THE SALSA PROGRAMMING LANGUAGE
1.1.2 RELEASE TUTORIAL

By

Carlos A. Varela, Gul Agha, Wei-Jen Wang,
Travis Desell, Kaoutar El Maghraoui, Jason LaPorte, and Abe Stephens

Rensselaer Polytechnic Institute
Troy, New York

February 2007

CONTENTS

CHAPTER 1
Introduction

1.1 The SALSA Distributed Programming Language

With the emergence of Internet and mobile computing, a wide range of Internet applications have introduced new demands for openness, portability, highly dynamic reconfiguration, and the ability to adapt quickly to changing execution environments. Current programming languages and systems lack support for dynamic reconfiguration of applications, where application entities get moved to different processing nodes at run-time.

Java has provided support for dynamic web content through applets, network class loading, bytecode verification, security, and multi-platform compatibility. Moreover, Java is a good framework for distributed Internet programming because of its standardized representation of objects and serialization support. Some of the important libraries that provide support for Internet computing are: `java.rmi` for remote method invocation, `java.reflection` for run-time introspection, `java.io` for serialization, and `java.net` for sockets, datagrams, and URLs.

SALSA (Simple Actor Language, System and Architecture) [5] is an actor-oriented programming language designed and implemented to introduce the benefits of the actor model while keeping the advantages of object-oriented programming. Abstractions include active objects, asynchronous message passing, universal naming, migration, and advanced coordination constructs for concurrency. SALSA is pre-processed into Java and preserves many of Java's useful object oriented concepts-mainly, encapsulation, inheritance, and polymorphism. SALSA abstractions enable the development of dynamically reconfigurable applications. A SALSA program consists of universal actors that can be migrated around distributed nodes at run-time.

1.2 Outline

This tutorial covers basic concepts of SALSA and illustrates its concurrency and distribution models through several examples. Chapter 2 introduces the actor model and how SALSA supports it. Chapter 3 introduces concurrent programming in SALSA, including token-passing continuations, join blocks, and first-class continuations. Chapter 4 discusses SALSA's support for distributed computing including asynchronous message sending, universal naming, and migration. Chapter 5 introduces several advanced coordination constructs and how they can be coded in SALSA. Chapter 6 defines actor garbage and explains how automatic actor garbage collection works in SALSA. Appendix A introduces how to specify name server options and how to run applications with different system properties. Appendix B provides debugging tips for SALSA programs. Appendix C provides brief descriptions of SALSA example programs. Appendix D introduces OverView, a toolkit for visualization of distributed systems. Appendix E describes the history of SALSA. Appendix F lists the SALSA grammar.

CHAPTER 2
Actor-Oriented Programming

SALSA is an actor-oriented programming language. This chapter starts first by giving a brief overview of the actor model in Section 2.1. Section 2.2 describes how SALSA supports and extends the actor model.

2.1 The Actor Model

Actors [1, 3] provide a flexible model of concurrency for open distributed systems. Actors can be used to model traditional functional, procedural, or object oriented systems. Actors are independent, concurrent entities that communicate by exchanging messages asynchronously. Each actor encapsulates a state and a thread of control that manipulates this state. In response to a message, an actor may perform one of the following actions (see Figure 2.1):

- Alter its current state, possibly changing its future behavior.
- Send messages to other actors asynchronously.
- Create new actors with a specified behavior.
- Migrate to another computing host.

Actors do not necessarily receive messages in the same order that they are sent. All the received messages are initially buffered in the receiving actor's message box before being processed. Communication between actors is weakly fair: an actor which is repeatedly ready to process messages from its mailbox will eventually process all messages sent to it. An actor can interact with another actor only if it has a reference to it. Actor references are first class entities. They can be passed in messages to allow for arbitrary actor communication topologies. Because actors can create arbitrarily new actors, the model supports unbounded concurrency. Furthermore, because actors only communicate through asynchronous message passing and because there is no shared memory, actor systems are highly reconfigurable.

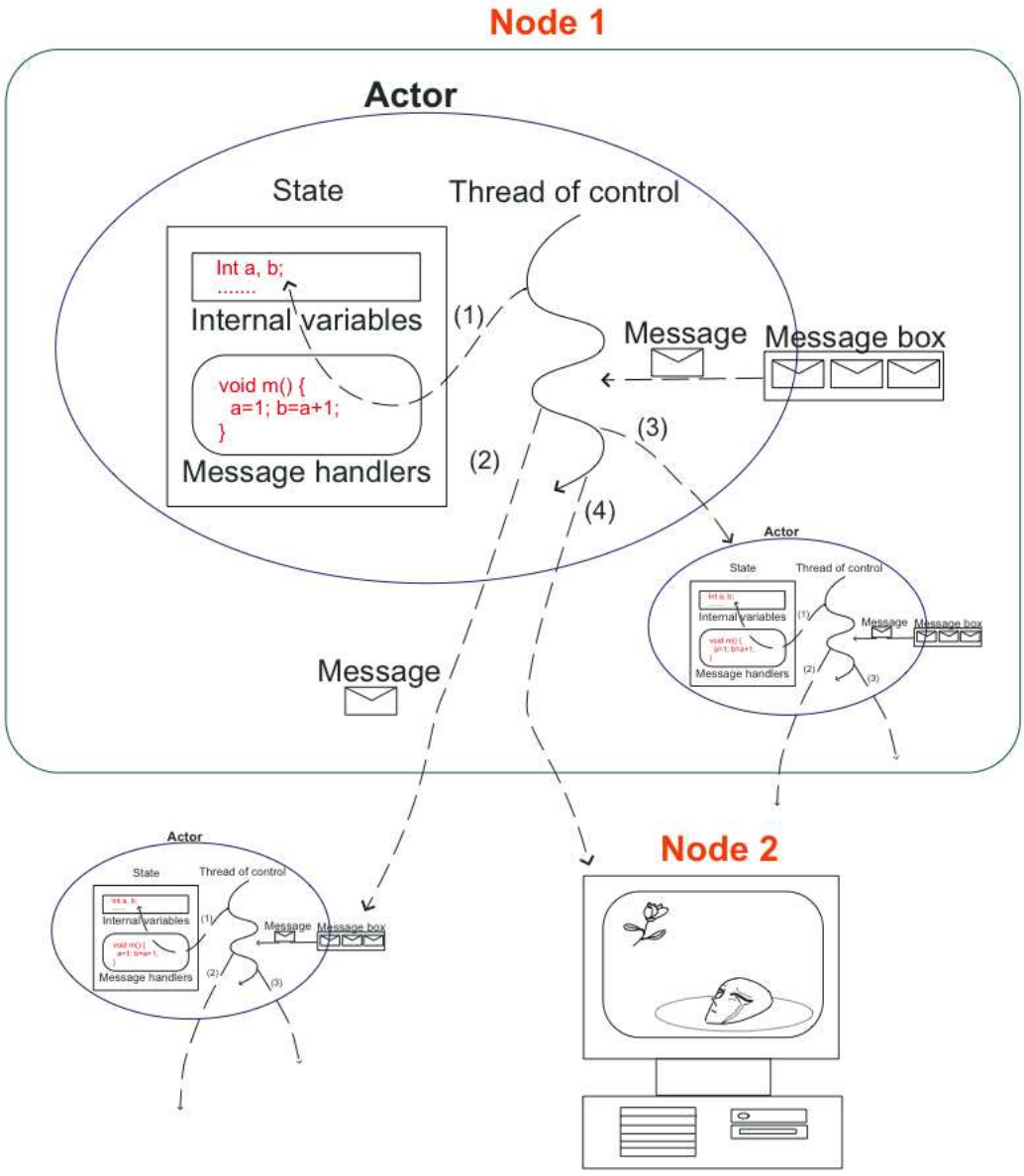

Figure 2.1: Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) send messages to peer actors, (3) create new actors, and/or (4) migrate to another computing host.

2.2 Actors in SALSA

SALSA programmers write behaviors which include encapsulated state and message handlers for actor instances:

- New actors get created in SALSA by instantiating particular behaviors (with the `new` keyword). Creating an actor returns its reference.
- The message sending operator (`<-`) is used to send messages to actors; messages contain a name that refers to the message handler for the message and optionally a list of arguments.
- Actors, once created, process incoming messages, one at a time.

While SALSA supports the actor model, it goes further in providing linguistic abstractions for common coordination patterns in concurrent and distributed applications. For concurrency, it provides token passing continuations, join blocks, first-class continuations, named tokens, and message properties. For distribution, remote actor creation, and remote referencing, it provides universal naming abstractions, location-transparent communication, and migration support. Furthermore, SALSA provides automatic local and distributed garbage collection.

CHAPTER 3
Writing Concurrent Programs

This chapter introduces concepts about basic concurrency coordination. Basic knowledge of Java programming is required.

3.1 Actor State Modification

SALSA is a dialect of Java, and it is intended to reuse as many features of Java as possible. SALSA actors can contain internal state in the form of Java objects or primitive types. However, it is important that this internal state must be completely encapsulated, that is, not shared with other actors. It is also important that the internal state be serializable [1]

The following piece of code illustrates how the internal state is modified, as follows:

```
behavior Cell {
  Object contents;

  Cell(Object initialContents){
      contents = initialContents;
  }

  Object get(){
      return contents;
  }

  void set(Object newContents){
      // update the variable 'contents' with
      // the new value, newContents
      contents = newContents;
  }
}
```

[1] SALSA, as of version 1.1.2, does not enforce object serializability. Programmers must ensure that encapsulated objects are serializable.

3.2 Actor Creation

The actor reference is a new primitive type in SALSA. There are three approaches to obtain an actor reference: either by actor creation statement, the `getReferenceByName()` function, or passed arguments from messages. This section concentrates on actor creation and reference passing.

Writing a constructor in SALSA programming is similar to object construction in Java programming. For instance, one can declare the `HelloWorld` actor as follows:

```
// An actor reference with type HelloWorld
HelloWorld myRef;
```

To create an actor instance and return a reference to `myRef`, one can write code as follows:

```
// Assume the constructor of HelloWorld is:
//    public HelloWorld() {}
myRef = new HelloWorld();
```

In SALSA, actor references are passed by reference, while object variables by value. Objects are passed by value to prevent shared memory and preserve encapsulation.

3.3 Message Passing

SALSA actors use asynchronous message passing as their basic form of communication. A SALSA *message handler* is similar to a Java method. Message passing in SALSA is implemented by asynchronous message delivery with dynamic method invocation. The following example shows how an actor sends a message to itself. Note that it is not a Java method invocation:

```
handler();  // equivalent to "self <- handler();"
```

Another type of message passing statement requires a target (an actor reference), a reserved token `<-`, and a message handler with arguments to be sent. For instance, an actor can send a message to the standardOutput actor as follows:

```
// send a message println() with an argument "Hello World",
// to the actor standardOutput.
standardOutput <- println("Hello World");
```

Note that the following expression is illegal because it is neither a Java method

invocation nor a message passing expression:

```
// Wrong! It does not compile!!!
// Assume 'a' is an actor reference.
a <- someObject.someHandler();
```

Message Passing in SALSA is by-value for objects, and by-reference for actors. Any object passed as an argument is cloned at the moment it is sent, and the cloned object is then sent to the target actor.

3.4 Coordinating Concurrency

SALSA provides three approaches to coordinate the behavior of actors: *token-passing continuations*, *join blocks*, and *first-class continuations*.

3.4.1 Token-Passing Continuations

Token-passing continuations are designed to specify a partial order of message processing. The token '`@`' is used to group messages and assigns the execution order to each of them. For instance, the following example forces the `standardOutput` actor, a predefined system actor for output, to print out "Hello World":

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

If a programmer uses '`;`' instead of '`@`', SALSA does not guarantee that the standardOutput actor will print out "Hello World". It is possible to have the result "WorldHello ". The following example shows the non-deterministic case:

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

A SALSA message handler can return a value, and the value can be accessed through a reserved keyword '`token`', specified in one of the arguments of the next grouped message. For instance, assuming there exists a user-defined message handler, `returnHello()`, which returns a string "Hello". The following example prints out "Hello" to the standard output:

```
// returnHello() is defined as the follows:
//     String returnHello() {return "Hello";}
returnHello() @ standardOutput <- println(token);
```

Again, assuming another user-defined message handler `combineStrings()` accepts two input Strings and returns a combined string of the inputs, the following example prints out "Hello World" to the standard output:

```
// combineStrings() is defined as follows:
// String combineStrings(String str1, String str2)
//   {return str1+str2;}
returnHello() @
combineStrings(token, "␣World") @
standardOutput <- println(token);
```

Note that the first token refers to the return value of `returnHello()`, and the second token refers to that of `combineStrings(token, " World")`.

3.4.2 Join Blocks

The previous sub-section has illustrated how token-passing continuations work in message passing. This sub-section introduces join blocks which can specify a barrier for parallel processing activities and join their results in a subsequent message. A join continuation has a scope (or block) starting with "`join{ `" and ending with "}". Every message inside the block must be executed, and then the continuation message, following `@`, can be sent. For instance, the following example prints either "Hello World SALSA" or "WorldHello SALSA":

```
join {
  standardOutput <- print("Hello␣");
  standardOutput <- print("World");
} @  standardOutput <- println("␣SALSA");
```

Using the return token of the join block will be explained in Chapter 5.

3.4.3 First-Class Continuations

The purpose of first-class continuations is to delegate computation to a third party, enabling dynamic replacement or expansion of messages grouped by token-passing continuations. First-class continuations are very useful for writing recursive code. In SALSA, the keyword `currentContinuation` is reserved for first-class continuations. To explain the effect of first-class continuations, we use two examples to show the difference. In the first example, statement 1 prints out "Hello World SALSA":

```
//The first example of using First-Class Continuations
...
void saySomething1() {
  standardOutput <- print("Hello ") @
  standardOutput <- print("World ") @
  currentContinuation;
}
....
//statement 1 in some method.
saySomething1() @  standardOutput <- print("SALSA");
```

In the following (the second) example, statement 2 may generate a different result from statement 1. It prints out either "Hello World SALSA", or "SALSAHello World ".

```
// The second example - without a First-Class Continuation
// Statement 2 may produce a different result from
// that of Statement 1.
...
void saySomething2() {
  standardOutput <- print("Hello ") @
  standardOutput <- print("World ");
}
....
//statement 2 inside some method:
saySomething2() @  standardOutput <- print("SALSA") ;
```

The keyword `currentContinuation` has another impact on message passing — the control of execution returns immediately after processing it. Any code after it will not be reached. For instance, the following piece of code always prints out "Hello World", but "SALSA" never gets printed:

```
// The third example - with a First-Class Continuation
// One should see "Hello World" in the standard output
// after statement 3 is executed.
...
void saySomething3() {
  boolean alwaysTrue=true;
  if (alwaysTrue) {
    standardOutput <- print("Hello ") @
    standardOutput <- print("World ") @
    currentContinuation;
  }
  standardOutput<-println("SALSA");
}
```

```
....
//statement 3 inside some method:
saySomething3() @ standardOutput <- println() ;
```

3.5 Using Input/Output (I/O) Actors

SALSA provides three actors supporting asynchronous I/O. One is an input service (`standardInput`), and the other two are output services (`standardOutput` and `standardError`). Since they are actors, they are used with message passing.

`standardOutput` provides the following message handlers:

- `print(boolean p)`
- `print(byte p)`
- `print(char p)`
- `print(double p)`
- `print(float p)`
- `print(int p)`
- `print(long p)`
- `print(Object p)`
- `print(short p)`
- `println(boolean p)`
- `println(byte p)`
- `println(char p)`
- `println(double p)`
- `println(float p)`
- `println(int p)`
- `println(long p)`
- `println(Object p)`
- `println(short p)`

- `println()`

`standardError` provides the following message handlers:

- `print(boolean p)`
- `print(byte p)`
- `print(char p)`
- `print(double p)`
- `print(float p)`
- `print(int p)`
- `print(long p)`
- `print(Object p)`
- `print(short p)`
- `println(boolean p)`
- `println(byte p)`
- `println(char p)`
- `println(double p)`
- `println(float p)`
- `println(int p)`
- `println(long p)`
- `println(Object p)`
- `println(short p)`
- `println()`

`standardInput` provides only one message handler in current SALSA release:

- `String readLine()`

Table 3.1: Steps to Compile and Execute a SALSA Program.

Step	What To Do	Action Taken
1	Create a SALSA program: Program.salsa	Write your SALSA code
2	Use the SALSA compiler to generate a Java source file: Program.java	java salsac.SalsaCompiler Program.salsa
3	Use a Java compiler to generate the Java bytecode: Program.class	javac Program.java
4	Run your program using the Java Virtual Machine	java Program

3.6 Developing SALSA Programs

This section demonstrates how to write, compile, and execute SALSA programs.

3.6.1 Writing SALSA Programs

SALSA abstracts away many of the difficulties involved in developing distributed open systems. SALSA programs are preprocessed into Java source code. The generated Java code uses a library that supports all the actor's primitives — mainly creation, migration, and communication. Any Java compiler can then be used to convert the generated code into Java bytecode ready to be executed on any virtual machine implementation (see Table 3.1).

3.6.2 `HelloWorld` example

The following piece of code is the SALSA version of `HelloWorld` program:

```
1.  /* HelloWorld.salsa */
2.  module examples;
3.  behavior HelloWorld {
4.    void act( String[] arguments ) {
5.      standardOutput<-print( "Hello" )@
6.      standardOutput<-print( "World!" );
7.    }
8.  }
```

Let us go step by step through the code of the `HelloWorld.salsa` program:

The first line is a comment. SALSA syntax is very similar to Java and you will notice it uses the style of Java programming. The module keyword is similar to the package keyword in Java. A `module` is a collection of related actor behaviors. A `module` can group several actor interfaces and behaviors. Line 4 starts the definition of the `act` message handler. In fact, every SALSA application must contain the following signature if it does have an `act` message handler:

```
void act( String[] arguments )
```

When a SALSA application is executed, an actor with the specified behavior is created and an `act` message is sent to it by the run-time environment. The `act` message is used as a bootstrapping mechanism for SALSA programs. It is analogous to the Java `main` method invocation.

In lines 5 and 6, two messages are sent to the `standardOutput` actor. The arrow (`<-`) indicates message sending to an actor (in this case, the `standardOutput` actor). To guarantee that the messages are received in the same order they were sent, the `@` sign is used to enforce the second message to be sent only after the first message has been processed. This is referred to as *a token-passing continuation* (see Section 3.4.1).

3.6.3 Compiling and Running `HelloWorld`

- Download the latest version of SALSA. You will find the latest release in this URL: http://wcl.cs.rpi.edu/salsa/

- Create a directory called `examples` and save the `HelloWorld.salsa` program inside it. You can use any simple text editor or Java editor to write your SALSA programs. SALSA modules are similar to Java packages. This means you have to follow the same directory structure conventions when working with modules as you do when working with packages in Java.

- Compile the SALSA source file into a Java source file using the SALSA compiler. It is recommended to include the SALSA JAR file in your class path. Alternatively you can use `-cp` to specify its path in the command line. If you

are using MS Windows use semi-colon (`;`) as a class path delimiter, if you are using just about anything else, use colon (`:`). For example:

java -cp salsa<version>.jar:. salsac.SalsaCompiler examples/*.salsa

- Use any Java compiler to compile the generated Java file. Make sure to specify the SALSA class path using `-classpath` if you have not included it already in your path:

  javac -classpath salsa<version>.jar:. examples/*.java

- Execute your program:

  java -cp salsa<version>.jar:. examples.HelloWorld

CHAPTER 4
Writing Distributed Programs

Distributed SALSA programming involves universal naming, theaters, service actors, migration, and concurrency control. This chapter introduces how to write and run a distributed SALSA program.

4.1 Worldwide Computing Model

Worldwide computing is an emerging discipline with the goal of turning the Internet into a unified distributed computing infrastructure. Worldwide computing tries to harness underutilized resources in the Internet by providing various Internet users a unified interface that allows them to distribute their computation in a global fashion without having to worry about where resources are located and what platforms are being used. Worldwide computing is based on the actor model of concurrent computation and implements several strategies for distributed computation such as universal naming, message passing, and migration. This section introduces the worldwide computing model and how it is supported by SALSA.

4.1.1 The World-Wide Computer (WWC) Architecture

The World-Wide Computer (WWC) is a set of virtual machines, or *theaters* that host one to many concurrently running universal actors. Theaters provide a layer beneath actors for message passing, remote communication, and migration. Every theater consists of a RMSP (Remote Message Sending Protocol) server, a local cache that maps between actors' names and their current locations, a registry that maps local actor names to their references, and a run-time environment. The RMSP server listens for incoming requests from remote actors and starts multiple threads to handle incoming requests simultaneously.

The WWC consists of the following key components:

- Universal naming service

Table 4.1: UAL and UAN Format.

Type	Example
URL	http://wcl.cs.rpi.edu/salsa/
UAN	uan://io.wcl.cs.rpi.edu:3000/myName
UAL	rmsp://io.wcl.cs.rpi.edu:4000/myLocator

- Run-time environment
- Remote communication protocol
- Migration support
- Actor garbage collection

4.1.2 Universal Naming

Universal naming allows actors to become *universal actors*. Universal actors have the ability to migrate, while anonymous actors do not. *Service actors* are special universal actors, with universal access privileges that do not get collected by the actor garbage collection mechanism.

Every universal actor has a *Universal Actor Name (UAN)*, and a *Universal Actor Locator (UAL)*. The UAN is a unique name that identifies the actor throughout its lifetime. The UAL represents the location where the actor is currently running. While the UAN never changes throughout the lifetime of a universal actor, its UAL changes as it migrates from one location to another. UANs and UALs follow the URI syntax. They are similar in format to a URL (see Table 4.1).

4.1.2.1 Universal Actor Names (UANs)

The first item of the UAN specifies the name of the protocol used; the second item specifies the name and port number of the machine where the Naming Server resides. This name is usually a name that can be decoded by a domain name server. You can also use the IP of the machine, but it should be avoided. The last item specifies the relative name of the actor. If a port number is not specified, the default port number (3030) for the name server is used.

4.1.2.2 Universal Actor Locators (UALs)

The first item specifies the protocol used for remote message sending. The second item indicates the theater's machine name and port number. If a port number is not specified, the default port number (4040) for the theater is used. The last part specifies the relative locator name of the actor in the given theater.

SALSA's universal naming scheme has been designed in such a way to satisfy the following requirements:

- Platform independence: names appear coherent on all nodes independent of the underlying architecture.
- Scalability of name space management
- Transparent actor migration
- Openness by allowing unanticipated actor reference creation and protocols that provide access through names
- Both human and computer readability.

4.2 WWC Implementation in SALSA

This section demonstrates how to write a distributed SALSA program and run it in the World-Wide Computer run-time environment.

4.2.1 Universal Actor Creation

A universal actor can be created at any desired theater by specifying its UAN and UAL [2]. For instance, one can create a universal actor at current host as follows:

```
HelloWorld helloWorld = new HelloWorld()
  at (new UAN("uan://nameserver/id"));
```

A universal actor can be created at a remote theater, hosted at host1:4040, by the following statement:

```
HelloWorld helloWorld = new HelloWorld()
  at (new UAN("uan://nameserver/id"),
      new UAL("rmsp://host1:4040/id"));
```

[2]Remember to start the naming server if using UANs in the computation.

An anonymous actor can be created as follows:

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

Notice that an anonymous actor cannot migrate.

4.2.2 Referencing actors

Actor references can be used as the target of message sending expressions or as arguments of messages. There are three ways to get an actor reference. Two of them, the return value of actor creation and references from messages, are available in both distributed and concurrent SALSA programming. The last one, `getReferenceByName()`, is an explicit approach that translates a string representing a UAN into a reference. In SALSA, only references to service actors (see Section 4.2.4) should be obtained using this function. Otherwise, SALSA does not guarantee the safety property of actor garbage collection, which means one can get a phantom reference (a reference pointing to nothing). The way to get a reference by `getReferenceByName()` is shown as follows:

```
AddressBook remoteService= (AddressBook)
  AddressBook.getReferenceByName("uan://nameserver1/id");
```

Sometimes an actor wants to know its name or location. An actor can get its UAL (location) by the function `getUAL()` and UAN (universal name) by `getUAN()`. For example:

```
UAL selfUAL= this.getUAL();
UAN selfUAN = this.getUAN();
```

4.2.3 Migration

As mentioned before, only universal actors can migrate. Sending the message `migrate(<ual>)` to an universal actor causes it to migrate seamlessly to the designated location. Its UAL will be changed and the naming service will be notified to update its entry.

The following example defines the behavior `MigrateSelf`, that migrates the `MigrateSelf` actor to location UAL1 and then to UAL2. The `migrate` message takes as argument a string specifying the target UAL or it can take the object `new UAL("UAL string")`.

```
module examples;

behavior MigrateSelf {
  void act(String args[]){
    if (args.length != 2) {
      standardOutput<-println(
      "Usage:" +
      "java -Duan=<UAN> examples.MigrateSelf <UAL1> <UAL2>");
      return;
    }
    self<-migrate(args[0]) @
    self<-migrate(args[1]);
  }
}
```

4.2.4 Actors as Network Service

There are many kinds of practical distributed applications: some are designed for scientific computation, which may produce a lot of temporary actors for parallel processing; some are developed for network services, such as a web server, a web search engine, etc. Useless actors should be reclaimed for memory reuse, while service-oriented actors must remain available under any circumstance.

The most important reason for reclamation of useless actors is to avoid memory leakage. For example, after running the `HelloWorld` actor (shown in Section 3.6) in the World-Wide Computer, the World-Wide Computer must be able to reclaim this actor after it prints out "Hello World". Reclamation of actors is formally named *actor garbage collection*.

Reclamation of useless actors introduces a new problem: how to support non-collectable service-oriented actors at the language level. This is important because a service-oriented actor cannot be reclaimed even if it is idle. For instance, a web service should always wait for requests. Reclamation of an idle service is wrong.

Services written in the C or Java programming languages use infinite loops to listen for requests. A SALSA service cannot use this approach because loops inside a message handler preclude an actor from executing messages in its message box. The way SALSA keeps a service actor alive is by specifying it at the language level - a SALSA *service actor* must implement the interface `ActorService` to tell the

actor garbage collector not to collect it.

The following example illustrates how a service actor is implemented in SALSA. The example implements a simple address book service. The `AddressBook` actor provides the functionality of creating new <name, email> entities, and responding to end users' requests. The example defines the `addUser` message handler which adds new entries in the database. The example also defines the `getEmail` message handler which returns an email string providing the user name.

```
module examples;

import   java.util.Hashtable;
import   java.util.Enumeration;

behavior AddressBook implements ActorService{
  private Hashtable name2email;

  AddressBook() {
   // Create a new hashtable to store name & email
   name2email = new Hashtable();
  }

  // Get the email of this person
  String getEmail(String name) {
    if (name2email.containsKey(name)) {
      // If name exists
      return (String) name2email.get(name);
    } else {
      return new String("Unknown user");
    }
  }

  // Add a new user to the system, returns success
  boolean addUser (String name, String email) {
    // Is the user already listed?
    if (name2email.containsKey(name) ||
        name2email.contains(email)) {
      return false;
    }
    // Add to our hash table
    name2email.put(name, email);
    return true;
  }

  void act(String args[]) {
```

```
    if (args.length > 0) {
      standardOutput<-println(
      "Usage:" +
      "java␣-Duan=<UAN>␣-Dual=<UAL>␣examples.AddressBook" );
      return;
    }
    else {
      standardOutput<-println("AddressBook␣at:␣") @
      standardOutput<-println("uan:␣" + this.getUAN()) @
      standardOutput<-println("ual:␣" + this.getUAL());
    }
  }
}
```

The `AddressBook` actor is bound to the UAN and UAL pair specified in the command line. This will result in placing the `AddressBook` actor in the designated location and notifying the naming service.

To be able to contact the `AddressBook` actor, a client actor first needs to get the remote reference of the service. The only way to get the reference is by the message handler `getReferenceByName()`. The example we are going to demonstrate is the `AddUser` actor, which communicates with the `AddressBook` actor to add new entries. Note that the `AddUser` actor can be started anywhere on the Internet.

```
module examples;

behavior AddUser {
  void act(String args[]) {
    try{
      if (args.length == 3 ) {
        AddressBook book =
        (AddressBook)AddressBook.getReferenceByName(
          new UAN(args[0]) );
        book<-addUser(args[1], args[2]);
        return;
      }
    } catch (Exception e) {standardError<-println(e);}
    standardError<-println(
      "Usage:" +
      "java␣examples.AddUser␣<bookUAN>␣<Name>␣<Email>" );
  }
}
```

4.3 Run-time Support for WWC Application

The section demonstrates how to start the naming service and theaters.

4.3.1 Universal Naming Service

The UANP is a protocol that defines how to interact with the WWC naming service. Similar to HTTP, UANP is text-based and defines methods that allow lookup, updates, and deletions of actors' names. UANP operates over TCP connections, usually the port 3030.

Every theater maintains a local registry where actors' locations are cached for faster future access. One can start a naming service as follows:

java -cp salsa<version>.jar:. wwc.naming.WWCNamingServer

The above command starts a naming service on the default port 3030. You can specify another port as follows:

java -cp salsa<version>.jar:. wwc.naming.WWCNamingServer -p 1256

4.3.2 Theaters

One can start a theater as follows:

java -cp salsa<version>.jar:. wwc.messaging.Theater

The above command starts a theater on the default RMSP port 4040. You can specify another port as follows:

java -cp salsa<version>.jar:. wwc.messaging.Theater 4060

4.3.3 Running an Application

Whenever a WWC application is executed, a theater is dynamically created to host the bootstrapping actor of the application and a random port is assigned to the dynamically created theater. A dynamically created theater will be destroyed if no application actor is hosted at it and no incoming message will be delivered to the theater.

Now let us consider a WWC application example. Assuming a theater is running at `host1:4040`, and a naming service at `host2:5555`. One can run the `HelloWorld` example shown in Section 3.6 at `host1:4040` as follows:

java -cp salsa<version>.jar:. -Duan=uan://host2:5555/myhelloworld -Dual=rmsp://host1:4040/myaddr examples.HelloWorld

As one can see, the standard output of host1 displays "Hello World". One may also find that the application does not terminate. In fact, the reason for non-termination at the original host is that the application creates a theater and the theater joins the World-Wide Computer environment. Formally speaking, the application does terminate but the host to begin with becomes a part of the World-Wide Computer.

CHAPTER 5
Advanced Concurrency Coordination

This chapter introduces advanced constructs for coordination by using named tokens, join block continuations, and message properties.

5.1 Named Tokens

Chapter 3 has introduced token-passing continuations with the reserved keyword `token`. In this section, we will focus on the other type of continuations, the *named tokens*.

> *Warning:*
> *Tokens can ONLY be used as arguments to messages. SALSA 1.1.2 does not allow to use tokens as part of expressions (e.g., x+2) or to be used as return values (e.g., return token).*

In SALSA, the return value of an asynchronous message can be declared as a variable of type `token`. The variable is called a *named token*. Named tokens are designed to allow more expressibility by allowing explicit definition of continuations. For example, a token-passing continuation statement can be re-written by name token continuations:

```
//line 1 is equivalent to lines 2-3
1. hello() @ standardOutput<-print(token);

2. token x = hello();
3. standardOutput<-print(x);
```

Name tokens can be used to construct a non-linear partial order for computation, which cannot be expressed by token-passing continuations. The following example cannot be re-written by token-passing continuations:

```
token x = a<-m();
token y = b<-o();
token z = c<-p();

d<-p(x,y);
```

```
e<-q(y,z);
f<-r(x,z);
```

The following example uses name tokens to implement the Fibonacci number application:

```
module examples;

behavior Fibonacci {
  int n;

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

  int add(int x, int y) {return x + y;}

  int compute() {
    if (n == 0)  return 0;
    else if (n <= 2)  return 1;
    else {
      Fibonacci fib1 = new Fibonacci(n-1);
      Fibonacci fib2 = new Fibonacci(n-2);
      token x = fib1<-compute();
      token y = fib2<-compute();
      //using name tokens and first-class continuations
      add(x, y) @ currentContinuation;
    }
  }

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

Named tokens may be assigned to non-primitive type values, message sending expressions, or other named tokens. Examples are shown as follows:

```
1. token y  = a<-m1();
2.
3. token z = y;
4.
5. y = b<-m2(y);
6. self<-m() @ c<-m3(token, z, y);
```

The following example shows how to use named tokens. Lines 1-2 are equivalent to lines 3-5 using fewer token declarations:

```
//lines 1-2 are equivalent to lines 3-5
1. token x  = a<-m1();
2. x = b<-m2(x);

3. token x  = a<-m1();
4. token y  = b<-m2(x);
5. x = y;
```

The following example demonstrates how named tokens are used in loops:

```
1. token x  = a<-m1();
2. for (int i = 0; i < 10; i++) x = b<-m2(x, i);
```

The previous example is equivalent to the following example:

```
a<-m1() @
b<-m2(token, 0) @
b<-m2(token, 1) @
b<-m2(token, 2) @
b<-m2(token, 3) @
b<-m2(token, 4) @
b<-m2(token, 5) @
b<-m2(token, 6) @
b<-m2(token, 7) @
b<-m2(token, 8) @
token x = b<-m2(token, 9);
```

To learn more about named tokens, we use the following example to illustrate how the named token declaration works and to prevent confusion:

```
1.    token x  = a<-m1();
2.
3.    for (int j = 0; j < 10; j++) {
4.        b<-m2(x);
5.        x = c<-m3(x);
6.        d<-m4(x);
7.    }
```

The token is updated as soon as the code is processed. In the `for` loop on lines 3-7, for each iteration of the loop, the value of `token x` in `b<-m2` and `c<-m3` is the same. However, the value of `token x` in `d<-m4` is the token returned by `c<-m3`, and thus equal to the value of `token x` in the message sends on lines 4 and 5 in the next iteration of the loop.

5.2 Join Block Continuations

Chapter 3 skips some details of join blocks. This section introduces how to use the return values of statements inside a join block and by implementing message handlers which can receive the result of a join block.

A join block always returns *an object array* if it joins several messages to a reserved keyword `token`, or a named token. If those message handlers to be joined do not return (`void` type return), or the return values are ignored, the join block functions like a barrier for parallel message processing.

The named token can be applied to the join block as follows:

```
1. token x = join {
2.     a<–m1();
3.     b<–m2();
4. };
```

The following example illustrates how to access the join block return values through tokens. In lines 16-20, the message `multiply` will not be processed until the three messages `add(2,3)`, `add(3,4)`, and `add(2,4)` are processed. The token passed to `multiply` is an array of `Integers` generated by the three `adds` messages. The message handler `multiply(Object numbers[])` in lines 3-7 receives the result of the join block.

```
1. behavior JoinContinuation {
2.
3.   int multiply(Object numbers[]){
4.     return ((Integer)numbers[0]).intValue() *
5.            ((Integer)numbers[1]).intValue() *
6.            ((Integer)numbers[2]).intValue();
7.   }
8.
9.   int add(int n1, int n2) {
10.    return n1 + n2;
11.  }
12.
13.  void act(String args[]) {
14.
15.    standardOutput<–print("Value: ") @
16.    join {
17.      add(2,3);
18.      add(3,4);
19.      add(2,4);
```

```
20.     } @ multiply( token ) @ standardOutput<-println( token );
21.   }
22.
23.}
```

5.3 Message Properties

SALSA provides four message properties that can be used with message sending: `priority`, `delay`, `waitfor`, and `delayWaitfor`. The syntax used to assign to a message a given property is the following, where <`property name`> can be either `priority`, `delay`, `waitfor`, and `delayWaitfor`:

actor<-myMessage:<property name>

5.3.1 Property: `priority`

The `priority` property is used to send a message with high priority. This is achieved by placing the message at the beginning of the actor's message box when it is received. For instance, the following statement will result in sending the message `migrate` to the actor, `book`, with the highest property.

```
book<-migrate("rmsp://europa.wcl.cs.rpi.edu:4040/"):priority;
```

For example, Let us assume that the local host is overloaded, the message box of Actor `book` is full, and the remote host to migrate has extra computing power. Using the `priority` property by attaching it to the `migrate` message may improve the performance.

5.3.2 Property: `delay`

The `delay` property is used to send a message with a given delay. It takes as arguments the delay duration in milliseconds. The property is usually used as a loose timer. For instance, the following message `awaken` will be sent to the actor, `book`, after a delay of 1s.

```
// The message awaken() will be delivered to Actor book
//  after a delay of 1 second.
book<-awaken():delay(new Integer(1000));
```

5.3.3 Property: `waitfor`

The `waitfor` property is used to wait for the reception of a token before sending a message. The property can add variable continuation restrictions dynamically, enabling a flexible and-barrier for concurrent execution. The following example shows that the message `compare(b)` can be delivered to Actor `a` if Actors `a` and `b` have migrated to the same theater:

```
token x = a<-migrate("rmsp://europa.cs.rpi.edu:4040");
token y = b<-migrate("rmsp://europa.cs.rpi.edu:4040");
a<-compare(b) : waitfor(x,y);
```

5.3.4 Property: `delayWaitfor`

SALSA 1.1.2 does not support multiple properties. `delayWaitfor` is a temporary solution to support `delay` and `waitfor` in the same message. The `delayWaitfor` property takes the first argument as the delay duration in milliseconds, and the remainder as tokens. For instance, the message `compare(b)` can be delivered to Actor `a` if Actors `a` and `b` have migrated to the same theater and after a delay of 1 second:

```
// The message compare(b) will be delivered to Actor a
//  if Actors a and b has migrated to the target theater
//  and after a delay of 1 second.
token x = a<-migrate("rmsp://europa.cs.rpi.edu:4040");
token y = b<-migrate("rmsp://europa.cs.rpi.edu:4040");
a<-compare(b) : delayWaitfor(new Integer(1000),x,y);
```

CHAPTER 6
Actor Garbage Collection

Actor garbage collection is the mechanism used to reclaim useless actors. A system can fail because of memory leakage, resulting from uncollected garbage actors. Manual garbage collection can solve this problem if an application does not require a lot of dynamic memory allocation operations, however it is error-prone and can reduce programmers' efficiency. As the size of the application becomes larger and more complex, automatic garbage collection becomes preferable, because faulty manual garbage collection can cause memory security issues. Additionally, manual garbage collection is contrary to high-level programming. From the perspective of software engineering, there should be a focus on the development of functionalities, not on concerns which can be effectively automated. The garbage collection mechanism used by SALSA is automatic.

Many object-oriented programming languages support automatic garbage collection, such as Smalltalk, Scheme, and Java. Unfortunately, these garbage collection algorithms cannot be used for actor garbage collection directly, because actors encapsulate a thread of control that repeatedly waits for incoming messages to process, and these approaches do not collect active objects. The encapsulated thread of control results in the essential difference of actor garbage collection and object garbage collection.

6.1 Actor Garbage Definition

The definition of actor garbage relates to meaningful computation. Meaningful computation is defined as having the ability to communicate with any of the *root actors*, that is, to access any resource or public service. The widely used definition of live actors is described in [4]. Conceptually, an actor is live if it is a root or it can either potentially: 1) receive messages from the root actors or 2) send messages to the root actors. The set of actor garbage is then defined as the complement of the set of live actors. To formally describe the SALSA actor garbage collection (GC)

model, we introduce the following definitions:

- **Blocked actor**: An actor is blocked if it has no pending messages in its message box and it is not processing any message. Otherwise it is *unblocked*.
- **Reference**: A reference indicates an address of an actor. Actor A can only send messages to actor B if A has a reference pointing to B.
- **Inverse reference**: An inverse reference is a conceptual reference from the target of a reference to its source.
- **Acquaintance**: Let actor A have a reference pointing to actor B. B is an acquaintance of A, and A is an inverse acquaintance of B.
- **Root actor**: An actor is a root actor if it encapsulates a resource, or if it is a public service — such as I/O devices, web services, and databases (i.e., a service actor).

The original definition of live actors is denotational because it uses the concept of "potential" message delivery and reception. To make it more operational, we use the term "*potentially live*" [2] to define live actors.

- **Potentially live actors**:
  - Every unblocked actor and root actor is potentially live.
  - Every acquaintance of a potentially live actor is potentially live.
- **Live actors**:
  - A root actor is live.
  - Every acquaintance of a live actor is live.
  - Every potentially live, inverse acquaintance of a live actor is live.

6.2 Actor Garbage Collection in SALSA

This section introduces the assumptions and the actor garbage collection mechanism used in SALSA.

6.2.1 The Live Unblocked Actor Principle

In actor-oriented programming languages, an actor must be able to access resources which are encapsulated in service actors. To access a resource, an actor requires a reference to the service actor which encapsulates it. This implies that actors keep persistent references to some special service actors — such as the file system service and the standard output service. Furthermore, an actor can explicitly create references to public services. For instance, an actor can dynamically convert a string into a reference to communicate with a service actor, analogous to accessing a web service by a web browser using a URL.

Without program analysis techniques, the ability of an actor to access resources provided by an actor-oriented programming language implies explicit reference creation to access service actors. The ability to access local service actors (e.g. the standard output) and explicit reference creation to public service actors make the statement true: *every actor has persistent references to root actors*. This statement is important because it changes the meaning of actor GC, making actor GC similar to passive object GC. It leads to the *live unblocked actor principle*, which says every unblocked actor is live. Since each unblocked actor is: 1) an inverse acquaintance of the root actors and 2) defined as potentially live, it is live according to the definition of actor GC. With the live unblocked actor principle, every unblocked actor can be viewed as a root. Liveness of blocked actors depends on the transitive reachability from unblocked actors and root actors. If a blocked actor is transitively reachable from an unblocked actor or a root actor, it is defined as potentially live. With persistent root references, such potentially live, blocked actors are live because they are inverse acquaintances of some root actors.

Notice that the live unblocked actor principle may not be true if considering resource access restrictions. This implies that different security models may result in different sets of actor garbage, given the same actor system. At the programming language level, SALSA assumes the live unblocked actor principle.

6.2.2 Local Actor Garbage Collection

The difficulty of local actor garbage collection is to obtain a consistent global state and minimize the penalty of actor garbage collection. The easiest approach for actor garbage collection is to stop the world: no computation or communication is allowed during actor garbage collection. There are two major drawbacks of this approach, the waiting time for message clearance and a degradation of parallelism (only the garbage collector is running and all actors are waiting).

A better solution for local actor garbage collection is to use a snapshot-based algorithm [7], which can improve parallelism and does not require waiting time for message clearance. SALSA uses a snapshot based algorithm, together with the SALSA garbage detection protocol (the pseudo-root approach) [6], in order to get a meaningful global state. A SALSA local garbage collector uses the meaningful global snapshot to identify garbage actors.

One can observe the behavior of the local actor garbage collector by specifying the run-time environment options *-Dgcverbose -Dnodie* as follows:

java -cp salsa<version>.jar:. -Dgcverbose -Dnodie examples.HelloWorld

To start a theater without running the local actor garbage collection, one can use the option *-Dnogc* as follows:

java -cp salsa<version>.jar:. -Dnogc examples.HelloWorld

6.2.3 Optional Distributed Garbage Collection Service

Distributed actor garbage collection is more complicated because of actor migration and the difficulty of recording a meaningful global state of the distributed system. The SALSA garbage detection protocol and the local actor garbage collectors help simplify the problem — they can handle acyclic distributed garbage and all local garbage.

The current SALSA distributed actor garbage collector is implemented as a logically centralized, optional service. When it is triggered to manage several hosts, it coordinates the local collectors to get a meaningful global snapshot. Actors referenced by those outside the selected hosts are never collected. The task of identifying

garbage is done in the logically centralized service. Once garbage is identified, a garbage list is then sent to every participating host.

A distributed garbage collection service collects garbage for selected hosts (theaters). It collects distributed garbage providing a partial view of the system. SALSA also supports hierarchical actor garbage collection. To build the garbage collection hierarchy, each distributed garbage collection service requires its parent (at most one) and its children. The usage of the distributed garbage collection service is shown as follows:

java -cp salsa<version>.jar:. gc.serverGC.SServerPRID <n> <parent> <child1 or host1> <child2 or host2> .....

n specifies that the service should be activated every n seconds. $n = -1$ means that the service only executes once and then terminates. `parent` specifies the address of the parent service, with the format <`ip`>:<`port`>. Invalid format indicates that the service is the root service. <`child1 or host1`> indicates the address of the child service or the target theater, with the format <`ip`>:<`port`>. Notice that a theater is always a leaf.

To run the distributed garbage collection once, one can use the command as follows:

java -cp salsa<version>.jar:. gc.serverGC.SServerPRID -1 x <host1> <host2> .....

To run it every 40 seconds, use:

java -cp salsa<version>.jar:. java gc.serverGC.SServerPRID <40> x <host1> <host2> .....

Note that n should be large enough, or else performance can be severely degraded. The default is $n = 20$.

LITERATURE CITED

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

[2] Peter Dickman. Incremental, distributed orphan detection and actor garbage collection using graph partitioning and Euler cycles. In *WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, pages 141–158, Bologna, October 1996. Springer-Verlag.

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

[4] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actors. In *OOPSLA'90 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 126–134. ACM Press, October 1990.

[5] Carlos A. Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 ACM Conference on Object-Oriented Systems, Languages and Applications*, 36(12):20–34, December 2001.

[6] Wei-Jen Wang and Carlos A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In *Advances in Grid and Pervasive Computing, First International Conference, GPC 2006*, volume 3947 of *Lecture Notes in Computer Science*, pages 360–372. Springer, May 2006.

[7] Wei-Jen Wang and Carlos A. Varela. A non-blocking snapshot algorithm for distributed garbage collection of mobile active objects. Technical Report 06-15, Dept. of Computer Science, R.P.I., October 2006.

APPENDIX A
Name Server Options and System Properties

A.1 Name Server Options

The name server can be run with several arguments. Running the name server with the command `-h` provides all the possible options.

```
java wwc.naming.NamingServer -h

usage:

     java ...WWCNamingServer

     java ...WWCNamingServer -h

     java ...WWCNamingServer -v

     java ...WWCNamingServer -p portNumber

options:

     -h:  Print this message.

     -v:  Print version number.

     -p portNumber:  Set the listening port to portNumber.  Default port
        number is 3030.
```

A.2 System Properties

SALSA programs can be executed with a set of system properties:

```
-Dport=<port number>:  To specify the port number that the automatically
    started theater will listen to.  Otherwise, a random port number is
    used.

-Didentifier=<id>:  To specify the relative locator of the bootstrapping
    actor's UAL.
```

```
-Duan = <uan>:  To specify the UAN of the bootstrapping actor.

-Dual= <ual>:  To specify the UAL of the bootstrapping actor.

-Dnogc:  The local garbage collector will not be triggered

-Dgcverbose:  To show the behavior of local GC

-Dnodie:  Making a dynamically created theater alive
```

Here comes the example:

```
java -Dport = 5050 -Didentifier = actor/hello HelloWorld
```

A theater is started at the current host (e.g. `europa.wcl.cs.rpi.edu`). The dynamically created theater listens on port 5050, and the `HelloWorld` actor has the UAL:

```
rmsp://europa.wcl.cs.rpi.edu:5050/actor/hello
```

APPENDIX B
Debugging Tips

- Make sure you really understand the actor model, its message passing semantics, and the concurrency coordination abstractions built in SALSA.
- Message passing and remote procedure calls are totally different. A named token variable does not have the result immediately. It has the result only after the message gets executed.
- Objects in messages have pass-by-value semantics. This means that object arguments are cloned and then sent. A latter modification on these object arguments does not change the objects which were originally sent.
- Since the SALSA compiler does not support type checking in this version, you may need to go through the Java source code. The code related to your program is on the bottom of the generated Java source code. Do not try to modify other parts irrelevant to your SALSA source code.
- Please note that a typo in a message sending statement does not generate Java or SALSA compile-time errors. You have to be very careful with that. A run-time error will be generated instead.
- Most people confuse `self` with `this`. `this` means "this actor", while `self` "the actor reference" pointing to itself. `self` can only be used as a target of messages, or an argument to be passed around. `this` can be used for object method invocation. To send your references to other actors, use `self`. Using `this` is wrong.

APPENDIX C
Learning SALSA by Example Code

One can download the SALSA source code at

`http://wcl.cs.rpi.edu/salsa/`,

which includes several good examples.

C.1 Package `examples`

Package `examples` are useful for learning SALSA. Examples consist of:

- `examples.addressbook`: The address book example shown in Section 4.2.
- `examples.cell`: It implements a cell actor that has two message handlers: `set` to set the value, and `get` to get the value of the cell. Both versions of distributed and single host examples are provide.
- `examples.chat`: Two `Speaker` actors run as services and start a chat session which is triggered by the `Chat` actor.
- `examples.fibonacci`: The recursive `fibonacci` application.
- `examples.Heat`: A simple simulation of heat flow. Both distributed and single host versions are provided.
- `examples.helloworld`: The `HelloWorld` example.
- `examples.messenger`: An example showing message delivery.
- `examples.migration`: An example to show how to migrate an actor.
- `examples.multicast`: A group of examples showing how to implement several multicast protocols.
- `examples.nqueens`: A program that tries to solve the N-Queens problem.

- `examples.numbers`: Examples of actor inheritance and concurrency coordination.
- `examples.ping`: An example showing the echo server and the ping client.
- `examples.ticker`: A ticker example: a non-terminating actor
- `examples.trap`: This numerical application approximates the integral of a function over an interval [a, b] by using the trapezoidal approximation.

C.2 Package `tests`

Package `tests` is used for testing SALSA, including language and run-time environment tests.

- `tests.language`: Tests for SALSA language constructs.
- `tests.language.babyfood`: Tests for inheritance.
- `tests.localgc`: Correctness tests for local actor garbage collection.
- `tests.distributed`: Correctness tests for distributed actor garbage collection.

APPENDIX D
Visualization of SALSA Applications with OverView

D.1 Introduction

OverView is a toolkit for visualization of distributed systems, and is designed to be generic (that is, able to be applied to many different distributed systems), scalable (that is, to scale up to very large distributed systems), and dynamic (that is, functioning as well online as it does offline). OverView is written in Java and is designed to work with any arbitrary Java system via a custom unintrusive profiling mechanism and a simple declarative language (called the Entity Specification Language) which describes how to map Java method invocations into a high-level description of visualization events. Figure D.1 shows how OverView visualizes an actor creation event.

Please note that OverView requires Java 1.5 to compile and run.

D.2 Using OverView with SALSA Programs

To visualize SALSA programs, SALSA has been instrumented by including event-sending behavior. In this way, any executed SALSA program may be visualized without any additional configuration.

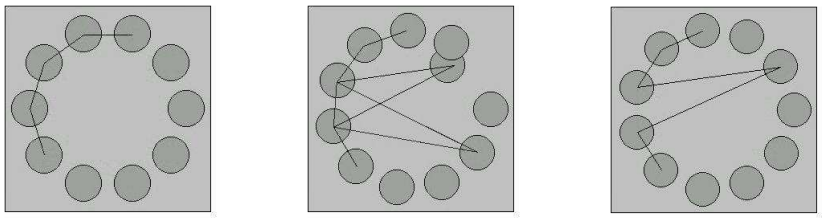

Figure D.1: These three figures, from left to right, were captured from OverView when an actor was created. The outer square represents a SALSA theater, while the inner circles represent SALSA actors. Lines between actors represent SALSA message sending behavior.

The easiest way to use OverView to visualize SALSA programs is to download the latest OverView-instrumented SALSA binary, which can be found at:

```
http://wcl.cs.rpi.edu/overview/
```

You should download `overview<version>.jar` and `salsa<version>i.jar`, and place them in a convenient location. When executing your SALSA application, simply ensure that both JAR files are in your Java classpath (by either using the `CLASSPATH` environment variable, or by using the `-cp` command line switch).

If you wish to compile OverView and SALSA from source and instrument SALSA yourself, please see the respective documentations for OverView and SALSA.

To run OverView and visualize your SALSA application, you must keep in mind that `event sinks` must be started before `event sources`; in practical terms, this means that the OverView visualization must be running before you start your SALSA program.

To do so is relatively simple: you will need an *OverView Daemon* (OVD) to collect and forward events, and an *OverView Presenter* (OVP) to display the visualization.

```
java overview.ovd.OverViewDaemon
```

```
java overview.ovp.OverViewPresenter <host:port of OVD>
```

If you are running OVP and OVD on the same machine, OVD's host:port will generally be `localhost:6060`.

After OverView is running, you may start your SALSA theaters, name servers, and so on. Merely note that every SALSA program that is run (including theaters!) must have the command line switch `-DovHost=<host:port of OVD>` to enable event sending, and to tell SALSA where to send events.

If you wish to use the instrumented version of SALSA without OverView, simply don't specify `-DovHost`*! SALSA programs will then run as usual, without trying to send events.*

You can test OverView and SALSA with the following SALSA example:

```
java -DovHost=<host:port of OVD> examples.fibonacci.Fibonacci 6
```

You should begin to see a visualization of the sequence of Fibonacci numbers being calculated recursively, using SALSA actors.

APPENDIX E
History of SALSA

SALSA has been developed since 1998 by Carlos A. Varela, who was a Ph.D. student directed by Gul Agha at University of Illinois at Urbana-Champaign (UIUC). During the UIUC period, Gregory Haik, a student of Jean-Pierre Briot from the University of Paris 6, visiting Gul Agha's lab, made contributions to the transport layer of SALSA from versions 0.2 to 0.3.3. Carlos A. Varela founded the Worldwide Computing Laboratory and continued the development of SALSA when he became a faculty member at Rensselaer Polytechnic Institute in 2001. Many of his students have participated in the SALSA project since then, including Wei-Jen Wang, Travis Desell, Kaoutar El Maghraoui, Jason LaPorte, Abe Stephens, and Robin Toll. Abe Stephens and Travis Desell were the major contributors of SALSA 0.4 to 0.5. They worked together from 2001 until Abe Stephens graduated. Robin Toll also made contributions to the SALSA compiler during this period of time. During 2003 and 2004, Travis Desell worked on SALSA 0.6 and 0.7, which then became the foundation of SALSA 1.0. Advanced features such as message properties, name tokens, and join block continuations were introduced at that time. Wei-Jen Wang and Kaoutar El Maghraoui joined the SALSA project since 2003. Kaoutar El Maghraoui made major contributions to the development of the communication layer of the SALSA 0.7. Wei-Jen Wang has been a major developer since 2004. He made another major change on SALSA in 2004 and released SALSA 1.0 to 1.1.2 (the current release) in 2005 and 2006. He introduced actor garbage collection, fault-tolerance communication using persistent sockets and messages, actor name deletion support for naming services, and passing-by-value message delivery. Jason LaPorte joined the SALSA project in 2006. His contributions relate to the interface between SALSA and the OverView project. In 2007, Travis Desell redesigned SALSA from the ground up with performance and concurrency in mind, and named it SALSA Lite 2.0.

APPENDIX F
SALSA Grammar

The SALSA grammar is listed as follows:

```
CompilationUnit ::=
  [ModuleDeclaration]
  (ImportDeclaration)*
  BehaviorDeclarationAttributes
  (BehaviorDeclaration | InterfaceDeclaration )
  <EOF>

ModuleDeclaration ::=
  "module" Name ";"

ImportDeclaration ::=
  "import" <IDENTIFIER> ("." (<IDENTIFIER> | "*") )* ";"

BehaviorDeclarationAttributes ::=
  ("abstract" | "public" | "final")*

InterfaceDeclaration ::=
  "interface" <IDENTIFIER> ["extends" Name] InterfaceBody

Name ::=
  <IDENTIFIER> ("." <IDENTIFIER>)*

InterfaceBody ::=
  ( "{"
    (StateVariableDeclaration | MethodLookahead ";" )*
    "}"
  )*

BehaviorDeclaration ::=
  "behavior" <IDENTIFIER>
  ["extends" Name]
  ["implements" Name ("," Name)*]
  BehaviorBody

MethodLookahead ::=
  MethodAttributes ( Type | "void" )
  <IDENTIFIER> FormalParameters
  ["throws" Exceptions]
```

```
BehaviorBody ::=
  "{"
    ( Initializer | NestedBehaviorDeclaration |
      StateVariableDeclaration | MethodDeclaration |
      ConstructorDeclaration
    )*
  "}"

NestedBehaviorAttributes ::=
  ("abstract" | "public" | "final" | "protected" |
   "private" | "static"
  )*

NestedBehaviorDeclaration ::=
  NestedBehaviorAttributes BehaviorDeclaration

Initializer ::=
  ["static"] Block

StateVariableAttributes ::=
  ("public" | "protected" | "private" | "volatile" |
   "static" | "final"     | "transient"
  )*

StateVariableDeclaration ::=
  StateVariableAttributes
  Type
  VariableDeclaration
  ("," VariableDeclaration)* ";"

PrimitiveType ::=
  "boolean" | "char"  | "byte"   | "short" | "int" |
  "long"    | "float" | "double"

Type ::=
  (PrimitiveType | Name) ( "[" "]" )*

VariableDeclaration ::=
  <IDENTIFIER> ("[" "]")*
  ["=" (Expression | ArrayInitializer)]

ArrayInitializer ::=
  "{"
  [ (Expression | ArrayInitializer)
    ("," (Expression | ArrayInitializer) )*
  ]
```

```
  "}"

AssignmentOperator ::=
  "="   | "*="   | "/="   | "%=" | "+=" | "-=" |
  "<<=" | ">>="  | ">>>=" | "&=" | "^=" | "|="

Expression ::=
  Value
  (
   ((Operator | AssignmentOperator) Value) |
   ("?" Expression ":" Value)
  )*

Operator ::=
  "||" | "&&" | "|"  | "^"  | "&"  | "==" | "!="  |
  ">"  | "<"  | "<=" | ">=" | "<<" | ">>" | ">>>" |
  "+"  | "-"  | "*"  | "/"  | "%"  | "instanceof"

Value ::=
  [Prefix] Variable [Suffix] (PrimarySuffix)*

Prefix ::=
  "++" | "--" | "~" | "!" | "-"

Suffix ::=
  "++" | "--"

Variable ::=
  ["(" Type ")"]
  (
    Literal | Name | "this"  | "super" |
    AllocationExpression | "(" Expression ")"
  )

PrimarySuffix ::=
  "." "this" | "." AllocationExpression |
  "[" Expression "]" | "." <IDENTIFIER> |
  Arguments

ResultType ::=
  Type | "void"

Literal ::=
  IntegerLiteral | FloatingPointLiteral |
  CharacterLiteral | StringLiteral |
  BooleanLiteral | NullLiteral |
```

```
  TokenLiteral

Arguments ::=
  "(" [Expression ("," Expression)*] ")"

AllocationExpression ::=
  "new" PrimitiveType ArrayDimsAndInits |
  "new" Name
    (ArrayDimsAndInits | (Arguments [BehaviorBody]))
    [BindDeclaration]

BindDeclaration ::=
  "at" "(" Expression ["," Expression] ")"

ArrayDimsAndInits ::=
  ( "[" Expression "]")+ ("[" "]")* |
  ("[" "]")+ ArrayInitializer

FormalParameters ::=
  "("
    [ ["final"] Type <IDENTIFIER> ( "[" "]" )*
      ( "," ["final"] Type <IDENTIFIER> ( "[" "]" )* )*
    ]
  ")"

ExplicitConstructorInvocation ::=
  "super" Arguments ";"

ConstructorDeclaration ::=
  MethodAttributes <IDENTIFIER> FormalParameters
  ["throws" Exceptions]
  "{"
    [ExplicitConstructorInvocation] (Statement)*
  "}"

ConstructorAttributes ::=
  ("public" | "protected" | "private")*

MethodDeclaration ::=
  MethodAttributes
  (Type | "void") <IDENTIFIER> FormalParameters
  ["throws" Exceptions] Block

MethodAttributes ::=
  ("public"   | "protected" | "private" | "static" |
   "abstract" | "final"     | "native"
```

```
  )*

Exceptions ::=
  Name ("," Name)*

Statement ::=
  ContinuationStatement |
  TokenDeclarationStatement |
  LocalVariableDeclaration ";" |
  Block |
  EmptyStatement |
  StatementExpression ";" |
  LabeledStatement |
  SynchronizedStatement |
  SwitchStatement |
  IfStatement |
  WhileStatement |
  DoStatement |
  ForStatement |
  BreakStatement |
  ContinueStatement |
  ReturnStatement |
  ThrowStatement |
  TryStatement |
  MethodDeclaration |
  NestedBehaviorDeclaration

Block ::=
  "{" ( Statement )* "}"

LocalVariableDeclaration ::=
  ["final"] Type
  VariableDeclaration ("," VariableDeclaration)*

EmptyStatement ::=
  ";"

StatementExpression ::=
  Value [AssignmentOperator Expression]

LabeledStatement ::=
  <IDENTIFIER> ":" Statement

SwitchStatement ::=
  "switch" "(" Expression ")"
  "{" (SwitchLabel (Statement)* )* "}"
```

```
SwitchLabel ::=
  "case" Expression ":" | "default" ":"

IfStatement ::=
  "if" "(" Expression ")" Statement ["else" Statement]

WhileStatement ::=
  "while" "(" Expression ")" Statement

DoStatement ::=
  "do" Statement "while" "(" Expression ")" ";"

ForInit ::=
  [ LocalVariableDeclaration |
    ( StatementExpression
      ("," StatementExpression)*
    )
  ]

ForCondition ::=
  [Expression]

ForIncrement ::=
  [ StatementExpression
    ("," StatementExpression)*
  ]

ForStatement ::=
  "for"
  "(" ForInit ";" ForCondition ";" ForIncrement ")"
  Statement

BreakStatement ::=
  "break" [<IDENTIFIER>] ";"

ContinueStatement ::=
  "continue" [<IDENTIFIER>] ";"

ReturnStatement ::=
  "return" [Expression] ";"

ThrowStatement ::=
  "throw" Expression ";"

SynchronizedStatement ::=
```

```
  "synchronized" "(" Expression ")" Block

TryStatement ::=
  "try" Block
  (
    "catch" "(" ["final"] Type <IDENTIFIER> ")" Block
  )*
  ["finally" Block]

ContinuationStatement ::=
  (MessageStatement "@")*
  (MessageStatement | "currentContinuation") ";"

MessageStatement ::=
[NamedTokenStatement] (MessageSend | JoinBlock)

JoinBlock ::=
  "join" Block

NamedTokenStatement ::=
  (<IDENTIFIER> | "token" <IDENTIFIER>) "="

MessageSend ::=
  [Value "<-"] <IDENTIFIER> MessageArguments
  [":" MessageProperty]

MessageProperty ::=
  <IDENTIFIER> [Arguments]

MessageArguments ::=
  "(" [Expression ("," Expression)*] ")"

TokenDeclarationStatement ::=
  "token" <IDENTIFIER> "=" Expression ";"

IntegerLiteral ::=
  <INTEGER_LITERAL>

FloatingPointLiteral ::=
  <FLOATING_POINT_LITERAL>

CharacterLiteral ::=
  <CHARACTER_LITERAL>

StringLiteral ::=
  <STRING_LITERAL>
```

```
BooleanLiteral ::=
  "true" | "false"

NullLiteral ::=
  "null"

TokenLiteral ::=
  "token"

<IDENTIFIER> ::=
  <LETTER> (<LETTER>|<DIGIT>)*
```