CSCI-6500 Final Project Proposal
Actor Domain Language Design and Implementation

Wei Huang Ana Milanova Carlos Varela
Email: huangw5@cs.rpi.edu
RIN: 660793992

November 9, 2010

1 Project Description

1.1 Motivation

Concurrent programming is hard. One of the most important reasons is nowadays' popular programming languages are heavily relied on *shared memory* for currency. Although shared memory approach is efficient and easy for transaction control, it also leads to the problem of deadlocks, race conditions, etc. In addition, it does not quite fit for distributed computing and web services, which are inherently concurrent.

Actor Model [1], on the other hand, gives an alternative for concurrency. Actors share nothing, and only communicate via messages. It is good for programmers since they don't need to worry about locks anymore. Also, it is born for concurrency, thus well fitting for Miltie-core processors and distributed computing. However, it is inefficient for sharing nothing if actors all locates in a single machine. A typical example is the *ant colony optimization* simulation, where a large number of autonomous entities have to communicate with a constantly changing environment. On the other hand, coordination and transaction control are not easy as well.

Hence, we propose the *Actor Domain*(AD), a programming language that mixes shared memory concurrency and Actor Model concurrency.

1.2 Actor Domain

In Actor Domain, actors and objects are grouped in different *domains*, in which actors can share these domain-owned objects. Actors from different domains share nothing but can communicate via messages. A typical Actor Domain example is shown in Figure 1, where `d1` and `d2` are domain identifiers, `o1` and `o2` are objects, and `a1` to `a4` are actors. Solid line means directly access by method invocation while dash line means communication via

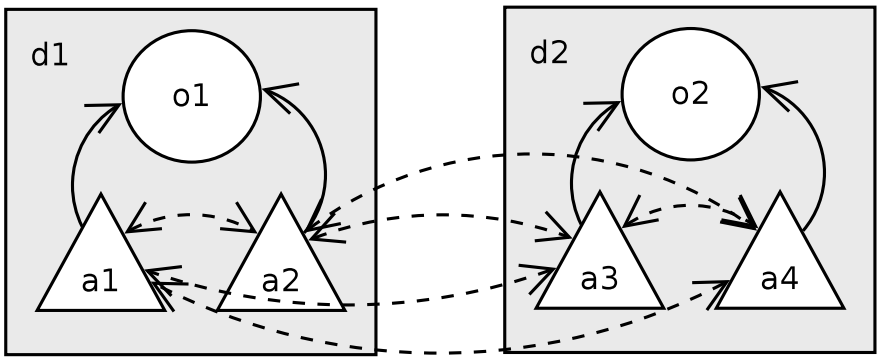

Figure 1: A Typical Actor Domain

message passing. Thus `a1` and `a2` can concurrently access `o1` by using locks like shared memory concurrency, while `a1`,`a2`,`a3`, and `a4` can send each other messages like Actor Model. Note that `o1` and `o2` are completely hidden by `d1` and `d2`, respectively. In general, AD becomes complete shared memory if all actors are created in a single domain, and becomes Actor Model when each domain contains only one actor.

1.3 Example

We present one of our motivation examples, *Inverted Index*, which shows how we can mix shared memory concurrency and Actor Model in AD.

In Actor Model, when input a bunch of documents to index, we can divide these documents into N parts, and assign them to N actors, finally we merge sub-indexes built by N actors. As we can see, the merge operations can be critical to the performance of the whole system. Since actors are sharing nothing, they have to make full copies of their sub-indexes, which will be sent to actors which will conduct the merge operations.

Making full copies is not efficient. In AD, however, we can decrease the number of merge operations by dividing into less parts and assign each part to a domain, where several actors can concurrently read the documents and synchronously updated their shared sub-index. A SALSA-like pseudo code is shown in Figure 2.

At line 6 and 7, two actors are created in their own domains. Thus, they share nothing during the runtime. Then, we create two workers for each `IndexBuilder`in the current domain at line 18 and 19, making them share the same empty index object `sharedIndex`.

When compared to pure Actor Model implementation, although it requires two workers to synchronously access `sharedIndex` inside `IndexBuilder` when building the index, we can save the a number of merge operations and possible deep copy of sub-index built by each actor because of message passing.

On the other hand, it has the advantages of distribution and easy analysis compared to the pure shared memory implementation. Since actors in different domains share nothing, we can even migrate a domain during the execution for load balancing. Also, the number of actors in the same domain can be restricted to the number of processors/cores, facilitating the static analysis of the program and fully utilizing multi-processors/cores

```
1 behavior Main {
2   InvIndex buildIndex(Document[] docs) {
3     // Spilt docs into 2 parts, stored in splited
4     Document[] splited = Helper.split(docs, 4);
5     // Create 2 actors in new domain
6     IndexBuilder b1 = new IndexBuilder();
7     IndexBuilder b2 = new IndexBuilder();
8     token InvIndex i1 = b1!build(splited[0]);
9     token InvIndex i2 = b2!build(splited[1]);
10    // Merge when all builders finished
11    return this!merge(i1, i2);  }
12  InvIndex merge(token InvIndex i1, token InvIndex i2) { ... } }
13
14 behavior IndexBuilder {
15  token InvIndex build(Documents[] docs) {
16    // Create an empty index object
17    InvIndex sharedIndex = new InvIndex();
18    Worker|this.d| w1 = new Worker|this.d|();
19    Worker|this.d| w2 = new Worker|this.d|();
20    // w1 and w2 will build the index respectively and store to sharedIndex
21    Document[] splited = Helper.split(docs,2);
22    token Boolean t1 = w1!build(docs[0], sharedIndex);
23    token Boolean t2 = w2!build(docs[1], sharedIndex);
24    return this!waitAndReturn(t1,t2,sharedIndex);  }
25  InvIndex waitAndReturn(...){...} }
26
27 behavior Worker {
28  Boolean build(Document d, InvIndex index) {...}}
29
30 class Document {...}
31 class InvIndex {...}
```

Figure 2: Inverted Index

machines.

After we implement Actor Domain, we will compare the performance of SALSA and AD with this inverted index example.

1.4 Objectives

There are basic objectives and advanced objectives of this final project. Those basic objectives are:

1. Design the Actor Domain language, including language syntax, type system, and operational semantics
2. Implement the compiler of Actor Domain
3. Implement the runtime of Actor Domain

Advanced objectives include:

1. Implement actor distribution and migration
2. Quantify the performance lost due to the distribution and migration support
3. Implement the Inverted Index example in local setting and compare the performance with SALSA

2 Related Work

Two techniques have been widely used or studied for concurrent programming: *shared memory* and *Actor Model*.

2.1 Shared Memory

Concurrency of today's popular programming languages, like Java, C++, C#, etc. is all relied on shared memory, which leads to the problems of deadlocks, race conditions, and so on. Deadlock *prevention*, *avoidance*, and *detection* have been studied and used for years both in local and distributed environment.

In recent years, researchers have been studying how to protect shared memory by using *type systems*. Clarke etc. propose a static *ownership* type system to limit visibility of object references and restrict access paths to objects, thus controlling a system's dynamic topology [3]. Dietl and Muller give a less restrictive *owner-as-modifier* ownership type system, in which only the owner of an object has the *write* privilege [4]. *Loci* provides a simple type system for thread-local data in Java [10]. It partitions the heap of a program into a number of isolated *heaplets*, each of which is one-to-one mapping to a thread. Hence, the objects in a heaplet are only accessible from its mapping thread, preventing unexpected read/write conflicts for the whole system. [9] presents a type system for data-centric synchronization, where fields of objects are grouped into an atomic set that must be updated atomically.

2.2 Actor Model

Actor Model, proposed by Carl Hewitt etc. in 1973 [6], has been gaining popularity in recent years. Based on message passing and sharing nothing, the Actor Model is inherently concurrent [1]. Erlang [2] is a functional language that follows the Actor Model for concurrency. Scala [5], influenced by Erlang, provides actor libraries for concurrency. Kilim [7] actors use ultra-lightweight threads as well as safe, zero-copy message passing based on a type system. Previous work designed and implemented an actor-based language SALSA [8], which aims at mobile and Internet computing.

3 Paper to Present

I am going to present a paper: actors that unify threads and events [5].

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, 1998.

[4] W. Dietl and P. Mller. Universes: Lightweight ownership for jml. *JOURNAL OF OBJECT TECHNOLOGY*, 4(8):5–32, 2005.

[5] P. Haller and M. Odersky. Actors that unify threads and events. In *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION)*, pages 171–190, 2007.

[6] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[7] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP*, pages 104–128, 2008.

[8] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[9] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP*, pages 304–328, 2010.

[10] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for java. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 445–469, Berlin, Heidelberg, 2009. Springer-Verlag.