THE SALSA PROGRAMMING LANGUAGE
2.0.0*alpha* 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

November 2009

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) [3] 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. Appendix A describes different options for starting SALSA theaters. Appendix B discusses advanced options for SALSA theaters and how to extend the SALSA theater framework. Appendix C introduces how to specify name server options. Appendix D provides debugging tips for SALSA programs. Appendix E provides brief descriptions of SALSA example programs. Appendix G presents the SALSA 2.0.0*alpha* 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. Section 2.3 discusses the SALSA 2.0 implementation of concurrency for reduced memory usage and improved performance.

2.1 The Actor Model

Actors [1, 2] 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

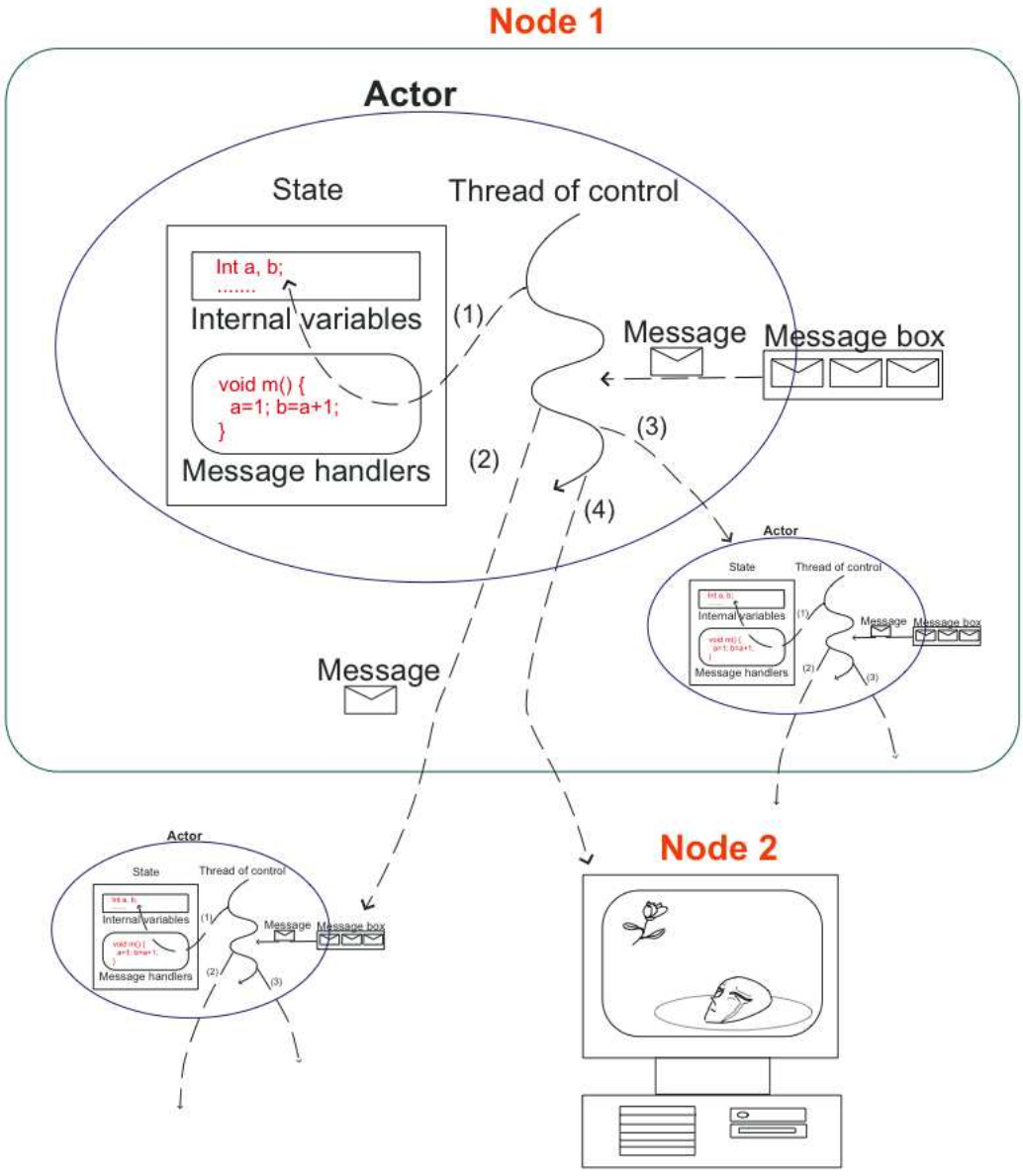

Figure 2.1: Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) send messages to peer actors, (3) create new actors, and/or (4) migrate to another computing host.

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.

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.

2.3 Concurrency in SALSA 2.0.0*alpha*

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 2.0.0*alpha*, 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. As the actor model enforces state encapuslation, actors can only change their own state and send messages to references of other actors. There is no primitive for the state of another actor.

There are two approaches to obtain a new actor reference: by constructing an actor with the `new` statement, or referencing an already existing actor with the `reference` statement (see Section 4.1.3 for more details). Actor references may also be passed in the arguments of a message. 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 and primitives are passed by value. As SALSA 2.0.0*alpha* has a focus on performance, objects are also passed by reference to actors running on the same theater (see Chapter 4 for more on theaters), but by value to actors running on different theaters (as they are serialized when passed in a message). This violates state encapsulation, so it is recommended that a user wrap an object in a Cell actor if it will be used by multiple actors, especially if they will be on different theaters. It is important to note that in SALSA, arrays are Java objects, and have the same pass by reference if local, but pass by value if remote semantics.

3.3 Actor Removal

By default, SALSA 2.0.0*alpha* applications do not use garbage collection. To remove an actor, send it a `destroy()` message. When this message is processed,

the actor will be flagged for garbage collection and removed along with all the Java objects in encapsulated by its state. Java objects which are known by other live actors will not be garbage collected. Garbage collection for SALSA programs which do not use remote communication has been implemented in SALSA 2.0.0*alpha*. To enable this see Appendix A.

3.4 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.

3.5 Coordinating Concurrency

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

3.5.1 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 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")`.

3.5.2 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.

3.5.3 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() ;
```

3.6 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()`

3.7 Developing SALSA Programs

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

3.7.1 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).

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 salsa_lite.core.compiler.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

3.7.2 `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 3.5.1).

3.7.3 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:. salsa_lite.core.compiler.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 for the World-Wide Computer (WWC)

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. This chapter first describes semantic support world-wide computing is described in Section 4.1. Following this, the WWC architecture is described and instructions on starting WWC daemons are given in Section 4.2. The chapter concludes with examples of running different SALSA applications on the WWC.

4.1 SALSA Semantics for World-Wide Computing

SALSA supports distributed computing on the WWC with semantics for universal actor naming, universal actor creation, remote actor referencing and actor migration. Remote message passing is done transparently, using the same semantics as in non-distributed SALSA programs.

4.1.1 Universal Naming

The Universal Actor Naming Protocol (UANP) is a protocol that supports universal actor naming and discovery. It defines how SALSA applications and theaters communicate with *Name Servers*. 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, however lookup of new actors, as well as actors whose locations have changed and have become unknown, must go through a name server.

To enable remote communication between actors, actors need some way of

Table 4.1: UAN Format

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

uniquely referencing each other. This is doing through *Universal Actor Names (UANs)*. UANs are unique names that identify actors throughout their lifetime. UANs follow the URI syntax and are similar in format to a URL (see Table 4.1).

Universal naming allows an actor to register its UAN at a name server, becoming a *universal actor*. Name servers keep up to date information about the location of each universal actor that has been registered with them. This allows actors running in different SALSA programs and on different theaters to look each other up and communicate. When an actor has a UAN, it can also *migrate* between theaters.

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, however this is a practice that 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 Universal Actor Creation

A universal actor can be created at any desired theater by specifying its UAN using the `at` statement[2]. For instance, one can create a universal actor at the 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" and port 4040, by the following statement:

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

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

4.1.3 Referencing Universal 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 construction with the `new` statement and references from messages, are available in both distributed and concurrent SALSA programming. A reference to an actor can also be obtained by using the `reference` statement by contacting a name server with a known UAN as follows:

```
AddressBook remoteService =
  reference AddressBook(new UAN("uan://nameserver1/id"));
```

Sometimes an actor wants to know its name or location. An actor can get its UAN by the method `getUAN()` and location with the `getHost()` and `getPort()` methods. For example:

```
UAN selfUAN = this.getUAN();
String selfHost = this.getHost();
int selfPort = this.getPort();
```

4.1.4 Migration

As mentioned before, only universal actors can migrate. Sending the message `migrate(<host>, <port>)` to an universal actor causes it to migrate seamlessly to the designated location. The name server will be notified to update its entry.

The following example defines the behavior `MigrateSelf`, that migrates the `MigrateSelf` actor to location host1:port1 and then to host2:port2. The `migrate` message takes as argument a `String` specifying the target host and an `int` specifying the target port.

```
module examples;

behavior MigrateSelf {
  void act(String args[]){
    if (args.length != 4) {
      standardOutput<-println(
      "Usage:" +
      "java␣-Duan=<UAN>␣examples.MigrateSelf␣" +
      "<host1>␣<port1>␣<host2>␣<port2>");
      return;
    }
    self<-migrate(args[0], Integer.parseInt(args[1])) @
```

```
    self<-migrate(args[2], Integer.parseInt(args[3]));
  }
}
```

4.2 The World-Wide Computer (WWC) Architecture

The World-Wide Computer (WWC) consists of instances of two different daemons – *theaters* and *name servers*. Theaters are actor execution environments and that host one to many concurrently running actors. They provide services for actor migration and remote communication. Name servers handle discovery and lookup of actors between different theaters.

4.2.1 Theaters

A theater is a daemon that waits for incoming actors and messages and provides various services for their execution such as remote message passing, migration.

One can start a theater as follows:

java -cp salsa<version>.jar:. salsa_lite.wwc.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:. -Dport=4060 salsa_lite.wwc.Theater

4.2.2 UANP and Name Servers

One can start a name server as follows:

java -cp salsa<version>.jar:. salsa_lite.core.naming.NameServer

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

java -cp salsa<version>.jar:. salsa_lite.core.naming.NameServer -p 1256

4.3 Running SALSA Applications on the WWC

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 3.7 at `host1:4040` as follows:

java -cp salsa<version>.jar:. -Dactor_uan=uan://host2:5555/myhelloworld -Dactor_host=host1 -Dactor_port=4040 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 initial host becomes a part of the World-Wide Computer.

4.3.1 AddressBook Example

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.

Services written in the C or Java programming languages use infinite loops to listen for requests. A SALSA service can not use this approach because loops inside a message handler preclude an actor from executing messages in its message box. Actors with UANs are by default services actors and not reclaimed by the SALSA runtime, as they can be looked up at any time from remote hosts by their UAN.

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.addressbook;

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

//  Address book implementation taken from CORBA examples
//  in Sun's Java CORBA Tutorial.
//
//
// AddressBook
//
// This behavior is responsible for implementing
// three message handlers
//
//   * String getEmail ( String email );
//   * String getName ( String name  );
//   * boolean addUser ( String name, String email );
//
//

behavior AddressBook {
  private Hashtable name2email;

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

  // Get the name of this email user
  String getName(String email) {
    System.err.println("getting name by email: " + email);

    if (name2email.contains(email)) {
      // Begin search for that name
      Enumeration e = name2email.keys();

      while(e.hasMoreElements()) {
        String name = (String) e.nextElement();
        String e_mail = (String) name2email.get(name);

        // Match on email?
        if (email.compareTo(e_mail) == 0) {
          return name;
        }
      }
```

```
    }
    return new String("Unknown user");
  }

  // Get the email of this person
  String getEmail(String name) {
    System.err.println("Getting email by name: " + name);

    // If user exists
    if (name2email.containsKey(name)) {
      // Return email address
      return (String) name2email.get(name);
    }

    // User doesn't exist
    return new String("Unknown user");
  }

  // Add a new user to the system, returns success
  boolean addUser (String name, String email) {
    System.err.println("added user: " + name + ", " + email);

    // Is the user already listed
    if (name2email.containsKey(name) ||
       name2email.contains(email)) {
      // If so, return false
      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 -Dactor_uan=<UAN> " +
      "-Dactor_host=<host> " +
      "-Dactor_port=<port> examples.addressbook.AddressBook" );
      return;
    }

    standardOutput<-println("AddressBook at: ") @
    standardOutput<-println("\tuan: " + getUAN()) @
    standardOutput<-println("\thost: " + getHost() + ", " +
             "port: " + getPort());
```

```
  }
}
```

The `AddressBook` actor is bound to the UAN, host and port specified in the command line. This will result in placing the `AddressBook` actor in the designated location and notifying the name 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 using the `reference` statement. 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.addressbook;

behavior AddUser {

  void act(String args[]) {
    if (args.length != 3 ) {
      standardOutput<-println( "Usage: java examples." +
      "addressbook.AddUser" +
      " <AddressBookUAN> <Name> <Email>" );
      return;
    }
    try {
      AddressBook book = reference AddressBook(new UAN(args[0]));
      book<-addUser(args[1], args[2]);
    } catch (Exception e) {
      System.err.println("error creating uan: " + args[0]);
      System.err.println("\texception: " + e);
      e.printStackTrace();
      System.exit(0);
    }
  }
}
```

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`. This section focuses on another type of continuations, *named tokens*.

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 allow token passing continuations greater expressive power, through explicit definition of token passing. For example, a token-passing continuation statement can be rewritten using named tokens:

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

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

Named 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 written using only 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 a concurrent, recursive Fibonacci calculation:

```
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) @
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 do not return

(ie., they are all a `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 2.0.0*alpha* provides three message properties that can be used with message sending: `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: `delay`

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

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

5.3.2 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 processed by Actor `a` after Actors `a` and `b` have migrated to the same theater:

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

5.3.3 Property: `delayWaitfor`

SALSA 2.0.0*alpha* 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 only be processed by Actor `a` after 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("europa.cs.rpi.edu", 4040);
token y = b<-migrate("europa.cs.rpi.edu", 4040);
a<-compare(b) : delayWaitfor(1000, x, y);
```

LITERATURE CITED

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

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

[3] 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.

APPENDIX A
Starting SALSA Applications

In SALSA 2.0.0*alpha*, theaters have been redesigned to be highly extensible. Currently, three seperate theater implementations are available depending on the needs of the application. By default, SALSA applications are run within a WWC Theater, which enables incoming and outgoing communication to other theaters. However, if a SALSA application will not be using any remote communication, it can be run within a local theater (one without any remote communication services) for improved performance. Additionally, garbage collection via reference counting has been implemented for local theaters and can be turned on if the SALSA actors have been compiled using the `"-Denable_gc"` flag. For example, to compile the `HelloWorld` actor for use in a local theater with garbage collection:

```
java -Denable_gc salsa_lite.core.compiler.SalsaCompiler HelloWorld.java
```

Remote garbage collection has not yet been implemented in SALSA 2.0.0*alpha*.

The following system properties can be used when running a SALSA program:

`-Dlocal`: This will run the program in a local theater without any remote communication services enabled.

`-Dlocal_gc`: If the SALSA actors have been compiled using the `"-Denable_gc"` flag, they will be run in a local theater with garbage collection.

`-Dwwc`: This will run the program in a WWC theater (this is the default theater).

`-Dport=<port>`: This will start a WWC theater listening on the specified port

`-Dactor_uan=<actor_uan>`: This will start the SALSA actor with the specified UAN.

`-Dactor_host=<host>`: This will start the SALSA actor at a theater with the specified host (if -Dactor_port is not specified, 4040 is the default).

`-Dactor_port=<port>`: This will start the SALSA actor at a theater with the specified port (-Dactor_host must be specified).

System properties are set between the `java` command and the actor class, for example:

```
java -Dwwc -Dport=4141 -Dactor_uan="uan://nameserver:3131/helloworld"
examples.helloworld.HelloWorld
```

Will start a `HelloWorld` actor in a WWC theater listening on port 4141, and register the actor with the UAN, `"uan://nameserver:3131/helloworld"`.

APPENDIX B
SALSA Theater Options

There are also many options for running and extending SALSA theaters. In SALSA 2.0.0*alpha*, the concurrency of a theater is fixed at runtime and is independent of the number of actors running on it. This allows SALSA programmers to improve performance of applications by setting the number of threads in a theater to a quantity suitable to the hardware the theater is running on. Additionally, SALSA 2.0.0*alpha* uses a set number of ID generation objects to reduce the overhead in generating unique actor IDs. The following system properties can be set when running either a SALSA program or a theater:

`-Dtsthreads<number of transport service threads>`: This specifies the number of threads used by the transport service to send messages to remote theaters. Connections are left open between remote theaters, however only a number of actors or messages equal to the number of service threads can be in transit at any one time. Increasing this value may improve the performance of applications that send many messages or actors simultaneously.

`-Dnstages=<number of stages>`: This specifies the number of threads used to process messages by actors. Actors run on a stage, with only one actor processing a message on a stage at a time. Typically setting this value large than the number of processors or cores available on a computer will improve performance.

`-Dnidgens=<number of ID generators>`: This specifies the number of ID generation objects used (the default is 50). Increasing this number may improve performance with applications that generate large amounts of actors.

B.1 Extending Theaters

SALSA developers may also extend the current SALSA theaters with their own services. SALSA theaters use the following services:

`ErrorService`: This service specifies the `standardError` actor, which should write error messages.

`OutputService`: This service specifies the `standardOutput` actor, which should write output messages.

`ReceptionService`: This service handles incoming communication from other theaters.

`TransportService`: This service handles outgoing communication to other theaters.

`NamingService`: This service handles communication with name servers.

`StageService`: This service handles the execution of actors and message processing.

`GarbageService`: This service handles garbage collection.

The interfaces for these services can be found in the `salsa/core/services/` directory. For examples of their implementation, local theaters use the following implementations of these services, which can be set as a group using the `"-Dlocal"` or `"-Dlocal_gc"` system properties when running a SALSA actor, and then overwritten individually by changing the given system properties with a new class.

- `-Derror="salsa_lite.local.LocalErrorService"`
- `-Doutput="salsa_lite.local.LocalOutputService"`
- `-Dreception="salsa_lite.local.LocalReceptionService"`
- `-Dtransport=null`
- `-Dnaming=null`
- `-Dstage="salsa_lite.local.LocalStageService"`
- `-Dgarbage="salsa_lite.local_gc.LocalGarbageService"` – Only if `"-Dlocal_gc"` is specified, null otherwise.

And WWC theaters use the following implementations of the theater services, which are used by a SALSA actor or WWC theater by default. These can also be set as a group using the `"-Dwwc"` system property when running a SALSA actor and then overwritten individually.

- `-Derror="salsa_lite.wwc.io.WWCErrorService"`
- `-Doutput="salsa_lite.wwc.io.WWCOutputService"`
- `-Dreception="salsa_lite.wwc.reception.WWCReceptionService"`
- `-Dtransport="salsa_lite.wwc.transport.WWCTransportService"`
- `-Dnaming="salsa_lite.wwc.naming.WWCNamingService"`
- `-Dstage="salsa_lite.wwc.stage.WWCStageService"`
- `-Dgarbage=null`

APPENDIX C
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.

```
usage:

     java salsa_lite.core.naming.NameServer

     java salsa_lite.core.naming.NameServer -h

     java salsa_lite.core.naming.NameServer -v

     java salsa_lite.core.naming.NameServer -p portNumber

options:

     -h:  Print this message.

     -v:  Print version number.

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

APPENDIX D
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 E
Learning SALSA by Example Code

One can download the SALSA source code at

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

which includes several good examples.

E.1 Package `examples`

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

- `examples.addressbook`: The address book example shown in Section 4.3.
- `examples.fibonacci`: The recursive `fibonacci` application.
- `examples.helloworld`: The `HelloWorld` example.
- `examples.migration`: An example to show how to migrate an actor.
- `examples.trap`: This numerical application approximates the integral of a function over an interval [a, b] by using the trapezoidal approximation.
- `examples.matrixmultiply`: A matrix maultiply example in SALSA.
- `examples.mergesort`: An implementation of merge sort algorithm in SALSA.

E.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 F
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 Gul Agha, has contributions to the naming service of the very early SALSA. 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 the early version of SALSA at RPI. They worked together from 2001 until Abe Stephens graduated. Robin Toll has helped the early SALSA compiler during this period of time. From 2003 to 2004, Travis Desell has committed a thorough change on the early SALSA, which then became the foundation of the current version of SALSA. 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 has major contributions to the development of the communication layer of the latest SALSA. Wei-Jen Wang has been a major developer since 2004. He made another major change on SALSA in 2004 and released it in 2005. 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 2008, Travis Desell completely rewrote SALSA, for version 2.0. This rewrite focused on improving the performance and lowering the overhead of SALSA applications, while eliminating deadlocks and allowing for more flexibility in extending SALSA theaters.

APPENDIX G
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>)*
```