Netlogo, An Open Simulation Environment

Benoit Gaudou, Christophe Lang, Marilleau Nicolas, Guilhelm Savin, Sébastien Rey-Coyrehourcq, Jean-Marc Nicod

1

NetLogo, an Open Simulation Environment

1.1. Introduction to extensions in NetLogo

NetLogo is a generic simulation environment in the sense that it was not designed with any specific domain of application in mind. NetLogo offers a wide range of features and generic operators to its users. Additionally, to make up for any missing features, NetLogo is compatible with other platforms and libraries, as we will demonstrate throughout this book.

There is a vast library of extensions available to users, allowing them to integrate additional functionality that is not present in the native version of NetLogo, but which might nonetheless be necessary for the development of a given model. An official library of extensions is available on the official NetLogo Website. We will explore some of these extensions later in this chapter. But many modelers have also developed their own extensions to tackle specific problems that are of interest to them. These extensions are developed with an open Java API. We will discuss this in more detail in section 1.2.

Conversely, NetLogo can also be called and controlled by other programs, such as OpenMole[1], Python[2] and R[3]. To do this, NetLogo provides a Java API

Chapter written by Benoit GAUDOU, Christophe LANG, Nicolas MARILLEAU, Guilhelm SAVIN, Sébastien REY COYREHOURCQ and Jean-Marc NICOD.

1 http://www.openmole.org/.

2 https://www.pythong.org/.

3 https://www.r-project.org/.

that allows models to be loaded, executed and gives access to their variables and methods. The usage of this API is presented in detail in section 1.3.

1.1.1. *Examples of typical NetLogo extensions*

There are many different types of extension. The GitHub page of the NetLogo platform[4] gives one possible list of examples. This list distinguishes between internally developed extensions, which are included with the platform (e.g. GIS or network), and extensions developed by the community, which have to be installed manually (section 1.1.2). In Chapter 3 of Volume 1 [BAN 15a], we presented a number of these extensions (GIS and network) to showcase some of additional functionnalities of NetLogo.

Some of these extensions include language extensions, which allow the modeler to manipulate more complex object types than those natively present in NetLogo. Indeed, the language of the platform has relatively few complex structures (unlike most programming languages) and primitives for manipulating them. For instance, the `array`, `table` and `matrix` extensions are now included with NetLogo. However, extensions such as `string` and `file` are external.

More generally, the functionalities of the NetLogo language can be augmented with a wide range of extensions, for example to achieve better network management (`network` and `nw`), to provide more primitives for network analysis (additional metrics and indicators) or to integrate geographical data represented in vector form into NetLogo models (`GIS`). This is not an easy task, but this is absolutely necessary, as NetLogo models are natively based on a grid-based discrete environment. As another example, the SQL extension allows models to interact with a database by sending SQL-formatted queries and receiving data in response. Finally, the `sound` and `MIDI` extensions allow sounds to be integrated into NetLogo models.

There are a number of extensions that enable NetLogo to interact with other tools. This interaction can take various different forms: it might simply involve reading or writing files that are compatible with a third-party application. For example, it is possible to process image files (*bitmap*), tabular data (*csv*), Java

4 https://github.com/Netlogo/Netlogo/wiki/Extensions.

system properties (*Props*), POV rays (*RayTracing*), VRML (*VRML*), NetCDF (*NetCDF*), etc. This allows modelers with different backgrounds to increase the realism of their simulations by exploiting real data in useful formats.

Other extensions allow deeper forms of interaction by directly integrating third-party functionality into NetLogo; for this kind of interaction, NetLogo must be able to connect with another application to send requests and retrieve results, such as Matlab (*MATLAB*), Prolog (*NetProLogo*), IODA [KUB 11] (*IODA*) and Graphstream[5] (*gs*).

For example, the NetLogo language can be extended with primitives allowing it to benefit from the scientific calculation tool R [THI 10], and in particular to call R functions from within a NetLogo model. The R extension for NetLogo can be downloaded on the Netlogo-R-Extension Website[6]. It fulfills the task of communicating data between the two tools, and in particular performs type conversions from one language to the other. A working installation of R is required. There is also a reverse extension that allows NetLogo to be called from R, known as RNetlogo[7].

Finally, there are several extensions enabling NetLogo to connect with various types of hardware (sensors, actuators, etc.). Examples include the extensions *Arduino* (microcontroller), *GoGo* (sensors) and *wiimote* (game controller with an accelerometer).

1.1.2. *Installing and using extensions in models*

The extensions used by NetLogo are located in the `extensions` folder in the NetLogo root directory. Each extension has its own separate folder.

To use an extension that is not included with NetLogo, it has to first be downloaded (usually as an archive file), unzipped and installed. Installation is extremely simple – the folder extracted from the archive has to copied into the `extensions` folder. The folder name must be the same as the name of the extension.

5 http://graphstream-project.org.

6 http://r-ext.sourceforge.net/.

7 http://cran.r-project.org/web/packages/RNetlogo/index.html.

To use the primitives provided by the extension in a NetLogo model, we have to first declare the extensions used by the model:

```
1 extensions [extension_name1 extension_name2]
```

To use a primitive defined in this extension, we simply call it by its name in the model prefixed by the name of the extension:

```
extension_name1:primitive_name parameters
```

For example, to extend NetLogo functionality to include additional time management functions, we can use the `time` extension, also known as the NetLogo Time Extension[8]. Once unzipped, the archive produces the folder `time-1.3.0` (which corresponds to Version 1.3.0) containing the source files of the extension, documentation, example models and `.jar` files (Java archives). To use it in a NetLogo model, we have to simply rename this folder as `time` instead of `time-1.3.0`, and copy it into the `extensions` folder[9].

In order to use this extension, we declare it in the model:

```
extensions [time]
```

We can now use the primitives defined by this extension using the prefix `time:`. For example, `time` allows us to create a date object (with the `create` primitive) and to manipulate it, in order to retrieve the day, month or year of this date (`get` primitive), to perform operations on dates (`plus` primitive) or to compare dates (`is-before`, `is-after` and `is-equal` primitives)[10]:

8 https://github.com/colinsheppard/time/.

9 In fact, for this extension (and most other extensions), only the `.jar` files (`time.jar` and `joda-time-2.2.jar`) are required. These contain the definitions of the new primitives.

10 Other examples are included with the extension.

```
    let my_date time:create "2016-02-28 17:28:07.777"
    print time:get "month" my_date
    print time:plus my_date 1.0 "year"
4   print time:is-before (time:create "2016-01-01")
                         (time:create "2018-01-01")
```

1.2. Designing and developing extensions

A project that allows minimal extensions to be easily compiled in Scala with SBT or Maven can be found within the GitHub repository Netlogo-extension-build-example[11].

1.2.1. *Environment for compiling extensions*

1.2.1.1. *Maven and Java*

Maven is a software build management system developed by the Apache Foundation. It works by defining and using Project Object Model (POM) files, which contain a set of instructions for successfully building the program.

The first step is to install Maven on the workstation.

Maven works by relying on repositories (local or online) containing the dependencies that must be downloaded at compilation. Since March 2016, NetLogo uses the online repository Bintray[12], and it is no longer necessary to manually add the `Netlogo.jar` file to your local repository. Development versions (NetLogo 6.0) are already available from the online repository. However, in this book, we will use the stable Version 5.3.1.

As a reminder, since only NetLogo versions 5.3 and later are available online, we will recall how to register a `.jar` file in the local repository of your device. Follow the instructions given in the documentation[13]. Once you are in the NetLogo `/app/` directory that you wish to install (replace `X.X` by the version number), you can run the following command from the command line to install the `.jar` file in the local Maven repository:

11 https://github.com/Spatial-ABM-with-Netlogo/Chapitre-A.

12 https://bintray.com/netlogo/NetLogo-JVM/netlogo.

13 https://maven.apache.org/guides/mini/guide-3rd-party-jars-local.html.

```
mvn install:install-file -Dfile=Netlogo.jar -DgroupId=org.nlogo
↪  -DartifactId=Netlogo -Dversion=X.X -Dpackaging=jar
```

For the development of most extensions, the `Netlogo.jar` file and the `scala-library` dependency will be enough. Other extensions that use specific NetLogo functions may, however, require other dependencies, most of which will be contained in the `.jar` files in the NetLogo `/app/` directory.

Users who wish to develop extensions in Scala or Java will need to pay attention to the version of NetLogo. NetLogo 5.3 is only compatible with Scala versions 2.9.x, and NetLogo 5.3.1 is only compatible with Scala versions 2.10.x. In both cases, the Java version needs to be 7.x. We have to wait for the next version of NetLogo before we can use Scala 2.10.x with Java 8.x.

The simplest solution for compiling an extension is based on the modification of the `JavaHOME` variable used by Maven. In Linux, simply type the following command in the terminal before calling the `mvn` command:

```
export Java_HOME=/path/to/jdk7/
```

The `pom.xml` file and the Maven project that can be used to compile a minimal Java extension may be obtained from the `Java-plugin-Netlogo-maven` project on GitHub. This project can be directly opened in the software workbench (or IDE) Java IntelliJ.

The `pom.xml` file contains the list of dependencies to be loaded locally or from the Maven repositories, and also the configuration of two plugins: `maven-compiler-plugin` and `maven-jar-plugin`. The first plugin allows the Java sources to be compiled by running the Maven command `mvn compile` in the project root directory. The second one allows the `.jar` to be created in the /target repository by running the Maven command `mvn install`.

1.2.1.2. *SBT and Scala*

Simple Build Tool (SBT) is a build system similar to Maven, but which is commonly used to compile sources written in Scala. NetLogo is compiled with SBT, since its most recent versions are written in Scala.

Unlike Maven, which manages dependencies by using `pom.xml` files, SBT uses files written in Scala to determine the structure of the project and its dependencies. One of the most important such files is `build.sbt`, which may be found in the project root directory. SBT uses the same online repositories as Maven to download the right dependencies for compiling and packaging extensions in development. The primary SBT file, named `build.sbt`, uses Netlogo-Extension-Plugin[14], which automatically downloads the right `Netlogo.jar` file and provides a simplified interface for packaging extensions.

Since NetLogo version 5.3.1 can only be compiled with Java 7, we must tell SBT where to find this version on your device: `sbt -Java-home /path/to/Java/home`.

Even though it may not be immediately useful in our case, note that it is possible to tell SBT which version of Java it should use to run Java programs by adding the following lines to `build.sbt`:

```
fork in run := true
JavaHome in run := Some(file("/path/to/Java/home/"))
```

Finally, the extension is compiled by running the `sbt compile` command, and the `.jar` is built by calling `sbt package`.

1.2.2. *Notes on type conversion between NetLogo and Java/Scala*

All numeric variables used in NetLogo extensions must be converted into the `Double` type, as this is the only numeric type accepted by NetLogo. There are tools available to developers for converting from Java/Scala to NetLogo. But conversion in the other direction is not so easy.

Type conversion from NetLogo to Java or Scala is more tricky, in particular for `LogoList` lists. Since NetLogo lists are able to contain different types, it is impossible to know in advance which types of objects are contained in the list variable. The only solution is to carefully *typecast*[15] each element in the list before performing any operations.

14 https://github.com/Netlogo/Netlogo-Extension-Plugin.

15 Also known as type coercion, this means converting a variable from one type to another.

1.2.2.1. *Java*

From Java to NetLogo: numeric type conversion can be performed with the command `Double.valueOf(valueToConvert)`. This command wraps the `double` variables in a `Double` class, which is understood by NetLogo.

From NetLogo to Java: handling lists requires generous use of `try/catch` blocks to detect and convert the types of objects contained in the list. We will illustrate this conversion when we present the code of the primitive for calculating the average of a list of values passed as parameters.

1.2.2.2. *Scala*

Support for automatic type conversion from Scala to NetLogo has been added by the developers of NetLogo *via* the following import, which can be added to the start of a program: `import org.nlogo.api.ScalaConversions.`

Calling the function `.toLogoObject` on any Scala data type (`Boolean`, `Float`, `Character`, `Short`, `Int`, `Float`, `Long`, `Double`, `Byte`, `Seq`) initiates the conversion process, which automatically returns a type that is compatible with NetLogo.

With SBT, it is possible to initialize an *interactive console*, which can access the set of dependencies included in the project. This interactive mode allows us to enter commands directly into a terminal without having to compile or package the extension first.

In the root directory, simply type the command `sbt console` into a terminal, followed by the following commands:

```
import org.nlogo.api.ScalaConversions._

val myIntValue:Int = 5
myIntValue.toLogoObject // return Java.lang.Double
val myFloatValue:Float = 2.2
myDoubleValue.toLogoObject // return Java.lang.Double
val myScalaList = Seq(2,3,8)
myScalaList.toLogoObject // return org.nlogo.api.LogoList
```

1.2.3. *Commentary of an example extension*

The `.jar` file created by Maven or SBT after executing the second command includes a valid manifest file, which is usually named `my-extension.jar`. This should be copied into the directory /app/extensions/my-extension in NetLogo 5.3, and then called in the program with the following code: `extensions [my-extension]`.

The extension named `my-extension`, which we compiled in the previous few sections, allows us to do three things:

– return the sequence of characters “hello world” (`print-message`);

– return the average of the numbers passed as parameters (`get-mean`);

– construct a list of random numbers with length equal to the variable passed as an argument to the primitive (`build-a-random-list`).

Calling the following command in the NetLogo observer returns the character sequence “hello world”: `print my-extension:print-message`.

Calling the following command in the NetLogo observer returns the average of the list of numbers passed as parameters: `print my-extension:get-mean list (10, 12, 15)`.

Calling the following command in the NetLogo observer returns a list of five random elements: `print my-extension:build-a-random-list 5`.

The extension code is given in Java in the below examples. Equivalent code in Scala is also available from the GitHub repository containing the examples for compilation with Maven and SBT.

1.2.4. *Minimum content of an extension*

In order for NetLogo to be capable of loading an extension, the `.jar` file should contain two elements: a `manifest` containing NetLogo-specific properties, and a class implementing the `ClassManager` interface of the `org.nlogo.api` package. The `.jar` file must contain all classes associated with the extension.

If additional software libraries are used, they can be placed in the extension folder in the NetLogo `extensions` directory.

1.2.4.1. *Manifest*

The `manifest` must contain the following three properties:

– `Extension-Name`, the name of the extension;

– `Class-Manager`, the extension class implementing `ClassManager`;

– `Netlogo-Extension-API-version`, the version of the NetLogo API used by the extension.

If an extension named `test` has `ClassManager` implemented by the class `MyExtension` in the package `org.test` and uses the NetLogo API 5.1, it needs to have the following `Manifest`:

```
Manifest-Version: 1.0
Extension-Name: test
Class-Manager: org.test.MyExtension
Netlogo-Extension-API-Version: 5.3
```

1.2.4.2. *The ClassManager*

To develop the `DefaultClassManager` of the extension, we can extend the `DefaultClassManager` class of `org.nlogo.api` to reduce the list of methods that we must implement for `load(PrimitiveManager)`. Passing the `PrimitiveManager` object as a parameter allows us to add new primitives (commands or reporters).

Consider the following minimal example of the extension `MyExtension`:

```
1 package org.test;

  import org.nlogo.api.DefaultClassManager;
  import org.nlogo.api.ExtensionException;
  import org.nlogo.api.PrimitiveManager;
6
  public class MyExtension extends DefaultClassManager {
      @Override
      public void load(PrimitiveManager primitiveManager) throws
           ExtensionException {
        //Declare primitives
11    }
  }
```

To define the call to the three primitives, we replace the comment in the `load` function with the following code:

```
primitiveManager.addPrimitive("print-message", new MyMessage());
primitiveManager.addPrimitive("get-mean", new ComputeMean());
primitiveManager.addPrimitive("build-a-random-list", new BuildRandomList());
```

1.2.5. *Snapshot of a primitive*

The three primitives are placed in three separate Java files, each of which contains the description of a primitive:

```

//BuildRandomList.Java
public class BuildRandomList extends DefaultReporter { ... }

//CountCharacter.Java
public class CountCharacter extends DefaultReporter { ... }

//MyMessage.Java
public class MyMessage extends DefaultReporter { ... }
```

These three classes extend the `DefaultReporter` interface, and so have to implement the following functions:

```
public Syntax getSyntax() {...}

public Object report(Argument args[], Context context) throws
     ExtensionException, LogoException {...}
```

In the next sections, we describe the way that these functions are called, and the results that they return.

1.2.5.1. *Displaying "hello world"*

These primitives do not take any input parameter, and simply return a message. The function `Syntax.reporterSyntax()` therefore only has one

single argument, which indicates the expected type to be returned. Since “hello world” has the type of a string, we use the code `Syntax.StringType()`.

```
public Syntax getSyntax() {
    return Syntax.reporterSyntax(Syntax.StringType());
}
```

Other types can be returned, such as `Syntax.NumberType()`, which indicates that a numerical value should be returned, `Syntax.ListType()`, indicating that a list is expected, or any other NetLogo object that can be manipulated by an extension, as shown by the list of functions defined in the `Syntax` object: `BooleanType()`, `AgentsetType()`, `TurtleType()`, `PatchType()`, `LinkType()`...

1.2.5.2. *Return the average value of an array of variable size*

We can define the call to this primitive in two different ways, either by using the syntax `Syntax.NumberType() | Syntax.RepeatableType()` to define a repeatable number, or by directly using a variable-size list `Syntax.ListType()`. Finally, it should also be noted that we can specify as many `Syntax.typeName` values as the number of arguments that we wish to be returned when we call the primitive. Thus, `new int[]{Syntax.NumberType() | Syntax.NumberType(), Syntax.StringType()}` indicates a primitive that takes three input arguments, i.e. two numbers and one sequence of characters.

```
// Way 1 using NumberType and RepeatableType
 public Syntax getSyntax() {
    return Syntax.reporterSyntax(new int[]{Syntax.NumberType() |
         Syntax.RepeatableType()},Syntax.NumberType());
}

// or Way 2 using ListType
public Syntax getSyntax() {
    return Syntax.reporterSyntax(new
         int[]{Syntax.ListType()},Syntax.NumberType());
}
```

In the first case, the primitive is called with: `print (my-extension:get-mean 0.0 5.0 10.0)`, and in the second case with: `print my-extension:get-mean list 0.0 5.0 10.0` or `print my-extension:get-mean [0.0 5.0 10.0]`. The second syntax, shown below, has the advantage of being more easily understood by beginners, but requires developers to check the content of the table before performing any operations. As discussed in the previous section on type conversion, the methods for retrieving the content of the `LogoList` variables return a collection of `Object` variables that need to be tested[16].

Arguments should always be recovered using the "safe methods" provided by the developers of NetLogo, which we wrapped into a method below.

```
private LogoList getListOrNull(Argument args[]) throws ExtensionException,
      LogoException {
         try {
             return args[0].getList();
         } catch (LogoException e) {
             return null;
         }
     }
```

The operation that converts `LogoList logoListNumbers` into `ArrayList<Double> numbers` is defined as follows in our code:

```
public Object report(Argument args[], Context context) throws
      ExtensionException, LogoException {

     final LogoList logoListNumbers = getListOrNull(args);

     // LogoList return an array of Object, so we need to cast to
          ArrayList[Double]
     Double[] logoDouble = null;
     try {
         Object[] objectArray = logoListNumbers.toArray();
         logoDouble = Arrays.copyOf(objectArray, objectArray.length,
              Double[].class);
     }catch (ClassCastException e){
         System.out.println("Cast Error, only numbers are supported here");
```

16 Both versions of the code are available from the GitHub repository online.

```
        }

        ArrayList<Double> numbers = new
             ArrayList<Double>(Arrays.asList(logoDouble));
18
        return average(numbers);

        }
```

Other conversion methods probably exist, but this topic is currently little documented on the official website. Note that in this last example wrapping the variable returned by the function `average(numbers)` in a `Double` is not strictly necessary, as Java can perform "autoboxing" in certain conditions: http://docs.oracle.com/Javase/tutorial/Java/data/autoboxing.html.

1.2.5.3. *Construct and return a table of variable size*

The first argument `new int[]{Syntax.NumberType()}` of the function `Syntax.reporterSyntax()` states that the primitive expects an integer input. The second argument `Syntax.ListType()` tells NetLogo that a list will be returned.

```
public Syntax getSyntax() {
    return Syntax.reporterSyntax(new
         int[]{Syntax.NumberType()},Syntax.ListType());
    }
```

Lists can be constructed using a "builder" provided by the developers:

```
LogoListBuilder list = new LogoListBuilder();
```

Adding a `Double` (not double) can be achieved with a simple loop as a function of the value `n`, assigned by calling the `getIntValue()` method of the class `Argument` on the table `args[0]`:

```
for (int i = 0; i < n; i++) {
       list.add(Double.valueOf(r.nextDouble()));
    }
```

1.2.6. *Future versions of the NetLogo API*

Although the NetLogo API has been relatively stable for several versions, it is expected to change with version 6.0.

Here are a couple of changes that have already been confirmed for the future version of the API:

– multiple classes will be renamed or reorganized in future. For example, a new package `org.nlogo.core` already uses classes from `org.nlogo.api`, `org.nlogo.nvm`, and `org.nlogo.agent`;

– `DefaultReporter` and `DefaultCommand` will be removed, and `org.nlogo.api.Reporter` and `org.nlogo.api.Command` will become easier to extend.

Since this version is still in development, more information can be found on the webpage dedicated to the transition[17].

Help will be available on the various channels of communication used by the developers of Netlogo: gitter[18], GitHub[19] and the NetLogo wiki, which details the extensions API[20], the discussion group[21], and the StackOverflow[22] website under the NetLogo tag.

1.2.7. *Extending the graphical interface*

The NetLogo API does not currently allow the development of dedicated graphics widgets. However, it is possible to add a new tab to the interface and directly manipulate its AWT/Swing canvas. We will use the GRAPHSTREAM[23] software library, which allows graphs to be dynamically manipulated. The purpose of the extension is to display a graph showing the set of turtles and the links between them.

17 https://github.com/Netlogo/Netlogo/wiki/Hexy-Extension-Transition-Guide.
18 https://gitter.im/Netlogo/.
19 https://github.com/Netlogo/Netlogo/issues.
20 https://github.com/Netlogo/Netlogo/wiki/Extensions-API.
21 https://groups.google.com/forum/#!forum/Netlogo-devel.
22 http://stackoverflow.com/questions/tagged/Netlogo.
23 http://graphstream-project.org

The following code provides the basic structure of the extension. We will later show how to develop one part of the missing content. The code is available in full on GitHub[24].

```
import org.nlogo.api.*;
import org.graphstream.graph.*;

public class GSExtension extends DefaultClassManager {
  protected Graph graph;
  protected ExtensionContext ctx;

  public Graph getGraph() { return graph; }

  public ExtensionContext getContext() { return ctx; }

  public void load(PrimitiveManager manager) throws ExtensionException {
    manager.addPrimitive("init", new DefaultCommand() {
      public void perform(Argument[] arg0, Context arg1) throws
       ↪  ExtensionException, LogoException {
        GSExtension.this.init((ExtensionContext) arg1);
      }
    });
  }

  public void init(ExtensionContext ctx) {
    this.ctx = ctx;
    this.graph = new AdjacencyListGraph("netlogo");
    this.graph.addSink(new GSNetLogoSink(this));

    addTab();
  }

  protected void addTab() { ... }
}
```

1.2.7.1. *NetLogo/GraphStream connection*

In a simulation, the NetLogo model and the graph coexist separately, and consistency needs to be maintained between them. Modifications affecting the NetLogo model must therefore update the graph, and *vice versa*. The `GSNetLogoSink` class of the extension is dedicated to managing this connection.

24 https://github.com/graphstream/gs-netlogo.

For the connection from NetLogo to the graph, we will use the functionality provided by the `NetLogoListener` interface of the API, which mainly consists of information about events that occur at the NetLogo interface. The method that we are interested in, `tickCounterChanged(double)`, informs us when the system undergoes a new iteration. We can therefore create events to describe the corresponding changes in the graph at these moments.

To "listen" to changes in the graph and propagate these changes to the NetLogo model, the `GSNetLogoSink` class implements the `Sink` interface of GraphStream, which connects to the graph.

`GSNetLogoSink` is structured as follows:

```
import org.nlogo.api.NetLogoAdapter;
import org.graphstream.stream.Sink;

public class GSNetLogoSink extends NetLogoAdapter implements Sink {
  protected World world;
  protected GSExtension ext;

  public GSNetLogoSink(GSExtension ext) {
    this.ext = ext;
  }

  // NetLogoListener

  public void tickCounterChanged(double arg0) { ... }

  // Sink

  public void nodeAdded(String sourceId, long timeId, String nodeId) {
   ↪  ... }

  public void nodeRemoved(String sourceId, long timeId, String nodeId) {
   ↪  ... }

  public void edgeAdded(String sourceId, long timeId, String edgeId,
   ↪  String fromId, String toId, boolean directed) { ... }

  public void edgeRemoved(String sourceId, long timeId, String edgeId) {
   ↪  ... }

  // ... other methods of Sink not used here
}
```

To establish the connection between NetLogo and GraphStream, we must create a shared procedure for identifying agents (nodes) and their connections (edges). We assume that these objects are characterized by a sequence of characters of the form `breedName.agentNumber`. We can therefore add a method that retrieves the identifier of a `Turtle` object:

```
public String getTurtleId(Turtle t) {
  return String.format("%s.%d", t.getBreed().printName(), t.id());
}
```

The `tickCounterChanged(double)` method needs to contain code allowing the NetLogo model to be compared with the contents of the graph. To do this, we need to iterate over the agents and their existing connections. This iteration is provided by the `World` object and its methods `turtles()` and `links()`. A minimal version of the function might look like this:

```
public void tickCounterChanged(double arg0) {
    Collection<Node> nodes = new HashSet<Node>();

    for (Agent a : world.turtles().agents()) {
        String nodeId = getTurtleId((Turtle) a);
        Node = ext.getGraph().getNode(nodeId);

        if (node == null)
            node = ext.getGraph().addNode(nodeId);

        nodes.add(node);
    }

    // Remove non-existent nodes
    Iterator<Node> itNodes = ext.getGraph().getNodeIterator();

    while (it.hasNext()) {
        Node n = it.next();

        if (!nodes.contains(n))
            it.remove();
    }

    Collection<Edge> edges = new HashSet<Edge>();

    for (Agent a : world.links().agents()) {
```

```
        Link l = (Link) a;
        String edgeId = getLinkId(l);
        Edge edge = ext.getGraph().getEdge(edgeId);

        if (edge == null)
            edge = ext.getGraph().addEdge(edgeId, getTurtleId(l.end1())
             ↪  getTurtleId(l.end2()), l.isDirectedLink());

        edges.add(edge);
    }

    // Remove non-existent edges
    Iterator<Edge> itEdges = ext.getGraph().getEdgeIterator();

    while (it.hasNext()) {
        Edge e = it.next();

        if (!edges.contains(e))
            it.remove();
    }
}
```

This is a minimal version of the function. It only updates the model in one direction, from NetLogo to GraphStream. Readers can refer to the project source code for more details.

1.2.7.2. *Creating a new tab*

The final part of the extension adds a new tab to the NetLogo interface on which the graph will be displayed. We will build on the `addTab()` method mentioned earlier. We could also add a separate primitive to make displaying the graph optional.

Tabs are managed by `JTabbedPane` objects (provided by Swing), which can be retrieved through the `App` object: `App.app().tabs()`.

```
protected void addTab() {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            if (v != null)
                v.close();
            World w = ctx.workspace().world();
            v = new Viewer(g,
                    Viewer.ThreadingModel.GRAPH_IN_ANOTHER_THREAD);
```

```
            v.setCloseFramePolicy(Viewer.CloseFramePolicy.HIDE_ONLY);
            v.addDefaultView(false);

            // Resize the graph to fit NetLogo.

            ↪  v.getDefaultView().getCamera().setGraphViewport(w.minPxcor(),
            ↪  w.minPycor(), w.maxPxcor(), w.maxPycor());

            Tabs tabs = App.app().tabs();
            tabs.addTab("GraphStream", v.getDefaultView());
        }
    });
}
```

1.2.8. *Example: the RungeKutta extension*

In this section, we will present a simple but concrete example of an extension in a few lines.

The `rungekutta` extension has been used for epidemiological simulation models in Chapter 3. It includes a `compute-SIR` function taking six `Double` parameters that calculates the evolution of the population stock passed as an argument using a fourth-order Runge–Kutta numerical integration method applied to the SIR equations. These equations and their solutions are described in more detail in Chapter 3.

```
override def getSyntax(): Syntax =
Syntax.reporterSyntax(Array(NumberType, NumberType, NumberType,
 ↪  NumberType, NumberType, NumberType, NumberType, NumberType,
 ↪  NumberType, NumberType, NumberType), ListType)
```

This function expects the following arguments. We will consider the case of an initial population of 100 individuals:

– the population stock S (99 susceptible individuals);

– the population stock I (1 infected individual);

– the population stock R (0 recovered individual);

– the Alpha parameter (rate of recovery I to R = 0.2);

– the Beta parameter (rate of infection S to I = 0.5/100);

– the integration step h.

The following report block passes these arguments to the function that calculates the evolution step:

```
@throws(classOf[ExtensionException])
@throws(classOf[LogoException])
override def report(args: Array[Argument], context: Context) =
{
  val S = args.apply(0).getDoubleValue
  val I = args.apply(1).getDoubleValue
  val R = args.apply(2).getDoubleValue

  val alpha = args.apply(3).getDoubleValue
  val beta = args.apply(4).getDoubleValue
  val h = args.apply(5).getDoubleValue

  rungeKuta4( Array(S,I,R), alpha, beta, h).toLogoList
}
```

Once the arguments have been safely retrieved using `getDoubleValue`, the values are passed to the `rungeKuta4(...)` function, which performs integration then returns an updated table of the SIR stock `Array(ds,di,dr)`. We still need to convert this table into `LogoList` using the automatic conversion function `.toLogoList`.

Here is an example of how this method can be called in NetLogo:

```
show rungeKuta:compute-SIR 99.0 1.0 0.0 0.2 (0.5 / 100)
0.01
```

This call returns the following values, which describe the propagation of infection:

```
[98.99504281588354 1.0029542312962296 0.00200295282023009]
```

The source code of this extension is available in the repository for Chapter 1.

1.3. Using NetLogo from other platforms

The reason for wanting to use a NetLogo model from outside the NetLogo interface, i.e. from another language, becomes apparent as soon as we become interested in automating NetLogo simulations in batch mode (model exploration), coupling different models together (using the output of one model as the input of another), or supporting compatibility with other programs.

The NetLogo interface is useful because it allows rapid and visual development within the context of an "agile"-type approach, which means that the model is developed and then tested in fast-paced cycles, so that any change is tested as soon as it is written. The interface has graphical objects, which can be used to rapidly construct visualizations of models, and a "batch" mode, which can be used to explore models. However, there are also limitations: the graphics layout is fixed and is not necessarily suitable for operational applications. Batch mode is limited to exhaustive model exploration and only allows integrative coupling (see section 2.5). To overcome these constraints, NetLogo provides a Java API (included in the official distribution) that provides an opening to other environments. This API makes it possible to interact with the model without needing a control interface (modify simulation parameters, retrieve results, execute, etc.), as well as to modify the model (by executing NetLogo commands as if they were entered into the interface "command center"). Due to this API, we can interact with NetLogo from programs such as R or languages such as Python. As described below, this interaction unfolds according to a client/server paradigm, in which software clients such as R or Python send requests to a Java server responsible for executing the NetLogo model.

This opening to other programs allows us, for example, to:

– take advantage of all of the features of the host language (R – to exploit all of its statistical primitives, Python – to use numerical calculation libraries (NumPy[25]), Java – to use JavaFX graphics elements and any other useful libraries);

– explore NetLogo models with specific search algorithms (simulated annealing, genetic algorithms or screening), or even combinations of algorithms. Given the amount of time required to execute some simulations

25 http://www.numpy.org.

and the size of the parameter space, exhaustive model exploration is often impossible. Choosing a suitable exploration algorithm is the crucial first step toward obtaining results within a reasonable time frame;

– couple models by channeling the output of one model to other models with essentially zero language-related constraints due to the numerous gateways available in Java for connecting with other languages.

In the next section, we will give a brief description of the approaches that can be used to establish an interface between NetLogo and Java, Python and R. We will omit any specific details relating to the implementation, and simply explore the basic principles of a simple example of coupling.

1.3.1. *Using NetLogo from Java*

To run NetLogo commands from Java, we first need to import a library.

```
import org.nlogo.app.App;
```

In order to run NetLogo programs, the `Netlogo.jar` file needs to be located in one of the directories known to Java (`classpath`). The same is true of the `lib` directory. The latter and the `.jar` file are included within the NetLogo distribution.

We will illustrate how to use Java to run a simulation with the example of a forest fire, `Fire.nlogo`, which is included within the NetLogo distribution.

```
new Runnable() {
    public void run() {
        try {
4         App.app().open("models/Sample Models/Earth Science/" +
              "Fire.nlogo");
        }
        catch(Java.io.IOException ex) {
          ex.printStackTrace();
        }}};
```

From here, it is very simple to run NetLogo commands with *App.app.command()*, as in the below example. In this case, we assign a value to a variable and run `setup`.

```
App.app.command("set number-of-turtles 100");
App.app.command("setup");
```

In this simulation, we need to be able to retrieve the values of the variables stored in Java. This will allow us to explore the model, either by means of sophisticated processing or suitable visual representations. To retrieve the value of a NetLogo variable, we must use the `report` function. In the example below, we display the value of the variable `number-of-turtles`.

```
System.out.println(App.app().report("number-of-turtles"));
```

This example is based on the execution of a NetLogo model in "singleton" mode. In this mode, running multiple simulations in parallel with the same model or different models is not possible. The origin of this limitation lies in the fact that these instructions manipulate static objects. The alternative is to use the notion of "workspace". Each simulation is assigned to a workspace, and so one workspace must be created for each simulation. Each workspace acts as a wrapper for the context of the simulation with which it is associated, saving its attributes, model and execution thread.

To implement this approach, we must create an instance of the `HeadlessWorkspace` class with its default constructor. We can then open a model and execute NetLogo commands. The example below reuses the previous code together with the "Fire" model to execute two simulations in parallel, each with different parameters:

```
import org.nlogo.headless.HeadlessWorkspace;
public class SimulationFire {
    public static void startModel(int nbTurtles, HeadlessWorkspace wSpace) {
        Runnable myThread = new Runnable() {
            public void run() {
                try {
                    wSpace.open("models/Sample Models/Earth Science/" +
                        "Fire.nlogo");
                    wSpace.command("set number-of-turtles " + nbTurtles);
```

```
                    wSpace.command("setup");
                    wSpace.command("repeat 50 [ go ]") ;
                }
                catch(Java.io.IOException ex) {
                    ex.printStackTrace();
                }
            }
        };
        myThread.start();
    }
    public static void main(String[] argv) {
        HeadlessWorkspace simulation1 = HeadlessWorkspace.newInstance();
        HeadlessWorkspace simulation2 = HeadlessWorkspace.newInstance();
        SimulationFire.startModel(100, simulation1) ;
        SimulationFire.startModel(200, simulation2) ;
  }
}
```

More details on these features are available on the NetLogo GitHub page[26]. There are technical subtleties relating to memory consumption, controlling threads and the choice of whether to execute *via* a graphical user interface (GUI) or the command line (Headless).

As well as allowing multiple executions, each with its own context, the Java API provides the key to interoperability with other platforms and development languages. Java has many possibilities and gateways to other languages (C, Python, R, etc.). The NetLogo APIs developed by the community for other programming languages build on these gateways and the native Java API distributed with NetLogo. In the next part of this section, we will consider two examples showing how to use NetLogo from other languages and applications: Python and RNetlogo.

1.3.2. *Using NetLogo from Python*

Python is a programming language widely used in science, and its popularity continues to grow. It has many different libraries, in particular *NumPy*, which is extremely useful for scientific computations and numerical simulations of mathematical models based on differential equations.

With *NumPy*, Python can be viewed as a way of combining mathematical models and multiagent models. Python also proves very useful for dynamically generating experimental protocols and automatically executing them.

26 https://github.com/Netlogo/Netlogo/wiki/Controlling-API.

There is no direct interface between Python and NetLogo. This means that a Java bridge (`JavaGateway`) is required, running as a background task. This bridge receives the NetLogo commands from Python and executes them in the model to obtain the desired results.

One example of such a Java bridge was developed by David Masad, and is available on the webpage *Bad Networking*[27]. We will distribute a modified version that allows multiple simulations to be executed. The sources and executable of the modified version can be downloaded from the GitHub page of this book, at https://github.com/Spatial-ABM-with-Netlogo.

The idea is to run a Java program that will act as a server. The Python program uses a library (package) that allows it to connect to this server. Each time that Python wishes to access NetLogo, it sends a request to the Java program. This program then executes the instructions in the NetLogo model to obtain the desired results.

Thus, executing a NetLogo model form Python unfolds in the following stages:

1) check that the Java bridge is running in the background, otherwise start it up;

2) connect Python to the Java-NetLogo bridge;

3) create as many workspaces as required to run simulations;

4) initialize the simulations with the right parameters;

5) execute the simulations;

6) analyze the results.

The Java server program works according to the above steps. An example of Java and Python code allowing multiple simulations to be simultaneously executed from Python and Java commands is available on the GitHub page of this book.

27 http://davidmasad.com/blog/Netlogo-from-python/http://davidmasad.com/blog/Netlogo-from-python/.

Once you have checked that the `Java-Netlogo` bridge is running properly, you need to create a `JavaGateway` object to establish a connection with the Java server.

```
# import
from py4j.Java_gateway import JavaGateway

# connect Python to the Java-Netlogo bridge
gw = JavaGateway()
bridge = gw.entry_point
```

It is now relatively simple to open an example model by creating a workspace in Java, which is identified by a number in Python:

```
# create one workspace for each simulation we wish to run
sample_models = "/Applications/Netlogo 5.0.2/models/Sample Models/"
forest_fire = "Earth Science/Fire.nlogo"
wks1 = bridge.createWorkspace()
wks2 = bridge.createWorkspace()
bridge.openModel(wks1,sample_models + forest_fire)
bridge.openModel(wks2,sample_models + forest_fire)
```

We can now execute the NetLogo commands by specifying the desired workspace with its number (this procedure is specific to our interface):

```
# Initialize the simulations with the desired parameters
# Parameters of the 1st simulation
bridge.command(wks1,"set density 62")
bridge.command(wks1,"random-seed 0")
bridge.command(wks1,"setup")

# Parameters of the 2nd simulation
bridge.command(wks2,"set density 50")
bridge.command(wks2,"random-seed 2")
bridge.command(wks2,"setup")

# Run the simulations
bridge.command(wks2,"repeat 50 [go]")
bridge.command(wks1,"repeat 50 [go]")

# Process the results of the simulation
...
```

We can now retrieve the values of the variables and display them:

```
...
# Process the results of the simulation
burned_trees = [0]*2
burned_trees[0] = bridge.report(wks1,"burned-trees")
burned_trees[1] = bridge.report(wks2,"burned-trees")

print "the average number of burned tree is: ",
     sum(burned_trees)/float(len(burned_trees))
```

As you can see, this works the same way as Java, except that we can now use the advanced features offered by Python to automatically execute a parametrized series of models and construct all sorts of visual results, for example using libraries such as `matplotlib`.

1.3.3. *Exploring and analyzing models with R*

NetLogo can be called from R using RNetlogo[28]. As presented in the article (https://www.jstatsoft.org/article/view/v058i02), this package is also based on the NetLogo Controller API (in Java), with an additional layer that provides a connection between Java and R. The package can be installed as usual with the following command in R:

```
install.packages("RNetlogo")
```

Once the package is installed, it simply needs to be loaded. This must be performed once for each NetLogo session, and is done with the following function:

```
library("RNetlogo")
```

28 http://rNetlogo.r-forge.r-project.org/.

We can now run NetLogo from R. As was the case for Java, there are two available modes: GUI and headless, i.e. with a graphical interface or from the command line. To launch the GUI mode[29]:

```
install.packages(c("JGR","Deducer","DeducerExtras"))
```

To actually run it, we need to execute the following commands:

```
Sys.setenv(NOAWT=1)
library(JGR)
Sys.unsetenv("NOAWT")
JGR()
```

Next, run NetLogo:

```
nl.path <- "/Applications/Netlogo 5.3.1/Java/"
NLStart(nl.path)
```

We can now control NetLogo with R. For example, we can load a model (or in our example a library of models using the function `NLLoadModel` in R), execute commands on models (with the function `NLCommand`) and modify the values of model parameters or execute individual model methods (for example, `setup` then `go`).

```
;; Load a model
model.path <- file.path("models", "Sample Models", "Earth Science",
    "Fire.nlogo")
model.library.path <- "/Applications/Netlogo 5.3.1/"

NLLoadModel(file.path(model.library.path, model.path))
;; Execute commands on this model
NLCommand("set density 77")
```

29 The path is the path to the folder with the `Netlogo.jar` archive. However, users running Mac OS X or Linux who wish to run NetLogo in GUI mode will need to use JGC. The installation steps are given on this page: http://www.deducer.org/pmwiki/pmwiki.php?n=Main.MacOSXInstallation.

```
8 NLCommand("setup")
  NLCommand("go")
  NLDoCommand(10, "go")
  NLDoCommandWhile("ticks < 200", "go")
```

We also need to be able to retrieve the values of variables. This can be done very simply with the `NLreport()` command.

```
  burned <- NLReport("number-of-turtles")
```

As you can see, running NetLogo from R is relatively simple. This allows you to exploit the powerful calculation functions available in R to explore your models.

1.3.4. *Discussion*

The three examples given above show that accessing and externally controlling a NetLogo model passes through the Java interface distributed with each version of NetLogo.

Java is an expressive language, which makes it possible to develop links to most platforms and languages. The communities of the most commonly used languages in science, such as R and Python, have already created implementations of these links (`RNetlogo` and `Python-Netlogo` respectively). Most of these links seem to follow a common pattern, using primitives to load the NetLogo model and then execute commands written in NetLogo.

If the need should ever arise, the Java interface could definitely be used to develop an *ad-hoc* links with specific functionality. This is the principle behind the Open-Mole platform, a distributed environment for model exploration, which we will discuss in section 5.6.

1.4. Deploying NetLogo models online

As well as the classical NetLogo application installed on personal computers, which comes with a well-stocked library of models, NetLogo also

exists on the Internet. The NetLogo Web[30] application is similar to the desktop version (with somewhat reduced functionality) and can be accessed from a web browser (section 1.4.1). The NetLogo community is extremely active on the Internet and on a number of Websites for publishing models. In the next section, we will present the two best-known of these websites: Modeling Commons[31] dedicated to distributing NetLogo models (section 1.4.2) and the more general-purpose web portal OpenABM[32] for publishing and sharing models (section 1.4.3).

1.4.1. *Netlogo Web*

NetLogo Web (http://www.Netlogoweb.org) is one of the official Internet websites of the NetLogo platform. It gives not only a download link for the desktop application, but also provides access to an online implementation of NetLogo *via* the web browser (see the section presenting NetLogo 1.1).

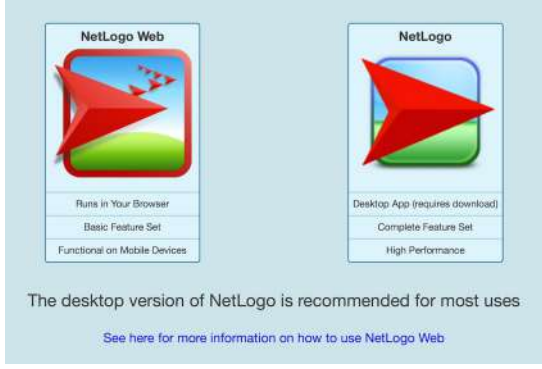

Figure 1.1. *NetLogo Web homepage (March 2016)*

The web version of the NetLogo application allows you to run the models available on the platform, but you can also upload your own models. The usual features are available: the command center, the code editor and information relating to the model description. You can run the application as if it were

30 http://www.Netlogoweb.org/.

31 http://modelingcommons.org/account/login.

32 https://www.openabm.org/.

installed on your personal computer. There are, however, some restrictions, as some features are not yet available. For example, extensions, some language-specific primitives, file reading and writing, 3D models and `BehaviorSpace` are not yet available.

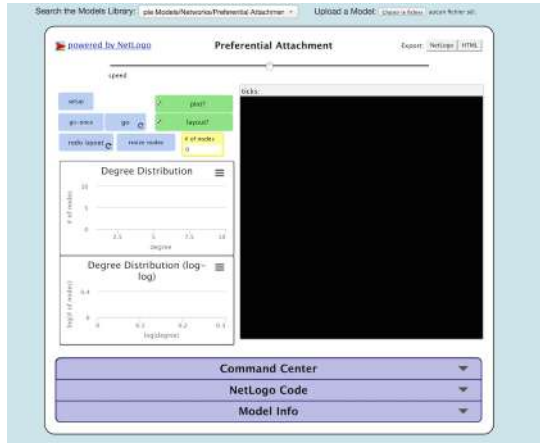

Figure 1.2. *Interface of one of the models available online (March 2016)*

Therefore, if your model uses one of the features unavailable in the online version, you will need to use the desktop version of NetLogo. If not, this platform is a great way of running models without having to install the application (Figure 1.2).

1.4.2. *Modeling Commons*

Modeling Commons[33] (Figure 1.3) is an Internet-based platform for facilitating collaboration between NetLogo modelers. Users can share their models, as well as edit, create and execute them.

The platform also allows users to save their own personal models, and specify the level of visibility. Models can be set to private, or restricted to a certain specific group of users.

33 http://modelingcommons.org.

Figure 1.3. *Modeling Commons homepage (March 2016)*

The first step is to create a user account (using the platform is completely free). This is only required if you wish to save models online, edit them, or comment on existing models. Browsing and downloading public models do not require logging in.

You can now upload a model (Figure 1.4). The platform will ask you to specify the name of the model, provide the filepath on your computer and optionally upload an image as an illustration. The reading and writing permissions of the model must then be selected. The model can be set to either public or private – collaborators can always be added at a later point.

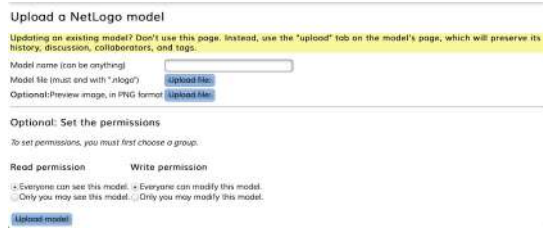

Figure 1.4. *Interface for uploading a model (March 2016)*

Permissions can be changed by adding collaborators. You can also write a model description and browse other related tabs: comments, model execution,

code, version history, auxiliary model files, models belonging to the same family (we will return to this concept later) and an update tab (Figure 1.5). This final tab allows you to upload an updated version of the model.

Figure 1.5. *Model management window (March 2016)*

The History tab gives an overview of all versions, and allows you to download each of them. You can also revert to a previous version.

Finally, the update function allows you to create a child version of the current parent model. Child models are created by performing the classical operation of forking, as is common practice within the programming community. The development of the parent and child models then follows independent paths. However, the relationship between the two remains visible in the History tab.

In summary, the Modeling Commons platform is oriented toward sharing NetLogo models. It provides simple and easy-to-access functionality. The ability to manage groups and organized models into projects helps to develop an effective workflow.

1.4.3. *OpenABM*

OpenABM is a consortium that unites teachers, researchers and professionals with the objective of promoting agent-based modeling. The Website[34] (Figure 1.6) offers a large collection of resources on related topics.

34 http://www.openabm.org.

They already have a very extensive library of community-submitted models. Each model is documented, and the source code is provided.

Figure 1.6. *OpenAbm.org homepage (March 2016)*

But this platform is much more than just a repository of agent models. The *Education* section of the website contains an extensive range of tutorials and documentation for helping to develop models. There are also links to online courses, textbooks and a YouTube channel[35].

A comprehensive selection of resources is available, such as links to development platforms, the websites of modeling-related journals and a well-stocked reading list. There is also a calender of topical events, a forum and job opportunities.

More than anything else, OpenABM is a platform for sharing models and resources on agent-based modeling. As a tool, it is truly comprehensive.

1.5. Conclusion

In this chapter, we showed how the openness of the NetLogo platform holds the key to a great amount of potential.

This is reflected first and foremost in its extensions, which are numerous. We non-exhaustively listed a couple of examples that we consider to be particularly significant, such as *array, R, gis, sound, raytracing*, etc.

35 http://www.youtube.com/user/CoMSESNet/.

We also took the opportunity to explain how to install and use NetLogo extensions, and showed how to personally design a new extension. We gave a list of compilation environments and explained the mandatory content of a minimal extension.

In section 1.3, we examined the possibility of using NetLogo from other platforms. We considered the cases of Java, Python and R. There are other platforms that can make calls to NetLogo, such as OpenMole, an environment dedicated to exploring models using high-performance computations. We will discuss this further in section 5.6.

Finally, we discussed the different ways of deploying models on the Internet. NetLogo Web allows models to be executed online, and Modeling Commons provides additional features to support collaboration. We ended the chapter by presenting OpenABM, a privileged hub for resources on relevant topics.