages are processed after the currently active epochs. Therefore, once all the threads finish the processing of the dummy messages, all the epochs that were active at the time of scheduling the dummy messages are already finished. At this point, it is certain that no epoch has or will have a handle that references the middle of the target list. Thus, the target list is merged to the base object. While the updates are being applied to the base object, concurrent readers may access the base object. These readers do not experience any inconsistency because the base object is a linearizable implementation of the abstraction and also viewing the updates both in the list and in the base object does not affect the values that the readers read.

A single thread is used for compaction. The compaction procedure does not affect the progress of the actors. The compaction procedure at each host is independent of other hosts and thus there is no global synchronization overhead.

Older elements tend to have no reference sooner so that the list is typically very short. In average, only a small window of active versions is needed to be maintained. We bound the maximum size of the list by a small constant. In pathological cases that the size of the list exceeds the limit, we create a new compacted object that will be used for all the new epochs. This technique mitigates the worst case access time.

An optimization to lower pressure on garbage collector and improve cache locality is to reuse cells of the list that is removed during compaction. A subtle point is that in the compaction, even after updating the pointer to the base object, there may be readers that are still traversing the removed list. Thus, we need to wait for the next RCU cycle before reusing the cells of the old list.

The implementation maintains one writer or multiple concurrent readers of the shared state. Thus, it preserves race-freedom of the encapsulated state. Due to independence of the versioning mechanism in reader hosts, fault isolation is preserved at the granularity of hosts. Both writer and reader actors can move between hosts. Therefore, sharing actors preserves the location transparency property as well.

4. Checking Tool

The translation of sharing programs to pure programs explained in Section 2.6 gives a direct method of implementing the semantics. While this translation is an essential part of the interaction equivalence result, a direct implementation of this translation is inefficient in terms of both time and space. Our optimized implementation benefits from sharing data and applies different techniques to implement the semantics efficiently. Thus, the optimized implementation is quite involved. To increase the confidence that the optimized implementation complies with the semantics, we have built a testing tool.

The testing tool executes a test program with the optimized implementation and logs the execution. It uses a direct

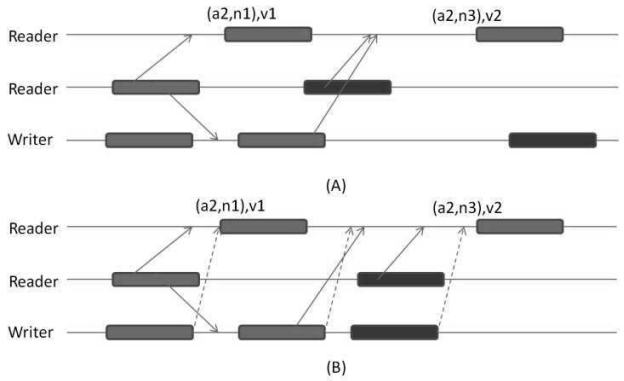

Figure 10. Play and Replay

```
import akka.sharing.checker.core.Actor._
  def main(args: Array[String]) {
    play({
      import akka.sharing.map.optimized.MapImp
//  replay({
//    import akka.sharing.map.reference.MapImp
//  detached({
//    import akka.sharing.map.optimized.MapImp

      val map = new MapImp[Int, Int]()
      val readMap = map.newReadRef()
      val writer = actorOf(new Writer(map))
      val reader = actorOf(new Reader(readMap))
      readMap.setReader(reader)
      writer.start()
      reader.start()
    })
  }
```

Figure 11. Sample Checker Code

implementation of the semantics as the reference implementation. Using the log, the checker replays the execution with the reference implementation and compares the configurations resulted from the two executions. A difference in the results suggests a bug in the optimized implementation. Variability in interleaving of independent actors can expose bugs. If the same procedure is repeated for a large number of times and the resulting configurations are the same, the confidence in the correctness of the implementation is increased.

We assume that the process of each actor in the test programs is a deterministic function of the current state of the actor and the input message. To have a deterministic replay of the execution, we want each actor to process the exact same sequence of messages that it has processed in the play. A unique id is assigned to each actor. The id of each actor is assigned when its parent creates it. The id of a child actor is the id of the parent actor concatenated with a child number. As the execution of each actor is serial, the messages that an actor sends can be serially numbered. The id of a message is the pair of the id of its sender actor and the number of the message in the sender. For each actor, we log an entry for each epoch. The entry is a pair of the processed message id and the version of the shared abstraction with which the processing is started. During the replay, the checker reads the log and makes sure that each actor processes the same sequence of messages with the same version of the shared abstraction. Out of order messages and updates are cached and applied at a later time.

Figure 10 shows an example of the play and replay procedure. The first two actors are readers and the third actor is the writer of the shared abstraction. Each block shows an epoch. Dark blocks show epochs that their execution time is different in the play and the replay. The upper diagram Figure 10.(A) shows a play with the optimized implementation and the lower diagram Figure 10.(B) shows the replay with the reference implementation. The arrows show sending of messages. In the lower diagram, dashed arrows show sending of updates to reader actors in the reference implementation. In the optimized implementation, the reader actors are transparently updated. The first reader actor processes message ((a2, n1), v1) and then message ((a2, n3), v2). This means that the first actor processes the message number 1 of actor 2 with version 1 of the abstraction and then processes the message number 3 of the actor 2 with version 2 of the abstraction. In the replay, the second epoch of the second actor executes later (moved to the right). Now, the first actor receives a message from the third actor before receiving the message of the second actor. Furthermore, the third epoch of the third actor is executed sooner (moved to the left). The first actor has received the third version of the shared abstraction before its second epoch. Despite these reorderings, the checker caches out-of-order messages and makes sure that the first actor processes the message number 3 from actor 2 with version 2 the shared abstraction.

We have written our checking tool on top of Akka[4], an actors library written in Scala[5]. The checker exposes the same interface as the Akka actors interface. Thus, the checker interface can be used during development and a single import statement needs to be changed for deployment. As Figure 11 shows, three modes are possible during the development. The play mode logs the execution, the replay mode replays the log and the detached mode performs an execution without logging. Switching between the play, replay and detached modes are as easy as commenting and uncommenting a couple of lines.

5. Related Works

Erlang's ETS (Erlang Term Storage)[6] tables can be shared among actors. Similar to our shared maps and sets, ETS tables provides atomicity for updates to a single key. On the other hand, they do not provide atomicity for accesses involving multiple keys. Therefore, these tables do not provide isolation. Moreover, they also have no notion of versioning and distribution.

Axum programming language introduced domains as nesting classes that enclose statically declared reader and writer actors. The state of the domain can be accessed by a single writer actor or concurrently by multiple reader actors. De Koster et al. [11] emphasized the need for a sharing mechanism as a compliment to the pure actor model and revisited domains. They defined a domain as an object that actors can have a view to. An actor can asynchronously request shared or exclusive view to a domain at runtime. Once the access right is granted to an actor, it keeps it during the epoch and can synchronously access the objects in the domain. Domains are a proposal for the same problem that this paper tackles. Both models extend the actor programming model with single-writer multiple-reader abstractions. Domains are introduced as an extra design element while sharing actors keep actors as the only one. In sharing actors, shared data appears as traditional encapsulated state. Domains allow two concurrent reads while sharing actors not only allow two concurrent reads but also concurrent write and read due to maintaining multiple

[4] http://akka.io/
[5] http://www.scala-lang.org/
[6] http://www.erlang.org/doc/man/ets.html

versions. Consistency is maintained by locking in domains. On the other hand, it is maintained by multiple versions in sharing actors. Therefore, sharing actors are amenable to distribution across nodes. Domains although preserve many properties are not formalized. The access level to a domain can be decided at runtime. On the other hand, our current sharing actors have fixed ownership.

The idea of accessing a shared object with different views is seen not only for the actors programming model but also in the context of sharing programming models. Demsky and Lam [5] define views as a partial object interface. The user can define several views of an object and their incompatibilities. A Code region declares its views to the objects that it accesses. Using view declarations, data-races can be found and locking implementations can be synthesized.

The pure actor theory requires sequential processing for an actor. Although this simplifies programming and reasoning, it can limit scalability. To allow parallelism inside an actor, Scholliers et al. [21] proposed parallel actor monitors (PAM) and Imam and Sarkar [12] proposed integration of the async-finish model. The former allows parallel processing of messages by a user-defined scheduler object. The latter allows parallel processing of a single message by launching asynchronous light-weight tasks. These works target intra-actor parallelism. On the other, sharing actors allow inter-actor parallel read and write.

Similar to our shared abstractions, a class of STM algorithms maintain multiple versions [18]. On the other hand, they neither directly provide incremental updates for distribution nor exploit the single-writer multiple-reader assumption of our scenario.

6. Conclusion

The proposed sharing actor theory extends the pure actor theory with single-writer multiple-reader sharing of state. It is proved that our added sharing is only semantic sugar for the pure actor model. We described an optimized implementation of the sharing actor theory. We presented a tool that checks the compliance of the implementation with the semantics. We are currently evaluating the performance of sharing against replication and delegation.

7. References

[1] Gul A. Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, Massachusetts, 1986.

[2] Joe Armstrong. Erlang - a survey of the language and its industrial applications. 1996. In Proceedings of The 9th Exhibitions and Symposium on Industrial Applications of Prolog (INAP), 16–18.

[3] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 7 (July 1970), 422-426.

[4] Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin, 2010. Orleans: A Framework for Cloud Computing, no. MSR-TR-2010-159, Microsoft Research.

[5] Brian Demsky and Patrick Lam. 2010. Views: object-inspired concurrency control. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10), Vol. 1. ACM, New York, NY, USA, 395-404.

[6] Sylvain Conchon and Jean-Christophe Filliatre. 2008. Semi-persistent data structures. In Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems (ESOP'08/ETAPS'08), Sophia Drossopoulou (Ed.). Springer-Verlag, Berlin, Heidelberg, 322-336.

[7] Gruber, Olivier and Boyer, Fabienne. 2013. Ownership-based Isolation for Concurrent Actors on Multi-Core Machines. In Proceedings of the 27th European conference on Object-oriented programming (ECOOP'13), 281-301.

[8] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410, 2-3 (February 2009), 202-220.

[9] Philipp Haller and Martin Odersky. 2010. Capabilities for uniqueness and borrowing. In Proceedings of the 24th European conference on Object-oriented programming (ECOOP'10), Theo D'Hondt (Ed.). Springer-Verlag, Berlin, Heidelberg, 354-378.

[10] Carl Hewitt. 1969. PLANNER: a language for proving theorems in robots. In Proceedings of the 1st international joint conference on Artificial intelligence (IJCAI'69). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 295-301.

[11] Joeri De Koster, Tom Van Cutsem, and Theo D'Hondt. 2012. Domains: safe sharing among actors. In Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions (AGERE! '12). ACM, New York, NY, USA, 11-22.

[12] Shams M. Imam and Vivek Sarkar. 2012. Integrating task parallelism with actors. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12). ACM, New York, NY, USA, 753-772.

[13] H. T. Kung and Philip L. Lehman. 1980. Concurrent manipulation of binary search trees. ACM Trans. Database Syst. 5, 3 (September 1980), 354-382.

[14] Edward A. Lee. 2003. Overview of the ptolemy project. Technical Report UCB/ERL M03/25, University of California, Berkeley.

[15] Mohsen Lesani and Antonio Lain. Semantics-preserving Sharing Actors (Appendices) http://www.cs.ucla.edu/~lesani/companion/agere13/

[16] Ian A. Mason and Caroyln L. Talcott. 1999. Actor languages. their syntax, semantics, translation, and equivalence. Theor. Comput. Sci. 220, 2 (June 1999), 409-467.

[17] Stas Negara, Rajesh K. Karmani, and Gul Agha. 2011. Inferring ownership transfer for efficient message passing. In Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11). ACM, New York, NY, USA, 81-90.

[18] Dmitri Perelman, Rui Fan, and Idit Keidar. 2010. On maintaining multiple versions in STM. In Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC '10). ACM, New York, NY, USA, 16-25.

[19] Krishna Sankar. 2009. Programming Erlang - Software for a Concurrent World by Joe Armstrong, Pragmatic Bookshelf, 2007, p. 536. J. Funct. Program. 19, 2 (March 2009), 259-261.

[20] Jan Schafer and Arnd Poetzsch-Heffter. 2010. JCoBox: generalizing active objects to concurrent components. In Proceedings of the 24th European conference on Object-oriented programming (ECOOP'10), Theo D'Hondt (Ed.). Springer-Verlag, Berlin, Heidelberg, 275-299.

[21] C. Scholliers, E. Tanter, and W. De Meuter. Parallel actor monitors. Technical report, 2010. vub-tr-soft-10-05.

[22] Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-Typed Actors for Java. In Proceedings of the 22nd European conference on Object-Oriented Programming (ECOOP '08), Jan Vitek (Ed.). Springer-Verlag, Berlin, Heidelberg, 104-128

[23] Carlos Varela and Gul Agha. 2001. Programming dynamically reconfigurable open systems with SALSA. SIGPLAN Not. 36, 12 (December 2001), 20-34

[24] Robert Virding, Claes Wikström, Mike Williams. 1996. Concurrent Programming in ERLANG (2nd Ed.). Joe Armstrong (Ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK