Programming Dynamically Reconfigurable Open Systems with SALSA

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

Gul Agha
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave
Urbana, IL 61801, U.S.A.
agha@cs.uiuc.edu

ABSTRACT

Applications running on the Internet, or on limited-resource devices, need to be able to adapt to changes in their execution environment at run-time. Current languages and systems fall short of enabling developers to migrate and reconfigure application sub-components at program-execution time.

In this paper, we describe essential aspects of the design and implementation of SALSA, an actor-based language for mobile and Internet computing. SALSA simplifies programming dynamically reconfigurable, open applications by providing universal names, active objects, and migration. Moreover, SALSA introduces three language mechanisms to help programmers coordinate asynchronous, mobile computations: token-passing continuations, join continuations and first-class continuations.

We provide some examples which illustrate how SALSA programs are not only dynamically reconfigurable and open, but also much more concise and easier to follow than comparable Java code. Furthermore, we provide empirical results which show SALSA's performance to be better than Java code using an actor library, and which illustrate the difference between local, local area, and wide area communication and migration. Finally, we discuss the implementation of our preprocessor which translates SALSA code into Java.[1]

General Terms

Open Systems, Programming Languages, Mobile Computing, Network Computing

Keywords

Actors, SALSA, Java, Internet, Continuations

[1]Available at http://osl.cs.uiuc.edu/salsa/

1. MOTIVATION

Consider a chess program running on the web, playing against Kasparov, Kramnik, or Deep Blue. Participants may start and stop collaborating in this effort with their computing power at any time. The chess program must therefore enable hosts to dynamically *join* and *leave* the global computation of best moves. Furthermore, different parts of the program may need to be relocated at run-time to improve locality – which is affected by the dynamic nature of the underlying network topology.

Consider a second example: a cellular phone or a watch with very strict power, memory and bandwidth limitations running a web browser. The web browser must be able to dynamically reconfigure itself, or to enable manual reconfiguration; for example, moving the HTML parsing and hypertext rendering components to a nearby computation server. When the cellular phone or watch user moves, the computation server must be dynamically changed. When the user arrives to his office or home, the web browser components must be re-composed in the user's desktop.

Internet and mobile computing place new demands on applications, which require them to be open and dynamically reconfigurable. Current programming languages and technologies fall short of enabling developers to write applications with such levels of adaptation, openness and reconfigurability at run-time.

Java, for example, has become widely used, arguably because of its support for dynamic Web content through applets, network class loading, bytecode verification, secure sandbox execution, and multi-platform source and byte code compatibility [19, 27]. A number of important high-level APIs for Internet programming are also available for Java including `java.rmi` with support for remote method invocations, `java.reflection` with support for run-time introspection, `java.net` with support for datagrams, sockets, and `URL`s, and `java.io` with support for object serialization.

All these facilities make Java a very promising technology for Internet and mobile computing. However, higher-level programming languages are required, in order for applications to be reconfigured, migrated to other platforms, and decomposed and re-composed arbitrarily. All of these operations need to be enabled while the applications are running.

SALSA (Simple Actor Language, System and Architecture) is a concurrent object oriented programming language intended to demonstrate how a few language extensions can go a long way in establishing a natural discipline for programming Internet and mobile computing applications. We have followed a pragmatic approach by using Java as the base language and the Internet as the target execution platform. In a spirit similar to C++, which introduced the advantages of object-oriented programming to C programmers, we believe that by developing a conservative extension of Java, we can introduce the benefits of actor-based programming to Java programmers. Our language, SALSA, can be easily pre-processed to Java, and preserves many of Java's useful object-oriented concepts – namely, encapsulation, inheritance, and polymorphism – while at the same time introducing a discipline of concurrent programming using actors over the Internet.

In this paper, we explain the relationship between SALSA modules, behaviors, actors, and messages; and Java packages, classes, objects and methods. Besides supporting the actor model of computation, SALSA's language constructs for concurrency include token-passing continuations, join continuations and first-class continuations. SALSA's integration with the Internet includes primitives for universal naming, remote communication – with the same syntax as local message passing – and migration.

An illustrative example of a SALSA program can be found in Figure 14. It shows a complete SALSA program representing a migrating Internet agent. This agent migrates through different Internet hosts and can receive a `printItinerary()` message from any other SALSA program running on the Internet. As a response to that message the agent prints the list of hosts it has visited so far.

Paper Structure

We first describe the structure of SALSA programs and its support for actor programming. Section 3 introduces token-passing continuations, join continuations and first-class continuations. Section 4 describes the `UniversalActor` implementation for dynamically distributing and moving computations over the Internet. Section 5 contains essential aspects of the language implementation including the SALSA actor library components, and the join continuation code generation algorithm. Finally, we relate SALSA to other programming languages and libraries for Internet and mobile computing.

2. ACTOR ORIENTED PROGRAMMING

The actor model of concurrent computation for distributed systems [1, 21] provides a unit of encapsulation for both state and a thread of control manipulating that state. Communication is purely based on asynchronous message passing. In response to a message (see Figure 1), an actor may:

- **Create new actors**,

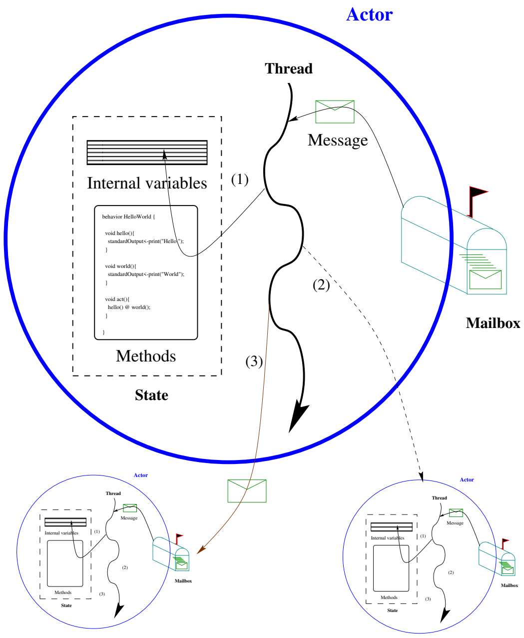

Figure 1: In response to a message, an actor can: (1) modify its local state, or (2) create new actors, or (3) send messages to acquaintances.

- **Send messages to known actors**, possibly including the addresses of other known actors (or *acquaintances*), and/or
- **Modify its own state**, possibly changing its future behavior.

There are no restrictions on the ordering of messages, i.e. an actor may receive messages in a temporal order different than the message sending order. However, the model provides weak fairness, i.e. an actor infinitely ready to receive a message will eventually get it.

2.1 SALSA's Support for Actors

The general architecture for compiling and executing SALSA programs is depicted in Figure 2. A program is transformed into Java source code, using a SALSA preprocessor. The generated Java code uses a library for actors developed especially for SALSA programs. After this step, any Java compiler can be used to produce the bytecodes for the program, which can then be executed on top of any Java virtual machine implementation.

A sample program printing "Hello World!" is shown in Figure 3. When a SALSA program is executed, an actor with the given program behavior is created, and an `act` message is sent to the actor by the run-time system, with any given command line arguments. `standardOutput` is a standard actor provided by the environment. An arrow ($\leftarrow$) indicates message sending to an actor. The at-sign (`@`) indicates a token-passing continuation. In this example, it restricts the

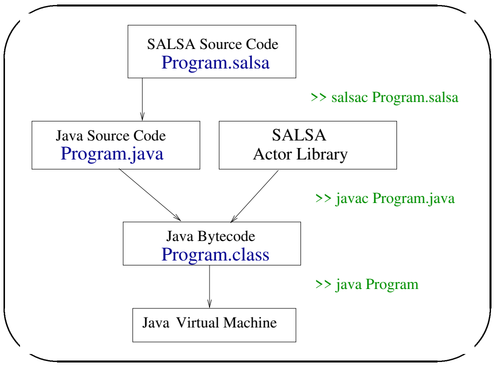

Figure 2: SALSA programs get preprocessed into Java programs which use an actor library, and get eventually compiled into Java bytecodes, for execution atop any Java virtual machine.

```
module helloworld;

behavior HelloWorld {
    void act(String arguments[]){
        standardOutput <- print("Hello ") @
        standardOutput <- print("World!");
    }
}
```

Figure 3: Hello World! Program in SALSA

second message to the `standardOutput` actor to be sent only after the first message has been processed.

SALSA programs are grouped into related actor behaviors, which are called `modules`. A `module` can contain several actor `interfaces` and `behaviors`. A behavior definition may extend another behavior (single inheritance). Every SALSA behavior extends a top-level `UniversalActor` behavior, similarly to Java, in which every class extends a basic `Object` class. A SALSA behavior can implement zero or more interfaces (multiple interface inheritance).

We illustrate how the basic primitives of the Actor model are supported by SALSA.

2.2 Actor Creation

To create a new actor, a SALSA behavior is instantiated and a reference to the new actor is returned. For example, a `HelloWorld` actor with the behavior shown above, is created as follows:

```
HelloWorld helloWorld = new HelloWorld();
```

Behavior definitions may contain `constructors` which support actor initializations with arguments. If the behavior is to be executed as a SALSA program and constructors are provided, they must include a constructor with no arguments for program initialization.

2.3 Message Sending

A SALSA message is implemented as a potential Java method invocation. Actors contain a mailbox which acts as a buffer for messages. Actors process messages sequentially from their local mailbox.

To send a message to an acquaintance actor, SALSA programs use the actor's reference, an arrow (←), the message name and possibly argument values. For example, to send a `print` message with the String argument `"Hello "` to the `standardOutput` actor, the following code is used:

```
standardOutput <- print("Hello ");
```

If an actor does not specify the target for a message, the message is put in its own mailbox. Sending a message to an actor, returns immediately, i.e. it is asynchronous. The value of a message sending expression is `void`. Local parameter passing is by reference, except for Java primitive types, which are passed by value. Remote parameter passing is by reference for universal actors, and by value for serializable Java objects.

2.4 Internal State Changes

An actor changes its state by updating its internal variables through assignments or local method invocations. Notice that only an actor can change its internal state since all variables are private. The only way to change another actor's state is by message passing.

Shared memory is not allowed in SALSA, as opposed to for example, Java `static` variables which are shared among all the instances of a given class.[2] The complete encapsulation of state and processing, afforded by the actor model, enables relatively straightforward actor migration.

3. CONTINUATIONS

In order to coordinate interaction among multiple, autonomous, asynchronous actors, we introduce three kinds of continuations: *token-passing continuations*, *join continuations*, and *first-class continuations*.

3.1 Token-Passing Continuations

A SALSA message is a potential Java method invocation. Since all actor communication is via asynchronous message passing, we need a strategy to "pass" the return values of methods invoked in a given actor (we name these values, `tokens`). To accomplish this, a message to an actor may contain a *customer* actor to which the token should be sent after message processing. SALSA allows the programmer to specify the customer, the message to send to such customer, and the position of the token in the arguments of such customer message. Finally, the continuation itself may have another continuation, thus enabling chains of continuations for tokens.

An example of a token-passing continuation follows:

[2]The current implementation of SALSA (v0.3.2) does not strictly enforce not sharing memory.

```
checking<-getBalance() @ savings<-transfer(token);
```

In this example, the `checking` account actor receives a `getBalance()` message, and when the `checking` account actor is done processing such message, it sends a `transfer` message to the `savings` account actor with the `token`, which is the return value of the original `getBalance()` message.

`token` is a special keyword with the scope provided by the last token-passing continuation. For example, in the expression $a_1 \leftarrow m_1(args)@a_2 \leftarrow m_2(token)@a_3 \leftarrow m_3(token);$, the first *token* refers to the result of evaluating $a_1 \leftarrow m_1$ while the second *token* refers to the result of evaluating $a_2 \leftarrow m_2$.
$a \leftarrow m$ is syntactic sugar for $a \leftarrow m(token)$. For example, the following two lines of code are equivalent:

```
fractal <- computePixel() @ screen <- draw(token);
fractal <- computePixel() @ screen <- draw;
```

When a continuation method is overloaded, SALSA dynamically chooses the most specific method according to the runtime type of the `token` [14, 11].

3.2 Join Continuations

SALSA supports join continuations: a customer actor receives an array with the tokens returned by multiple actors, once they have all finished processing their messages. A sample join continuation follows:

```
join(author1<-writeChapter(1),
     author2<-writeChapter(2)) @
editor<-review @ publisher<-print;
```

In this example, the `editor` actor will receive a `review` message with an array of `chapter` actors as a parameter, and then it will pass it along to the `publisher` for printing. This will happen only after all `author` actors finish processing their respective `writeChapter` messages. The join statement can also receive an array of actors which are all to receive the same message.

3.3 First-Class Continuations

First-class continuations enable actors to *delegate* computation to a third party, independently of the context in which message processing is taking place (i.e. independently of the current continuation for a given message's token).

To illustrate the design, let us look at an example using recursion. The infamous `fibonacci` computation is shown in Figure 4. Notice how the first time that the `compute()` message is sent to the `Fibonacci` actor, the `currentContinuation` is to print the value to standard output. Subsequent recursive `compute()` messages will have as `currentContinuation`, the result of combining the join continuation with the addition message and the previous `currentContinuation` taken from the recursive step.

```
module fibonacci;

behavior Fibonacci {

    int n;

    Fibonacci(int n){
        this.n = n;
    }

    int compute(){
        if (n < 2){
            return n;
        } else {
            Fibonacci fib1 = new Fibonacci(n-1);
            Fibonacci fib2 = new Fibonacci(n-2);
            join (fib1<-compute(), fib2<-compute())
                 @ add @ currentContinuation;
        }
    }

    int add(int numbers[]){
        return numbers[0]+numbers[1];
    }

    void act(String args[]){
        n = Integer.parseInt(args[0]);
        compute() @ standardOutput<-println;
    }
}
```

Figure 4: Fibonacci

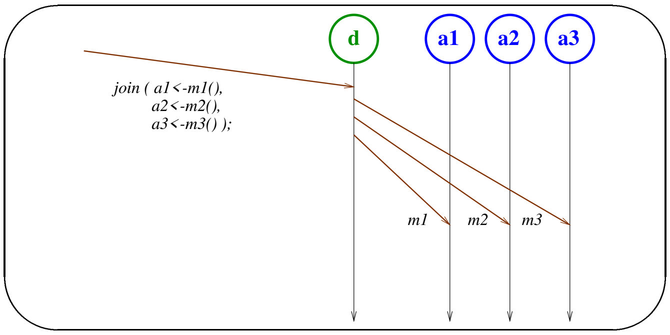

Figure 5: Multicast to three actors `a1`, `a2`, `a3`. The messages `m1`, `m2`, `m3` may be different

3.4 Example: Multicast Protocols

We present three variations of multicast protocols; the presentation serves two goals: first, it illustrates the code generation for join continuations, and second, it allows us to compare the performance and readability of the basic actor operations in SALSA and in Java code. The Java code uses the Actor Foundry [29], a library for implementing actor systems in Java.

We show the trace of a simple multicast protocol in Figure 5. Then, we introduce two variations of an acknowledged multicast protocol. By an acknowledged multicast protocol, we mean that the receipt and processing of a message by different members of an actor group is acknowledged to an outside actor.

```
module multicast;

behavior AcknowledgedMulticast{
    SimpleActor[] actors;
    void done(){}
    void act(String[] args){
        int howMany = Integer.parseInt(args[0]);
        SimpleActor[] actors = new SimpleActor[howMany];
        for (int i = 0; i< howMany; i++){
            actors[i] = new SimpleActor();
        }
        join(actors<-m()) @ done();
    }
}
```

Figure 6: Acknowledged Multicast Protocol

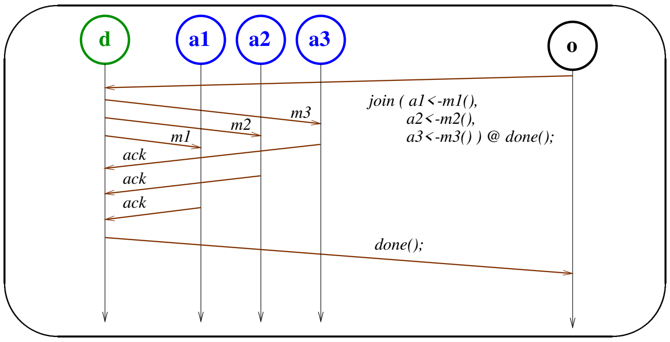

Figure 7: Acknowledged multicast: Actors acknowledge protocol director on message receipt

An implementation of acknowledged multicasting is shown in Figure 6. In this program, we create an array of simple actors – actors with an empty method `m()` – and we use a join continuation to send them all the message `m()`. When the `done()` message is received, all the actors have finished processing the `m()` methods. Figure 7 shows a possible trace of acknowledged multicasting, as generated by SALSA. Vertical lines represent local time (increasing in the downward direction) and diagonal lines represent message traversals between actors. Even though the messages are originally directed to actors `a1,a2,a3`, SALSA-generated code creates a join director, in charge of sending them their respective messages. The join director waits for acknowledgment of message receipt by the coordinated actors, before notifying the outside actor of protocol "finalization".

If we wanted to go further and implement *group knowledge* [15] (i.e. everybody in the group knows that everybody in the group got the message), we could use a two-phase acknowledged multicasting protocol. In the first phase, the director sends a message to every actor in the group; and in the second phase, the director tells every actor in the group that everybody acknowledged message receipt. Figure 8 shows a sample trace.

We have implemented these multicast protocols in SALSA v0.3.2 and the Actor Foundry v0.1.9 [29]. Figures 9 and 10 show the timings for these multicasting protocols. SALSA

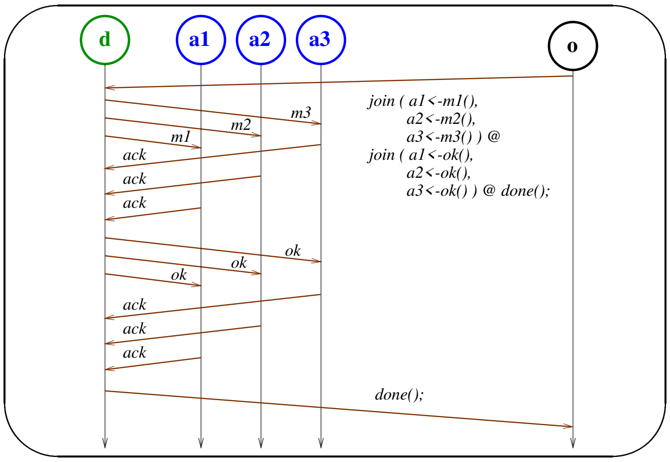

Figure 8: Group knowledge multicast: Actors `a1`, `a2`, and `a3` acknowledge message receipt and `ok` message. The `ok` message from director `d` signals original message acknowledgment by everybody.

code performs an order of magnitude better than Foundry code in Java. There are three reasons for the better performance:

- SALSA's recognition of locality of the participating actors and consequent use of local messages as opposed to reliable UDP.[3]
- The lightweight implementation of actors in SALSA's library is tightly integrated to the code generation process.
- The Actor Foundry's modular architecture enables customization of the transport layer, the request handler, and so forth, which results in a performance penalty.

Perhaps more important than performance is the comparison in the size of these programs. Table 1 shows the number of lines of code in the programs (including the code for time measurements) using the Actor Foundry, using SALSA, and for reference, the lines of Java code generated by the SALSA preprocessor for these examples. Notice how join continuations and token-passing continuations can drastically reduce the size and complexity of concurrent programs: programs are four times smaller on average in these examples.

4. INTERNET COMPUTING

In order to enable actor programs to be open and dynamically reconfigurable, we introduce the concept of a *World-Wide Computer*. The World Wide Computer (WWC) consists of a set of virtual machines, or *Theaters* hosting universal actors. Universal actors are reachable Internet-wide through their globally unique *Universal Actor Names*

[3] Newer versions of the Foundry have since fixed this problem.

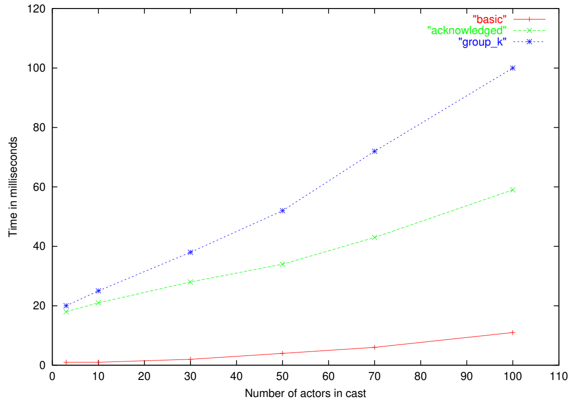

Figure 9: Performance of different multicast protocols (*in ms*) using SALSA 0.3.2

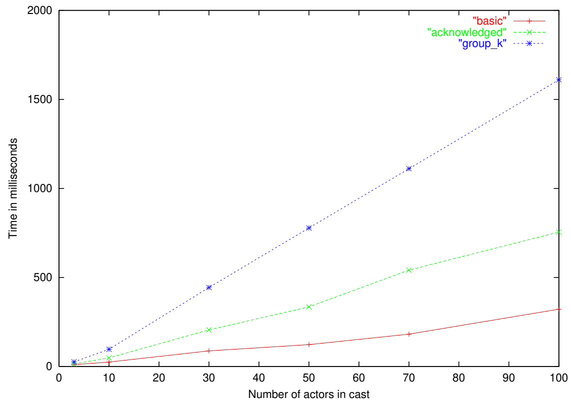

Figure 10: Performance of different multicast protocols (*in ms*) using the Actor Foundry 0.1.9

	Foundry	SALSA	Java
Shared Code	40	10	34
Basic Multicast	146	27	115
Acknowledged Multicast	60	21	134
Group-knowledge Multicast	73	24	183
TOTAL	319	82	466

Table 1: Lines of Code Comparison

```
behavior TravelAgent{

      void printItinerary(){...}

      void act(String[] args){
            TravelAgent a = new TravelAgent();
            try {
                  a<-bind("uan://wwc.travel.com/agent",
                          "rmsp://wwc.aa.com/agent");
            } catch (Exception e){
                  standardError<-println(e);
            }
      }
}
```

Figure 11: Universal Naming and Locator Binding

(UAN) – identifiers which persist over the life time of an actor, and can be used to obtain an authoritative answer regarding the actor's current Internet location, or theater. *Universal Actor Locators* (UAL) uniformly represent the current theater where an actor is running.

The top-level SALSA `UniversalActor` behavior provides methods which support the following Internet computing primitives:

- binding an actor to a name and an initial theater,
- getting references to and communicating with a universal actor, and
- migrating an actor from one theater to another.

SALSA programs can be executed on the WWC infrastructure because all behavior definitions extend the `UniversalActor` behavior.

4.1 Universal Naming

The `UniversalActor` behavior provides support for binding actors to UANs and setting them up in a given theater represented by a UAL. The code for a sample travel agent program is shown in Figure 11. This program creates a travel agent and binds it to a particular UAN (`uan://wwc.travel.com/agent`) and UAL (`rmsp://wwc.aa.com/agent`) pair. The binding process has two effects: (1) the naming server (`wwc.travel.com`) gets updated with the new <relative UAN, UAL> pair (in this case, <`''/agent''`, `''rmsp://wwc.aa.com/agent''`>) and (2) the actor becomes *universal* by migrating to the Theater (`wwc.aa.com`) specified by the UAL. After this program terminates, the actor becomes remotely accessible either by its name or by its locator.

4.2 Remote Communication

SALSA provides support for sending messages to remote actors using the *Remote Message Sending Protocol* (RMSP). SALSA-generated code automatically uses RMSP when actors are remote. SALSA programmers use the same syntax for local and remote message sending. The code for sending a `printItinerary()` message to the travel agent created above, is shown in Figure 12.

```
//
//  Getting a remote actor reference by name
//  and sending a message:
//

TravelAgent a = new TravelAgent();
a<-getReferenceByName("uan://wwc.travel.com/agent") @
  a<-printItinerary();

//
//  Getting the reference by location:
//

TravelAgent a = new TravelAgent();
a<-getReferenceByLocation("rmsp://wwc.aa.com/agent") @
  a<-printItinerary();
```

Figure 12: Actor Location and Remote Communication

```
//
//  Migrating a travel agent to a remote WWC Theater:
//

TravelAgent a = new TravelAgent();
a<-getReferenceByName("uan://wwc.travel.com/agent") @
  a<-migrate("rmsp://wwc.nwa.com/agent");
```

Figure 13: Universal Actor Migration

First, a local reference with the remote actor type (or `UniversalActor`) is created. Then, the reference gets "upgraded" from a local actor reference to a universal actor reference by using the `UniversalActor` behavior method `getReferenceByName(UAN)` or `getReferenceByLocation(UAL)`. Upon completion of this method, the local reference becomes an alias to the remote universal actor. Once we have such an alias, a message can be sent to the actor in the same way as in local message sending. Notice that a token-passing continuation is used to ensure that the reference to the remote actor is obtained before the message is sent to the actor.

4.3 Migration

SALSA provides support for migrating an actor to a given WWC Theater. For example, the code for migrating the travel agent above is shown in Figure 13. Notice that we first use the `getReferenceByName(uan)` method to obtain a reference to the agent and then we use the `migrate(ual)` method to trigger the migration. These two methods are provided by the top-level `UniversalActor` class.

When an actor migrates to a theater: universal actor acquaintances are passed by reference; Java serializable objects and primitive types are passed by value; anonymous actors and non-serializable Java objects generate a run-time exception; and environment actors (like `standardOutput`) are bound after migration to the new theater's environment actors. These environment actors resemble *ubiquitous* types in Sekiguchi and Yonezawa's calculus for code mobility [32].

Because of security considerations, the current implementation requires that code for a universal actor (such as the travel agent) should be present at the receiving theater. However, an alternative is to provide a security manager which makes it possible to download a remote actor and safely execute it. For example, the security manager could sandbox the code of the migrating actor, as is currently done with Java applets. Two additional alternatives to protect the theater hosting environment include authenticating actors (digitally signed code) after migration, and/or verifying that the actor's code respects a security policy (proof-carrying code). Protecting the actor from a malicious theater could be accomplished by requiring the explicit authorization of messages using a hierarchy of directors [36]. *Casts* of actors coordinated by a given director can also be used as a coarser unit of migration, in a spirit similar to Mobile Ambients [10].

4.4 Example: Worldwide Migrating Agent

The worldwide migrating agent example is presented here with two goals: first, to illustrate the high-level support afforded by SALSA programs for actor naming, migration and remote communication, and second, to demonstrate the open nature of SALSA programs due to universal naming. We also use the agent example to time actor communication and migration in different network settings illustrating the wide variety of bandwidth and latency scenarios existing today over the Internet.

Our worldwide migrating agent keeps track of its itinerary and can print it in its current location on request. The SALSA code is shown in Figure 14. The SALSA-generated Java code is about four times larger. Programming this example in pure Java would require much more effort, having to deal directly with naming, serialization and thread migration issues.

This agent program receives a universal name and a list of theaters to visit as command-line arguments. The agent is initially bound to its universal name and home theater. Once the agent has migrated to its home theater it records the local time in that theater. For each new theater in the list, the agents migrates there (updating its itinerary of theaters visited) and prints the current itinerary. If and when the agent revisits its home theater (`initialLocation`) it prints the time which has elapsed since its first visit. Notice that the original initial time travels with the agent as part of its state, but it only makes sense to use it in the agent's home theater.

We have used this example to make a preliminary measurement of performance of different aspects of remote communication and migration in WWC systems. We have used a testbed with three LANs and one WAN, as depicted in Table 2.

Table 3 shows the ranges of values we have measured in our tests with minimal actors and empty messages, as well as with larger actors (up to 100Kb of data).

We have not tried to optimize the current implementation of SALSA programs, the UAN service, or the RMSP server.[4]

[4]To perform these tests, we used SALSA version 0.3.2. The

Machine Name	Location	OS-JVM	Processor
yangtze.cs.uiuc.edu	Urbana, IL, USA	Solaris 2.5.1-JDK 1.1.6	Ultra 2
vulcain.ecoledoc.lip6.fr	Paris, France	Linux 2.2.5-JDK 1.2pre2	PII, 350MHz
solar.isr.co.jp	Tokyo, Japan	Solaris 2.6-JDK 1.1.6	Sparc 20

Table 2: World Wide Computer Testbed

Local actor creation	386μs
Local message sending	148μs
LAN message sending	30-60ms
WAN message sending	2-3 secs
LAN minimal actor migration	150-160ms
LAN 100Kb actor migration	240-250ms
WAN minimal actor migration	3-7secs
WAN 100Kb actor migration	25-30secs

Table 3: Local, LAN and WAN Time Ranges

The main goal of this data is to understand the impact of the different factors in terms of orders of magnitude, not in terms of absolute values. These numbers give us an estimate of the difference in communication and migration times in local, local-area and wide-area scenarios: local times are in microseconds, LAN times are in milliseconds and WAN times are in seconds.

5. LANGUAGE IMPLEMENTATION

SALSA's syntax is based on Java's. The SALSA preprocessor generates Java code, which uses and extends classes in an actor library developed in Java. The generated code can run stand-alone or on the WWC. The WWC run-time system consists of theaters and naming servers distributed on the Internet. The SALSA preprocessor and WWC run-time system are written in Java.[5]

5.1 SALSA Actor Library

Figure 15 shows the structure of the main classes which compose the SALSA actor library. The library contains three packages: `salsa.language`, `wwc.naming` and `wwc.messaging`.[6] In the figure, rectangles represent classes, with the class name in the top center, instance variables below in italics, and method names and argument types in plain style. Solid arrows point to a superclass and curved dotted arrows point to the class of a particular instance variable. We describe the `salsa.language` package in some detail.

The `salsa.language.Actor` class extends the `java.lang.Thread` class and therefore encapsulates a thread of execution directly. An actor contains a `mailbox`, and methods to send it a `Message` object and to process a given message. A main `run` method contains a loop that gets messages from its mailbox and processes them one by one. When a `java.lang.Thread` object is started (i.e. the thread gets actually created), the `run()` method, with the main actor loop gets executed.

The `salsa.language.Mailbox` class contains a `java.util.Vector` of messages, as well as methods to put a message into the mailbox and to get a message from the mailbox (used by the `run` method of the `Actor` class). Access to the mailbox is synchronized using the `lock` associated to the mailbox object.

The `salsa.language.Message` class abstracts over a message being sent from a `source` actor to a `target` actor. It includes a `java.lang.reflect.Method` to be eventually invoked in the target actor with a given array of `arguments`. Additionally, there is an optional token-passing `continuation` which is represented as another `Message` object. If a token is to be passed to such continuation as an argument, a `tokenPosition` value indicates the argument number in which the token should be passed along. Finally, the internal `withMessage` instance variable determines whether or not to pass `this` message as an argument to the original method. This last feature is useful for implementing first-class continuations.

The `salsa.language.UniversalActor` class extends the `Actor` class to make an actor *universal*, i.e. reachable via a Universal Actor Name (`uan`) or Locator (`ual`) by other actors in the World-Wide Computer. The class provides methods to bind a universal actor to a given name and initial theater, to get a reference to a remote universal actor either by its name or by its theater locator, and to migrate a universal actor to a remote theater.

The `salsa.language.JoinDirector` class contains an array of `Message` objects and an array of tokens to be filled with the returned value of the messages. It also contains a `tokensSet` integer which keeps track of how many actors have already returned their tokens and a `continuation` message to be used to notify a customer actor of completion of the join protocol. Section 5.3 describes how this class is used by the SALSA code generator to implement a join continuation statement.

5.2 Runtime System

Theaters are programs which host universal actors and therefore do not explicitly terminate (dæmons). A theater

local times were computed using JDK 1.1.8. on Linux 2.2.16 running on a Pentium III. Local message sending was two times faster than reported above using JDK 1.2 on Windows 2000, and three times faster using JDK 1.3 on Linux 2.2.16.

[5] SALSA Version 0.3.2. contains about 20,000 lines of Java code.

[6] The `wwc` packages were designed and implemented in collaboration with Grégory Haïk at the University of Paris 6.

```
module migration;

import wwc.naming.*;

behavior Agent {

    StringBuffer itinerary;
    UAL initialLocation;
    long initialTime;
    int hops;

    void startTime(){
        initialTime = System.currentTimeMillis();
    }

    void printItinerary(){
        standardOutput<-println(itinerary);
    }

    void addLocation(String ual){
        if (itinerary == null){
            itinerary = new StringBuffer(ual);
            initialLocation = new UAL(ual);
        }
        else {
            hops++;
            itinerary.append(" " + ual);
        }
    }

    void printTime(){
        standardOutput<-println("Migrated "
                                +hops+" times.");
        if (this.getUAL().equals(initialLocation)) {
            standardOutput<-println("Time ellapsed: "
                + new Long(System.currentTimeMillis()
                           -initialTime))@
            standardOutput<-println(
                "Migration avg time: "+ new Long
                ((System.currentTimeMillis()-initialTime)
                 /hops));
        }
    }

    void go(String[] args, Integer iObject){
        int i = Integer.intValue(iObject);
        if (i < args.length){
            addLocation(args[i]) @
                migrate(args[i]) @
                printItinerary() @
                printTime() @
                go(args, new Integer(++i));
        }
    }

    void act(String[] args){
        try {
            addLocation(args[1])@
                bind(args[0], args[1]) @
                startTime()@
                printItinerary() @
                go(args, new Integer(2));
        } catch (Exception e){
            standardOutput<-println(e);
            standardOutput<-println("Usage: "
                +"java migration.Agent UAN UAL (<UAL>)*");
        }
    }
}
```

Figure 14: Worldwide Migrating Agent

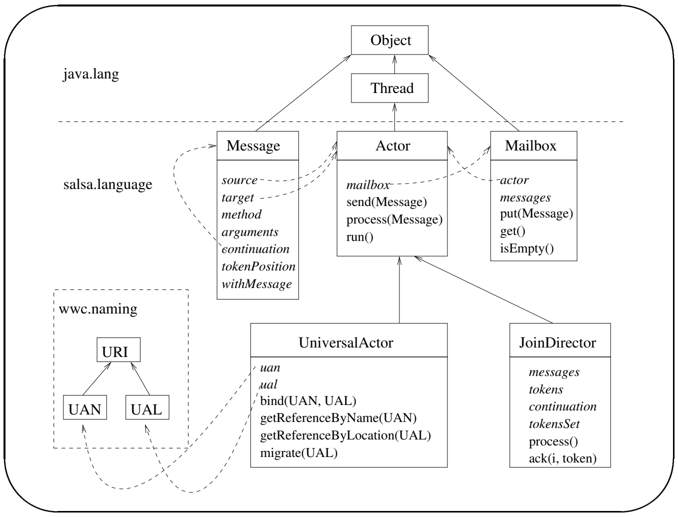

Figure 15: Class hierarchy diagram for the SALSA Actor library. SALSA behaviors get translated into Java classes which extend the `salsa.language.UniversalActor` class.

includes an RMSP server listening for incoming actors and communications. A theater also contains a hashtable mapping relative UALs to internal SALSA actor references. Universal Actor Naming Servers are also dæmons that communicate using the UAN Protocol (UANP), a TCP/IP-based protocol resembling HTTP, which provides methods for setting, getting, updating and deleting <relativeUAN, UAL> pairs.

The runtime system for SALSA programs is in charge of starting program execution by creating an instance of the main behavior and sending an `act(args)` message to such instance with given command-line arguments. The run-time system also needs to check for the conditions that enable program termination:

- All actors are local, i.e. there are no universal actors.
- All actor mailboxes are empty.
- No actor is currently processing a message.

The runtime system accomplishes this by setting up a lower priority thread that checks for these conditions in order to terminate the Java virtual machine.

In the current runtime system implementation, instead of keeping track of all non-universal actors and their mailboxes, every time a message gets sent to an actor, a synchronized global counter gets incremented. Likewise, every time a message has been processed, the counter gets decremented. When a SALSA program starts, the counter is set to 1 (corresponding to the `act(args)` message sent to the main actor). When such counter reaches 0, all messages ever sent in the system have been processed, and there are

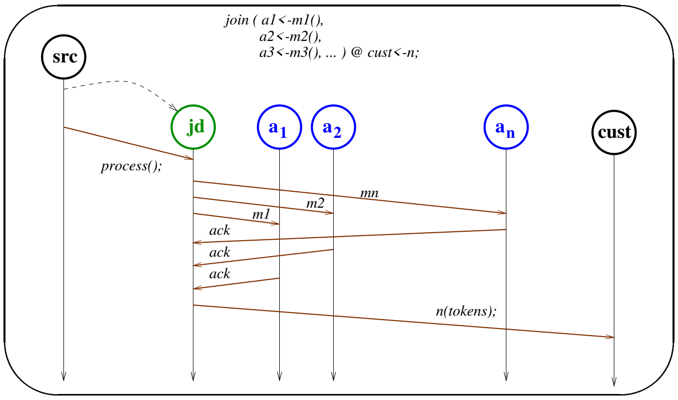

Figure 16: A join statement in SALSA generates code that creates a director actor in charge of executing the specific join protocol with the given participating actors.

no more messages pending, therefore the SALSA runtime system proceeds to terminate the execution of the Java virtual machine.[7]

5.3 Join Continuation Code Generation

The general strategy for implementing the join statement is to create a join director actor which takes care of implementing the join protocol: sending the messages within the join statement to the respective actors, collecting back the tokens, and contacting the customer actor with the compiled array of tokens after completion.

Figure 16 displays a generic join protocol, as implemented by the generated SALSA code for a particular join statement. In this case, the `src` actor executing the join statement will create a join director (an instance of the `salsa.language.JoinDirector` behavior) and send a `process()` message to such director.

In response to the `process()` message, the director sends the original messages to the actors participating in the join protocol, with the token-passing continuation `ack(i, token)`. Every actor contacts back the director, with two pieces of information: first, the position of the actor in the group of participating actors, and second, the token returned after the computation. When the join director has finished receiving all the respective tokens, it sends the customer actor the continuation along with the array of tokens returned, if such tokens are desired by the customer actor.

We have shown only a simple example of a join statement and its corresponding code generation. Note that the code generation algorithm has to deal with the most general case of any potential location for the "join" statement within a SALSA expression. Appendix B shows a more general example of the usage of the `join` statement, along with the Java code generated by SALSA.

[7] James Waldby proposed this idea for JVM termination.

6. DISCUSSION AND RELATED WORK

The separation of state (procedures and data) and processing (threads) into different entities in passive object oriented programming languages, the synchronous blocking interaction among objects, and the difficulties arising from shared memory, especially in distributed settings; motivate us to consider a different model of concurrent and distributed computation for mobile and Internet computing. The actor model is a good candidate because it promotes independence of computing entities to support migration, distribution, dynamic reconfiguration, openness, and efficient parallel execution.

Several libraries which implement the Actor model of computation have been developed in different object-oriented languages. Three examples of these frameworks are the Actor Foundry [29], Actalk [7] and Broadway [33]. Several actor languages have also been proposed and implemented to date, including ABCL [39], Concurrent Aggregates [12], Rosette [34], and Thal [25]. There are several advantages associated with directly using an actor programming language, as compared to using a library for actors:

- **Semantic constraints:** Certain semantic properties can be guaranteed at the language level. For example, an important property is to provide complete encapsulation of data and processing within an actor. Ensuring there is no shared memory or multiple active threads, within an otherwise passive object, is very important to guarantee safety and efficient actor migration.
- **API evolution:** Generating code from an actor language, it is possible to ensure that proper interfaces are always used to create and communicate with actors. In other words, programmers can not incorrectly use the host language. Furthermore, evolutionary changes to an actor API need not affect actor code.
- **Programmability:** Using an actor language improves the readability of programs developed. Often writing actor programs using a framework involves using language level features (e.g. method invocation) to simulate primitive actor operations (e.g. actor creation, message sending, etc). The need for a permanent semantic translation, unnatural for programmers, is a very common source of errors.

Our experience suggests that an active object oriented programming language – one providing encapsulation of state and a thread manipulating that state – is more appropriate than a passive object oriented programming language (even with an actor library) for implementing concurrent and distributed systems to be executed on the Internet.

The ABCL family of languages has been developed by Yonezawa's research group [39] to explore an object-oriented concurrent model of computation, based on Actors. ABCL has been developed in Common Lisp. One significant difference is that the order of messages from one object to another is preserved in their model. There are also three types of message passing mechanisms: past, now, and future. The *past* type of message passing is non-blocking as in actors. The *now* type is a blocking (RPC) message with the sender waiting for a reply. And the *future* type is a non-blocking message with a reply expected in the future. SALSA's more general continuation passing style can be used to implement *now* and *future* message passing.

THAL, an extension to HAL (High-level Actor Language) [23], was developed by Kim [25] to explore compiler optimizations and high performance actor systems. As a high performance implementation, THAL has taken away features from HAL like reflection, and inheritance. THAL provides several communication abstractions including concurrent call/return communication, delegation, broadcast and local synchronization constraints. THAL has shown that with proper compilation techniques, parallel actor programs can run as efficiently as their non-actor counterparts. Future research includes studying optimizations of SALSA actor programs, in particular, the actor model's data encapsulation enables eliminating most of Java's synchronization overhead.

Other researchers are trying to eliminate synchronization redundancies in Java code [31, 16, 13, 4, 6]. There is also a large effort currently taking place to improve the current Java memory model [28, 30, 18], especially in the case of execution on shared memory multiprocessor architectures with weak memory models, such as the SMP Alpha systems. More radical approaches include adding different categories of classes to Java including monitors [22] (e.g. [5]).

Gray et al. present a very complete survey of mobile agent systems [20] categorized by the programming languages they support. Agent systems supporting multiple programming languages include: Ara, D'Agents, and Tacoma. Java-based systems include Aglets [26], Concordia, Jumping Beans, and Voyager. Other systems supporting a non-Java single programming language include: Messengers [17], Obliq, Telescript, and Nomadic Pict [38].

Obliq [8] is a lexically-scoped, untyped, interpreted language, with an implementation relying on Modula 3's network objects. Obliq has higher-order functions, and static scope: closures transmitted over the network retain network links to sites that contain their free (global) variables.

Emerald [24], one of the first systems supporting fine-grained migration, includes different parameter passing styles, namely call by reference, call by move and call by visit. Instance variables can be declared *attached* allowing arbitrary depth traversals in object serialization.

Web combinators [9] are programming constructs introduced to model human behavior in accessing Web documents, taking into consideration transmition rates and failures. Mobile Ambients [10] model object group migration through different administrative domains (e.g. through firewalls).

The authors have proposed the use of *casts*, actor groups coordinated by a *director* actor, as a unit for mobility, security and coordination [36] in large-scale distributed systems. SALSA and the research reported in [35], constitute a first step towards an implementation of these ideas.

SALSA's Relationship to Java and Actors

Because SALSA syntax is similar to Java and because SALSA preprocessors generate Java code for program compilation and execution, it is possible to use Java objects as part of an actor's state. The only condition is that Java code be completely encapsulated within an actor; that is to say, no internal Java threads must ever leave the state of the actor, and thread synchronization must ensure safe shared memory access by multiple internal threads. It is therefore possible to reuse a large number of existing Java libraries within SALSA code. Another inherited advantage from using Java as the base language for implementing actor systems in SALSA is Java's support for inheritance and polymorphism. Actor behavior definitions may specialize other behaviors motivating code reuse. Variables declared to be of a super-behavior type, may actually point to different sub-behaviors at run-time enabling polymorphism and more generic software development.

SALSA differs from pure actor languages [1] in several respects. The `become` primitive has been replaced by a `ready` primitive, signaling that an actor has changed its state and that it is ready to process the next message, following [2]. This is more in spirit with active object oriented programming, where an actor is modeled as an active object and messages are modeled as potential method invocations buffered in a mailbox and processed sequentially in a global loop (implicitly invoking `ready`). More importantly, SALSA's support for token-passing continuations, join continuations, and first-class continuations enables programmers to control concurrency more easily than only using asynchronous message passing. Furthermore, the ability to create universal actor references from URL-like identifiers appears to break the actor acquaintance laws [3] – an actor may only communicate with actors for which it has explicitly received an address. However, actors need to explicitly be made universal, which can be seen as sending the actor's address to a well-known actor (the naming server) which is in turn contacted to get the actor's reference.

7. CONCLUSIONS

We have described the design and implementation of an actor programming language targeting open, dynamically reconfigurable Internet and mobile computing applications. SALSA is a concurrent, dynamically-typed, active object oriented programming language supporting token-passing continuations and join continuations for coordinating actor interactions, and tightly incorporating distributed computation with primitive constructs for Internet-based universal naming, remote communication and migration.

Acknowledgments

John Field, Brent Hailpern, Ganesan Ramalingam, and V. C. Sreedhar at IBM T.J. Watson Research Center provided very helpful comments to improve this paper. Jean-Pierre Briot, Grégory Haïk, and Reza Razavi at Université de Paris

6 and Carolyn Talcott at Stanford University have given us very useful feedback as well. Many ideas presented here are the result of countless discussions in the Open Systems Laboratory at UIUC; in particular, we would like to express our gratitude to Mark Astley, Nadeem Jamali, Yusuke Tada, Prassanna Thati, James Waldby, and Reza Ziaei for helpful discussions about this work. This research was made possible in part by support from the National Science Foundation (NSF CCR 96-19522), the Air Force Office of Scientific Research (AFOSR contract number F49620-97-1-03821), and the Department of Defense Advanced Research Projects Agency (DARPA contract number F30602-00-2-0586).

8. REFERENCES

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

[2] G. Agha. Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems. In E. Najm and J.-B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems IFIP Transactions*. Chapman and Hall, 1997.

[3] 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.

[4] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In A. Cortesi and G. Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 1999.

[5] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. *ACM SIGPLAN Notices*, 35(10):382–400, Oct. 2000.

[6] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 35–46, Denver, CO, Oct. 1999. ACM Press.

[7] J.-P. Briot. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'89)*, pages 109–129. Cambridge University Press, 1989.

[8] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Jan. 1995.

[9] L. Cardelli and R. Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, May/June 1999.

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

[11] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, NY, 1992.

[12] A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. MIT Press, 1993.

[13] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 1–19, N. Y., Nov. 1–5 1999. ACM Press.

[14] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, Oct. 2000.

[15] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[16] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver, British Columbia, June 18–21, 2000.

[17] M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant. Intra- and Inter-Object Coordination with MESSENGERS. In P. Ciancarini and C. Hankin, editors, *First International Conference on Coordination Languages and Models (COORDINATION '96)*, number 1061 in LNCS, Berlin, 1996. Springer-Verlag.

[18] A. Gontmakher and A. Schuster. Characterizations for Java memory behavior. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 682–686, Los Alamitos, Mar. 30–Apr. 3 1998. IEEE Computer Society.

[19] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[20] R. Gray, D. Kotz, G. Cybenko, and D. Rus. Mobile agents: Motivations and state-of-the-art systems. Technical report, Darmouth College, April 2000. Available at ftp://ftp.cs.dartmouth.edu/TR/TR2000-365.ps.Z.

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

[22] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Oct. 1974.

[23] C. Houck and G. Agha. HAL: A high-level actor language and its distributed implementation. In *Proceedings of the 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, IL, August 1992.

[24] E. Jul, H. M. Levy, N. C. Hutchinson, and A. P. Black. Fine-grained mobility in the Emerald system. *TOCS*, 6(1):109–133, 1988.

[25] W. Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1997.

[26] D. Lange and M. Oshima. *Programming and Deploying Mobile Agents with Aglets*. Addison-Wesley, 1998.

[27] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[28] J.-W. Maessen, Arvind, and X. Shen. Improving the Java Memory Model Using CRF. In *Proceedings of OOPSLA*, pages 1–12, 2000.

[29] Open Systems Lab. The Actor Foundry: A Java-based Actor Programming Environment, 1998. Work in Progress. http://osl.cs.uiuc.edu/foundry/.

[30] W. Pugh. Fixing the Java Memory Model. In *Proceedings of ACM Java Grande Conference*, pages 89–98, June 1999. See also JMM mailing list at http://www.cs.umd.edu/ pugh/java/memoryModel/.

[31] E. Ruf. Effective Synchronization Removal for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 208–218, 2000.

[32] T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems, Volume 2*, pages 21–36. Chapman & Hall, 1997.

[33] D. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996. TR UIUCDCS-R-96-1950.

[34] C. Tomlinson, W. Kim, M. Schevel, V. Singh, B. Will, and G. Agha. Rosette: An object oriented concurrent system architecture. *Sigplan Notices*, 24(4):91–93, 1989.

[35] C. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, University of Illinois at Urbana-Champaign, April 2001. http://osl.cs.uiuc.edu/Theses/varela-phd.pdf.

[36] 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.

[37] C. Varela and G. Agha. Linguistic Support for Actors, First-Class Token-Passing Continuations and Join Continuations. Proceedings of the Midwest Society for Programming Languages and Systems Workshop, October 1999. http://osl.cs.uiuc.edu/~cvarela/mspls99/.

[38] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. *IEEE Concurrency*, 8(2), April–June 2000.

[39] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.

APPENDIX

A. GRAMMAR CHANGES TO JAVA

We have based our actor language on Java's syntax and therefore we have used the grammar for Java as a starting point. However, to avoid confusion between SALSA and Java programs; we have changed the name for the most general language constructs: `package` to `module` and `class` to `behavior`. In any case, SALSA modules get preprocessed into Java packages and SALSA behaviors into Java classes which extend a base `UniversalActor` class provided by the SALSA Actor Library. Java `interfaces` remain the same in SALSA.

To support asynchronous message passing as the most basic communication primitive, we have extended Java's `PrimaryExpression` non-terminal from:

```
PrimaryExpression ::=  PrimaryPrefix ( PrimarySuffix )*
```

to:

```
PrimaryExpression ::=  PrimaryPrefix ( PrimarySuffix )*
                       ( MessageSendSuffix )?

MessageSendSuffix ::=  "<-" <IDENTIFIER> ( Arguments )?
```

Note that if the `Arguments` are not given, we know that it must be syntactic sugar for `(token)` within the context of a token-passing continuation.

Token-passing continuations and join continuations are supported in SALSA's grammar with the following rules:

```
ContinuationExpression
  ::=  "currentContinuation"
  |    PrimaryExpression
       ( "@" ContinuationExpression )?
  |    JoinStatement


JoinStatement
  ::=  "join" "(" ContinuationExpressionList ")"
       "@" ContinuationExpression

ContinuationExpressionList
  ::=  ContinuationExpression
       ( "," ContinuationExpression )*
```

Note that it is possible to have token-passing continuation expressions within the list of expressions in a `join` statement.

Finally, two extra possibilities for valid `Statements` were added to the language grammar:

```
Statement ::=  ContinuationExpression ";"
          |    JoinStatement
```

The `token` keyword was also added to the lexical tokens accepted by the parser, and a semantical check at code generation is performed to ensure it is correctly located within an argument list.

```
standardOutput<-print("Time: ") @
join(actors<-m())@ done()@
standardOutput<-print @
standardOutput<-println(" ms.");
```

Figure 17: Join Continuation Code

We have used the JavaCC parser generator and the JJTree abstract syntax tree creation module[8] to create the SALSA preprocessor. We directly manipulate the created abstract syntax tree to generate Java code. To see SALSA's full grammar, we refer the reader to [37].

B. CONTINUATION CODE GENERATION

B.1 Join Continuation Code Generation Example

In this appendix, we display the code generated for the portion of SALSA code in the acknowledged multicast protocol, containing a join continuation.

Figure 17 shows the code in SALSA and Figure 18 shows the Java code generated by the SALSA preprocessor. First, a join director is created (see Figure 16) with the actors participating in the acknowledge multicast protocol. Then the continuation used by the join director is created. This is a token-passing continuation involving `this` actor and the `standardOutput` actor with messages `done, print and println`. Notice how the `done` messages receives no arguments, while the `print` messages receive one argument. Notice also how there is a `token` being passed to the `standardOutput<-print` message, represented by a `0` in the `tokens` array. Once the continuation is created, it is given to the join director – with the `setContinuation` method. So far, there has been no computing, only getting the structures ready. Now, we can proceed to generate a token-passing continuation (represented as a `Message` object) with the `standardOutput` actor and `print(''Time:'')` message, followed by the `joinDirector<-process()` actually performing the join continuation protocol.

B.2 First-Class Continuation Code Generation

First-class continuations can be implemented by generating code that changes the signature of methods to include a formal parameter representing the `Message` instance under which this method is being invoked. Method invocations can then include a reference to the `message` object which contains a reference to the current `continuation`. Currently, first-class continuations have been designed and included in the language grammar. However, the current version of SALSA (v0.3.2) does not yet generate code for first-class continuations.

[8]Available at http://www.metamata.com/

```
JoinDirector _joinDirector1 = null;
  {
  Actor[] _targets1 = actors;
  String[] _methodNames1 = new String[actors.length];
  int[] _tokenPositions1 = new int[actors.length];
  Object [][] _arguments1 = new Object[actors.length][];
    for (int i = 0; i< actors.length; i++){
      _methodNames1[i] = "m";
      _tokenPositions1[i] = -1;
      _arguments1[i] = new Object[0];
    }

    try {
    _joinDirector1 =
      JoinDirector.createJoinDirector
      (this,
        _targets1,
        _methodNames1,
        _arguments1,
        _tokenPositions1,
        null); // to set _continuation1
    } catch (NoSuchMessageException _nsme){
      _nsme.printStackTrace();
    }
    Message _continuation1 = null;
    {
    Actor[] _targets = { this, standardOutput,
                         standardOutput };
    String[] _methodNames = { "done", "print",
                              "println" };
    Object[][] _arguments = { {}, {null}, {" ms."} };
    int [] _tokens = { -1, 0, -1 };
    try {
      _continuation1 =
        (Message.createMessage
        (this,
        _targets,
        _methodNames,
        _arguments,
        _tokens,
        _joinDirector1.getReturnType(),
        0));
    } catch (NoSuchMessageException _nsme){
      _nsme.printStackTrace();
    } catch (MessageCreationException _mce){
      System.err.println("SALSA Internal Error:"+
        _mce.getMessage());
    }
    }
  _joinDirector1.setContinuation(_continuation1);
  }
{
  Actor[] _targets = { standardOutput, _joinDirector1 };
  String[] _methodNames = { "print", "process" };
  Object[][] _arguments = { {"Time: "}, { } };
  int [] _tokens = { -1, -1 };
  try {
    standardOutput.send
      (Message.createMessage
      (this,
      _targets,
      _methodNames,
      _arguments,
      _tokens));
  } catch (NoSuchMessageException _nsme){
    _nsme.printStackTrace();
  } catch (MessageCreationException _mce){
    System.err.println("SALSA Internal Error:"+
      _mce.getMessage());
  }
}
```

Figure 18: Join Continuation Java Code as generated by SALSA preprocessor