THE SALSA PROGRAMMING LANGUAGE 1.0.2 RELEASE TUTORIAL

By

Worldwide Computing Lab.

Rensselaer Polytechnic Institute
Troy, New York

March 2005

CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 placed new demands and challenges such as openness, portability, ability to adapt quickly to changing execution environments, and highly dynamic reconfiguration. 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 a lot of support to 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) 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 hence 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 ne migrated to distributed nodes at run-time.

1.2 Structure of the Tutorial

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 continuations, 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. 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 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. Section 2 illustrates how SALSA supports the actor model.

2.1 The Actor Model

Actors [1] 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 receive messages not necessarily in the same order that they were 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 infinitely often ready to receive a message will eventually get 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 communicated 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, in particular because there is no shared memory, actor systems are highly reconfigurable.

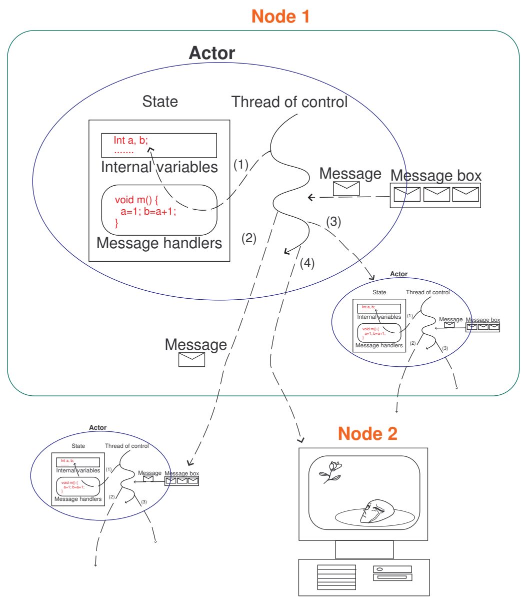

Figure 2.1: Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) create new actors, (3) send messages to peer actors, and/or (4) migrate to another computing host.

2.2 How SALSA Supports the Actor Model

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` ketword). Creating a actor returns its reference.
- The message 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.

2.3 Beyond the Actor Model

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

CHAPTER 3
Writing Concurrent Programs

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

3.1 SALSA Language Support for 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 test{
  Integer a;
  int b;

  void act(String[] args) {
    a=new Integer(3);
    b=a.intValue()+2;
  }
}
```

3.2 Actor Creation

The actor reference is one of the main new primitive values of SALSA. There are three approaches to get an actor reference: either by actor creation statement, the `getReferenceByName()` function, or passed arguments from messages. This section only concentrates on actor creation and reference passing.

Writing a constructor and constructing an actor in SALSA programming is similar to object construction in Java programming. For instance, one can declare

[1]SALSA, as of version 1.1.0, does not enforce state encapsulation and serializability. This discipline needs to be followed by programmers, especially for distributed applications.

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. No shared memory is possible among actors.

3.3 Message Passing

To achieve more parallelism, SALSA uses message passing. A SALSA method is named a *message handler*.

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 key word `<-`, 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 method 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:

```
// Wrong! It does not compile!!!
self <- someObject.someHandler();
```

To avoid unexpected race conditions by accessing the passed arguments, messages of SALSA enforces pass-by-value for every object, and pass-by-reference for every actor. 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 behaviors of actors: *token passing continuations*, *join continuations*, and *first-class continuations*.

3.4.1 Token passing continuations

Token passing continuations are designed for specify the order of message processing. The reserved keyword '`@`' 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 can 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 ouput:

```
// 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 continuations

The previous sub-section has illustrated how token passing continuations work in message passing. This sub-section introduces join continuations which can specify the barrier of parallel processing and join the results in the next message following by `@`. A join continuation has a scope (or block) starting at "`{join`" and ending with "}". Every message inside the block must be executed, and then the next message, followed by `@`, can be started for execution. For instance, the following example shows 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 is much more complicated. It is explained in the Chapter 5.

3.4.3 First-class continuations

The purpose of the 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 a recursive continuation code. In SALSA, the keyword `currentContinuation` is reserved for first-class continuations. To explain the effect of first-class continuations, we use two examples, with and without it, to show the difference. In the first example, statement 1 prints out "Hello World SALSA":

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

```
saySomething() @  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 First-Class Continuation
// statement 2 may produce a different result from
// that of statement 1.
...
void saySomething() {
  standardOutput <- print("Hello ") @
  standardOutput <- print("World ");
}
....
//statement 2 inside some method:
saySomething() @  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 is meaningless. For instance, the following piece of code always prints out "Hello World", but "SALSA" never gets printed:

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

3.5 I/O Actor Access

SALSA provides three kinds of I/O actors to support asynchronous I/O accesses. One of them is an input service (`standardInput`), and two are output services (`standardOutput` and `standardError`). Since they are actors, only asynchronous accesses are possible.

`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 Your SALSA Program.

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

3.6 Writing Your First SALSA Program

This section demonstrates how to write, compile, and execute your SALSA program

3.6.1 Steps to compile and execute your SALSA program

SALSA hides many of the details 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 byte code ready to be executed on any virtual machine implementation (see Table 3.1).

3.6.2 `Hello World` example

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

```
/* HelloWorld.salsa */
module examples;
behavior HelloWorld {
  void act( String[] arguments ) {
    standardOutput<-print( "Hello" )@
    standardOutput<-print( "World!" );
  }
}
```

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

The first line is a comment. SALSA 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 runtime environment. The act message is used as a bootstrap 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 (<-) indicated 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 `world` message to be sent only after the `hello` message has been processed. This is referred to as token-passing continuation.

3.6.3 Compiling and running `Hello World`

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

- 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. Note that SALSA modules are similar to java packages. So you have to respect the directory structure 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 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 (:).

```
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 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 web into a unified distributed computing infrastructure. Worldwide computing tries to harness underutilized resources in the web by providing to 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 World-wide computer (WWC) architecture

The world-wide computer 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 and remote communication. 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 between local actor references their names, and a runtime environment. The RMSP server runs forever listening for incoming requests from remote actors and starting multiples threads to handle incoming requests simultaneously.

The world-wide computer (WWC) consists of the following key components:

- Universal naming

Table 4.1: UAL and UAN.

TYPE	EXAMPLE
URL	http://www.cs.rpi.edu/wwc/
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 - UAN and UAL

Universal naming allows actors to become universal actors. Universal actors have the ability of migration, while anonymous actors don't. Service actors are a special kind of universal actors, which grant universally accessing privileges to every actor and never get collected by actor garbage collector.

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 through 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. The format of UAN and UAL follow the URI syntax. They are similar in format to a URL (see Table 4.1).

UAN: 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 this is a practice that should be avoided. The last item specifies the name of the actor. If a port number is not specified, the default port number (3030) of the name server is used.

UAL: The first item specifies the protocol used for remote message sending. The second item indicates the theater's machine name and port number. If no port is indicated, a random port is used. The last part specifies the location name of the actor in the given theater.

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

- Platform independence: names should 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.

4.1.3 Actor garbage collection

Actor garbage collection is the mechanism to reclaim useless actors. A system eventually fails because of the memory leakage, resulted by useless actors. Manual management of actor garbage collection is error-prone and always reduces programmers' throughput, which introduces the idea of automatic garbage collection.

Many object-oriented programming languages support automatic garbage collection, such as Smalltalk, Scheme, Java, ... etc. Unfortunately, these garbage collection algorithms cannot be used for actor garbage collection directly, because each actor has a thread of control encapsulated in it. An actor can create or delete references to other actors, while an object cannot. The thread of control results in the essential difference of actor garbage collection and object garbage collection.

To formally define garbage actors, we have to clarify the concept of *the root set of actors*. The root set of actors cannot be reclaimed due to its nature. The root set of actors are:

- human beings (communicated by I/O channels),
- I/O devices such as sensors, disks, ...etc, or

- user-defined services which never terminate or use manual actor management.

The root set of actors are definately live. Live actors are those which can potentially communicate with the root set of actors. A *garbage actor* is one which has all the following properties:

- It does not belong to the root set.
- It cannot potentially send messages to the root set.
- It cannot potentially receive messages from any of the root set.

The difficulty of local actor garbage collection is to get a consistent global state and minimize the penalty of actor garbage collection. The easiest approach for actor garbage collection is stop-the-world: no computation is allowed during actor garbage collection. There are two major drawbacks of this approach, the waiting time for message clearance and parallelism degrading (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, which can alleviate parallelism degrading and does not require the waiting time for message clearance. SALSA uses a snapshot based algorithm, together with the SALSA two-pahse reference exchanging protocol and the SALSA asynchronous message acknowledgement protocol, in order to get a meaningful global state. The above two protocols are together named *the SALSA garbage detection protocol*. A SALSA local garbage collector uses the meaningful global snapshot to identify garbage actors.

Distributed actor garbage collection is much more complicated because of the mobility of actors, the difficulty of recording a meaningful global state of the distributed system, and the independent execution of the distributed and local garbage collection. 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 service. It is not a required 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 participated hosts.

The next version of SALSA will provide two different types of distributed garbage collectors: the hierarchical centralized service, and the distributed garbage-identifying collector.

4.2 How SALSA Supports the Worldwide Computing Model

This section demonstrates how to write a distributed SALSA program and run it in the world-wide computer.

4.2.1 Universal actor creation

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

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

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

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

An anonymous actor can be created as follows:

```
HelloWorld helloWorld = new HelloWorld()
  at (null, null);
```

4.2.2 Referencing actors

Actor references can be used as the target of message sending or arguments of messages. There are three ways to get actor references. Two of them, the return value of actor creation and references from message, are available in both distributed and concurrent SALSA programming. The last one, `getReferenceByName()`, is an

[2] Remember to start the naming server if UANs are involved in the computing.

explicit approach that translates a string or a UAN into a reference. In SALSA, only references of service actors should be obtained from 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 remoteSVC= (AddressBook)
  AddressBook.getReferenceByName("uan://nameserver1/id");
```

The other approach to get a reference by `getReferenceByName()` is to use `ActorReference`:

```
ActorReference remoteSVC=  ActorReference.getReferenceByName(
                           "uan://nameserver1/id");
```

Sometimes an actor wants to know who or where it is. An actor can get its UAL (where) by the function `getUAL()` and UAN (who) by `getUAN()`.

4.2.3 Migration

As mentioned before, only universal actors can migrate. Sending the message ”migrate” to an universal actor causes it to migrate seamlessly to the designated location. Its UAL will be changed and the Naming server 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 `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 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 services, such as the web server, the 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, by 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 in 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.

A service written by C or Java programming language uses loops to listen requests, preventing termination of it. A SALSA service should not use this approach because loops preclude an actor from executing messages. The way SALSA keeps a service actor alive is by specifying it in 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 and how message passing is used in SALSA. The example implements a simple address book. The `AddressBook` actor provides the functionality of creating new <name, email> entities, and responding to end users' requests. The example defines the `AddUser` behavior, which communicates with the `AddressBook` to add new entries in the database.

```
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();
  }

  // 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[]) {
    try{
      if (args.length != 0) {
        standardOutput<-println(
        "Usage:" +
        "java␣-Duan=<UAN>␣-Dual=<UAL>␣examples.AddressBook" );
        return;
      }
    } catch (Exception e) {
      standardOutput<-println("AddressBook␣at:␣") @
      standardOutput<-println("uan:␣" + getUAN()) @
      standardOutput<-println("ual:␣" + getUAL());
    }
  }
}
```

The `AddressBook` actor is bound to the UAN and UAL pair. This will result in placing the `AddressBook` actor in the designated location and notifying the Naming server.

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.

```
module examples;

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

4.3 Run-Time Support for WWC Application

The section demonstrates how to start theaters, the naming service, and the optional distributed actor garbage collection service. It also demonstrates how to observe and stop the local actor garbage collection.

4.3.1 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
```

4.3.2 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 actors' names lookup, updates, and deletions. UANP operates over TCP connections, usually the port 3030. This port can be overwritten by another port number.

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

4.3.3 Running an application

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 prints out ”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.

4.3.4 Local actor garbage collection

One can observe the behavior of the local actor garbage collector by specifying the runtime 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

4.3.5 Optional distributed garbage collection service

A distributed garbage collection service collects garbage for selected hosts (theaters). It can collect distributed acyclic garbage much faster than local collectors do. Furthermore, It can collect distributed cyclic garbage in the selected hosts, while local collectors cannot. To run the distributed garbage collection once, one can use the command as follows:

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

To run it every n seconds, use:

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

Note that n must be large enough because the distributed collector is poorly implemented. The performance will be improved in the next version of SALSA.

CHAPTER 5
Advanced Concurrency Coordination

This chapter introduces concepts of coordination by using named tokens, token continuations of join blocks, and message properties.

5.1 Named Tokens

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

In SALSA, the return value of the asynchronous message can be declared as a variable with type `token`. The variable is called the *named token*. Named tokens are designed to tell the SALSA run-time environment that how the continuations work.

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, and lines 1-2 uses a few token declarations, as follows:

```
//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, as follows:

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

The above 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) @
x = b<-m2(token, 9);
```

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

```
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.    }
```

As the token is updated as soon as the code is processed, this leads to some interesting occurrences. 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 Continuations

Chapter 3 skips some important issues of join block continuations. In this section, we are going to introduce how to use the join block return values and how to implement the join block reception handler.

A join block always returns *an object array* if it does join 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:

```
token x = join {
     a<-m1();
     b<-m2();
};
```

The return value of the join block is received by the join block reception handler. Every join block reception handler must have only one argument with the type of the object array. 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 is the join block reception handler.

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

actor<-myMessage:<property name>

5.3.1 Property: `priority`

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

```
book<-addUser(args[1], args[2]):priority;
```

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. For instance, the following message `addUser` will be sent to the actor, `book`, after a delay of 1s.

```
book<-addUser(args[1], args[2]):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.

```
token email = book<-email(new String("kaoutar"));
book<-addUser(email, new String("kaoutar")):waitfor(email);
```

Note that the `addUser` message will be sent after the reception of the token `email`. The above example is equivalent to the following example:

```
book<-email(new String("kaoutar")) @
book<-addUser(email, new String("kaoutar"));
```

5.3.4 Property: `delayWaitfor`

The `delayWaitfor` property is used to send a message with a given delay and to wait for the reception of a token. It takes the first argument as the delay duration in milliseconds, and the remainder as tokens. For instance, the following message `addUser` will be sent to the actor, `book`, after a delay of 1s, and the message can be processed until the target actor has received `token email` and `token ready2go`.

```
token email =  book<-email(new String("kaoutar"));
token ready2go = standardInput<-readLine();
book<-addUser(email, new String("kaoutar"))
      :delayWaitfor(new Integer(1000),ready2go);
```

LITERATURE CITED

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

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.

 -Didentifier=<id>:  To specify the ID portion of the actor's ual

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

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

```
-Dnogc:  The local garbage collecotr 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 -Dindentifier = actor/hello HelloWorld
```

If the theater is started at the host: `europa.wcl.cs.rpi.edu`, the theater is listening 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 message passing and the concurrency coordination model in SALSA. Most people have troubles dealing with token continuations.

- 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 runtime error will be generated instead.

- Message passing and remote procedure calls are totally different. The token continuation does not have the result immediately. It has the result only after the message gets executed.

- Objects in messages are pass-by-value. This means the object arguments are sent with the cloned objects. A latter modification on these object arguments does not change the values which had been sent.

- Most people confused `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. Using `this` as an argument must cause troubles.

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 addressbook example shown in Section 4.2.
- examples.cell: It implements a cell actor that has two message habdlers: `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 `Speake` actors run as services and start a chat session which is triggered by the `Chat` actor.
- examples.fibonacci: A SALSA version of the `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 multicasts.
- examples.nqueens: A SALSA program that tries to solve the famous 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.
- examples.trap: The 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.
- tests.language.babyfood: Tests for inheritance.
- tests.gclocal: One is the `fibonacci` application implemented by actor chains, and the other computes $1 + 2 + ... + n$.
- tests.distributed: Containing only one application, the distributed N-Queens solution number.

APPENDIX D
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>)*
```