Debugging in AnyLogic

Nathaniel Osgood

MIT 15.879

May 16, 2012
Model Appropriateness Consideration
• Have we built the right model?
• Have we built the model right?
Have We Built the Right Model?
– This is the province of “validation”
– We can rarely validate the model – only seek to
  • Build confidence
  • Disconfirm it
– This is specific to model purpose
– Here, a lapse is either
  • an oversimplification of the situation
  • An inaccurate “dynamic hypothesis” as to how things work
Have We Built the Model Right?

• Did we implement our planned model logic as we had intended?
  – Did we want one thing and put in place mechanisms that entailed another thing
  – This is the province of classic testing & quality assurance
    • Peer reviews
    • Testing (e.g. Junit)

• Here, a lapse is typically a model “defect” (*bug*)
  – In this lecture, we will be dealing with identifying this sort of defect
Debugging: Faults, Failures

• A “fault” is an underlying defect
• A failure is a visible problem, e.g.
  – Model “crashes”
  – Model will not run
  – Model is reporting values that are patently impossible given the implications of our intensions
    • Carcasses arising and walking
    • People recovering from a lifelong illness
    • People moving on a surface that should be impassable (e.g. a river)
Surprises & Failures

• Often complex models (including ABMs) exhibit surprising emergent properties
  – There may be things we consider very implausible that are jointly implied of various pieces of our model specification
  – There may even be things we consider “impossible” given our intended model structure that are in fact implied by it – we just didn’t realize this!
Some Model “Surprises” Reflect...

- Mistakes in our implementation (divergence of “what we told the model to do” from “what we intended to tell the model to do”)
  - Typing “a/a+b” rather than “a/(a+b)”
  - Misunderstanding of how a type of model building block (e.g. a guard in a rate transition) “works”

- Unrealistic aspects of our plan (“what we intended to tell the model to do” had hidden inconsistencies with how the world works)

- Discoveries about what could happen in the world

- We are focusing here on the first of these issues, but need to realize that it often takes time to figure out in which category a given surprise lies!
What is Debugging?

• Debugging is the process of finding and removing the defects (faults) in our program, based on observations of “failures” or “aberrant behaviour”
Best Debugging Strategy: Avoiding It!

- Defensive Programming
- Offensive Programming

We will talk about best practices for these approaches in a separate lecture
Offensive Programming: Try to Get Broken Program to Fail Early, Hard

• Asserts: Proactively scan for and flag incorrect assumptions, aborting the program as a result
• Fill memory allocated with illegal values
• Fill object w/illegal data just before deletion
• Set buffers at end of heap, so that overwrites likely trigger page fault
• Setting default values to be illegal in enums
• We will talk about Assertions & Error Handling later this week
Assertion Goal: Fail Early!

• Alert programmer to misplaced assumptions as early as possible

• Benefits
  – Documents assumptions
  – Reduces likelihood that error will slip through
    • Helps discourage “lazy” handling of only common case
    • Forces developer to deal explicitly with bug before continuing
  – Reduces debugging time
  – Helps improve thoroughness of tests
Avoid Side Effects in Assertions

- Because assertions may be completely removed from the program, it is unsafe to rely on side effects occurring in them

```java
assert ++i < max;
```

Enabling Assertions in Java

• 2 ways
  – Usual: Via java runtime command line
    -enableassertions/-ea[descriptor]
  • e.g.
    -enableassertions:com.acme.Plotter
    -enableassertions:com.acme...
    -disableassertions/-da[descriptor]
  
  – Less common: via reflection (ClassLoader)
    public void setDefaultAssertionStatus(boolean enabled)
    public void setPackageAssertionStatus(String packageName, boolean enabled)
    public void setClassAssertionStatus(String className, boolean enabled)
Enabling Assertions in AnyLogic
Assertions in Later AnyLogic Versions

• In some later AnyLogic versions, should enable assertions only in the model itself
• This is simple to do
  – Uses the package name
• More details on this are available on request
AspectJ and Eclipse

• AspectJ is a language that allows for succinctly describing “cross cutting” functionality in programs – such as tracing or logging requests

• AspectJ can automatically insert tracing instrumentation into our code
  – This gives us many of the benefits of manual tracing program execution without the need for the markup & mark-down work

• If time permits, we will present this method on Friday
A Powerful Debugging Approach

• Simplify error occurrence as much as possible
• Locate fault source
  – Gather data or context that reproduces problem
    • Rip out whole areas of model to see simplest condition that
      (sometimes just seeing what eliminates error immediately clues in to
      what it might be)
    – Record what have done
do
      • Analyze data & form hypothesis about defect
      • Determine how to prove/disprove hypothesis
      • Prove or disprove hypothesis
      • Think about defect
  Until can fix defect
  – Look for similar errors that may not yet be found
  – Figure out what about process left vulnerable to this error
Important Elements

• “Localizing” problem (Simplifying model & input until discover minimum required mechanism)
  – Save away original model (so don’t modify!)
  – Comparing good & bad versions: What is different?
  – Note down what does & does not work
  – Seeing path of execution (particularly around fault location)

• Alternate between thinking & experimenting
• Observing model state (“situation”) at points preceding error
• Compare with previous versions that were working
• Read error messages given by AnyLogic
• Confirming certain assumptions are true prior to error
• Talk with someone about issue/perform a peer review
• Specify and investigate top hypotheses
Debugging AnyLogic

- AnyLogic’s researcher version now contains a debugger
- You can attach to AnyLogic from debuggers such as Eclipse
  - The key thing is to set anylogic to use a port
Debugging Options

• Debugging is the process of locating and fixing the faults behind observed failures
• Using output for manual tracing & reporting
  – A valuable option here is to use this interactively
• Using model navigation mechanisms to inspect information about the model
• Using AspectJ for tracing/logging
• Using tools like log4j for customizable logging
• Using an external debugger (e.g. via eclipse)
• Using AnyLogic Professional/Research debugger
Using output for manual tracing & reporting

• Pros
  – Minimal learning curve
  – Flexible
  – Easily targeted

• Cons
  – Requires time-consuming manual
    • “markup”
    • de-markup
  – Can require many build/simulation iterations to localize problem
  – Limited capacity of console
Output to the Console: How To

• System.err.println(String)
  – System.err.println("Sent cure message to person [" + associatedPerson + "]");
  – This will appear in red

• traceln(String)

• System.out.println(String)
public void Cure()
{
    associatedPerson.sendMessage("Cured!", associatedPerson);
    associatedPerson.sendMessage("Cured!", associatedPerson);
    System.err.println("Sent cure message to person [" + associatedPerson + "]");
}

@Override
public String toString()
{
    return "Entity for agent " + associatedPerson;
}

/**
 * This number is here for model snapshot storing purpose<br>
 * It needs to be changed when this class gets changed<br>
 */
private static final long serialVersionUID = 1L;
Interactive reporting

• AnyLogic’s support of interactive mechanisms allows us to custom-trigger reporting through user interface actions
  – Button push
  – Mouse click
• We can also use elements like sliders to change things in a way that hints as to the nature of a problem
• This reporting may be
  – Custom-built for debugging
  – Built in, but not typically used here
Hands on Model Use Ahead

Load Provided Shared Model: ABMModelWithBirthDeath
Person-Level View

Susceptible

Infective

Death

PregnancyStatus

NonPregnant

Pregnant

getPersonName

strName

person153

appearanceTime

InitialAge

1.894

CurrentAge

FertilityRateAgeSexEthnicity

EstablishOffspringBasedOnMothersConnections

EstablishOffspringLocationBasedOnMothersLocation

RandomSex

RandomEthnicity

RandomAge

isInReproductiveYears

IsInfected
Examining Contents of Collection

Pause model execution
Click here
Logging

• *Logging* is the process of recording a record (trace) of events during program execution
  – *Recording can be made to a database, files, text console, etc.*

• Logging can be performed at a variety of levels of detail

• Log4j is one logging framework
Logging with Log4j

• Use of config files to configure
• Different levels of logger
  – TRACE, DEBUG, INFO, WARN, ERROR and FATAL
• A given logger can be associated with Multiple output streams
• Doing error uploads to a server
• Sending email (?)
public class Logger {

    // Creation & retrieval methods:
    public static Logger getRootLogger();
    public static Logger getLogger(String name);

    // printing methods:
    public void trace(Object message);
    public void debug(Object message);
    public void info(Object message);
    public void warn(Object message);
    public void error(Object message);
    public void fatal(Object message);

    // generic printing method:
    public void log(Level l, Object message);
}

Example use of Log4j

// get a logger instance named "com.foo"
Logger logger = Logger.getLogger("com.foo");

logger.warn("Low fuel level.");

logger.info("general information");

// This request is disabled, because DEBUG < INFO.
logger.debug("Starting search for nearest gas station.");
Config File

Here are example configuration files

# Set root logger level to DEBUG and its only appenders to A1.
log4j.rootLogger=DEBUG, A1

# A1 is set to be a ConsoleAppender.

# A1 uses PatternLayout.
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
Config File: Suppressing Selective Information

log4j.rootLogger=DEBUG, A1

# Print the date in ISO 8601 format
%c - %m%n

# Print only messages of level WARN or above in the package com.foo.
log4j.logger.com.foo=WARN
Multiple Outputs

- log4j.rootLogger=debug, stdout, R
  log4j.appender.stdout=org.apache.log4j.ConsoleAppender
  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

- # Pattern to output the caller's file name and line number.
  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

- log4j.appender.R=org.apache.log4j.RollingFileAppender
  log4j.appender.R.File=example.log
  log4j.appender.R.MaxFileSize=100KB
  # Keep one backup file log4j.appender.R.MaxBackupIndex=1
  log4j.appender.R.layout=org.apache.log4j.PatternLayout
  log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
Using the External Eclipse Debugger with AnyLogic
External Debugging in Eclipse

- The “Eclipse” editor is one of the most popular extant software development tools
- Eclipse offers plug-ins of many sorts
  - Debuggers
  - Profilers
  - Visualization tools
  - Version control of models
- Eclipse can be used to debug AnyLogic models at the Java source-code level
Overview: Setting up External Eclipse Debugging in AnyLogic

• In anylogic, Set the jvm options for socket based debugging (e.g. eclipse)
  – go to "Properties" on the "Simulation" to run for the anylogic model
  – Set the "Java Machine Arguments" as follows:
    -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8321

• in eclipse, create a debug configuration
  – use "Remote Java Application”
    • no project
    • for "Connection Type", select "Standard (Socket Attach)"
    • for "Connection properties”, Use
      – Host: localhost
      – Port 8321
Steps Required for Eclipse Debugging

• One time set-up for a particular model
  – Set up AnyLogic to allow debugging connections
  – Set up Eclipse to know
    • How to connect to AnyLogic
    • Where to look for source code files

• Every time want to debug
  – Go to Eclipse
  – Tell debugger to connect to AnyLogic process
  – Interrupt process
  – Set breakpoints, etc.
One-Time Setup In AnyLogic

- `-Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8321`

- These go under the "Advanced" tab of the simulation run to use
Setting up Debug Configurations
Set up: Creating a Debugging Configuration in Eclipse
Setting Up Source Code Folders

Debug Configurations
Create, manage, and run configurations
Attach to a Java virtual machine accepting debug connections

Source Lookup Path:
- src.generated - C:\Users\Nate\AnyLogicWorkspace\EclipseDebuggingExample_BUILD
- anq1v2 - C:\Users\osgood\AnyLogicWorkspace\Anq1v2_BUILD\classes
- anq1v2 - C:\Users\osgood\AnyLogicWorkspace\Anq1v2_BUILD\src.generated
- Default
The AnyLogic Workspace is Located under the Usersr Folder
Once Set up, Can...

• Set breakpoints
• See the variables, with symbolic information
• Suggestions
  – Set a breakpoint on a thrown runtime exception (regardless of whether caught)
  – Throw a caught runtime exception from model startup code
  – When catch this in Eclipse, can then use to set breakpoints (including in other files)
Start AnyLogic Model (Experiment with Extra Debugging JVM Arguments)
Leave on Opening Screen for Now
(So We can Set up Eclipse)
In Eclipse, Open “Debug” Perspective
Following Connection
Open Up Java Files from the Workspace Folder for this Project to Inspect Source & Set Breakpoints
Now Can Set Breakpoints in Main.java or Elsewhere (Here: Person.java)

Double-click in dappled/stippled area on line where want to stop execution
Return to AnyLogic & Start Simulation via Button Push
When Breakpoint is Hit, Will See Reach Point
Can Single Step, Explore & Modify Variable Contents, etc.
Warning: Breakpoints are Not Shown in Source Window – Just in “Breakpoints” area
Press “Resume” to Continue – Awaiting a Breakpoint
Example Breakpoint in Main
Example Breakpoint in Person
Once at Breakpoint, Can Look at Variables, Single Step, etc.
Variables Displayed

```
Person mother = this;
Person offspring = get_Main().add_Population((double) 0, ethnicity, RandomSex(), this.IsInfected());
traceln("A baby has been born. Baby's id is " + offspring + " while the mother is " + this);
```

```
void EstablishOffspringConnectionsBasedOnMothersConnections(Person offspring, Person mother)
```
Terminating Execution from AnyLogic Console
Remembering Breakpoints

• Note Eclipse *does* remember breakpoints from session to session
• So breakpoints that set earlier in an anylogic session will work again even after close eclipse and restart it again
• Suggestions
  – Consider creating a common breakpoints (e.g. at Main.start)
  – Disable and enable breakpoints rather than deleting them
Example of Debugging Session

![Debugging Session Screenshot]
Another Route: Catching Exceptions at Defined Places of Interest
Example Setup: Set up Function to Trigger the Debugger

```java
try {
    throw new RuntimeException("arbitrary");
}
catch (RuntimeException e) {
    traceln("Throw & caught exception");
}
```
In Startup Code for Model, Call Function

```java
environment.deliverToRandom("Infect!");
TriggerDebugger();
```

Destroy Code:
Request Creation of Exception Breakpoint
Request as Breakpoint Regardless of Handling
Should Now be in List of Enabled Breakpoints
Back in Eclipse, the Debugger Should have been Triggered & at Exception Handler (If not, close “Main.java” and double-click on topmost “stack frame” (Where Exception is triggered))
Using the
AnyLogic Built-in Debugger
Running the Debugger
Running the Models

SIR Agent Based Model of Disease Diffusion

Model parameters

- Total population: 200
- Fraction initially infected: 0.05
- Contact rate: 5.0 contacts per day
- Infectivity: 0.05
- Average illness duration: 15.0 days
- Layout type: Ring
- Network type: Random
- Links per agent: 10
- Maximum link distance: 50.0
- Percent of long distance links: 0.05

Description

We distinguish between three different states of a person: Susceptible, Infectious, and Recovered. To reflect the fact that some people are already infected at the beginning of the simulation, the agent states at entry point have a fraction initially infected. The choice of the initial state is probabilistic and is based on a global model parameter Percent Initially Infected.

The transition to the state infectious models the event of disease being passed to the agent from a sick person. In the model, the trigger of that transition is a message "Infected". Once in the infectious state, the agent is able to pass the disease to others, therefore we are interested in his contacts. We assume that ContactRate is constant while in the infectious state, and we can use external transition Contact that will be repeatedly taken until the agent recovers. In the action of that transition, the agent chooses another agent from the people that he knows (this is defined by the social network), and sends him the message "Infected". However, not every contact results in infection being passed. Message is sent with the probability of infection Probability, which is yet another global parameter. If the message reaches an agent who is already infected or recovered, it is ignored. Finally, the transition Recovery is a timeout that should model the illness duration.

This model allows you to explore the dynamics of disease diffusion in these types of networks:
- Based on distance – people are linked if the geographical distance between them is at most more than a given value.
- Random – a person is linked to a random subset of the population.
- Small World (Watts 1999) – most links are neighbors, but there is a certain percent of long-range links. To establish neighborhood people can be placed, e.g., on a virtual ring.
- Scale Free (Barabasi, Albert 1999) – some people are "hubs" with lots of connections and some are "hubs".

The experiment design follows the paper:
Setting a Breakpoint

SIR Agent Based Networks

networkTypeToString - Function

```java
switch( type ) {
    case Environment.NETWORK_USER_DEFINED:
        return "Custom";
    case Environment.NETWORK_RANDOM:
        return "Random";
    case Environment.NETWORK_ALL_IN_RANGE:
        return "Based on distance";
    case Environment.NETWORK_RING_LATTICE:
        return "Ring lattice";
    case Environment.NETWORK_SMALL_WORLD:
        return "Small world";
    case Environment.NETWORK_SCALE_FREE:
        return "Scale free";
    default: return "Unknown";
}
```
When we Hit the Breakpoint...
Components to Direct Execution
Visible ("In-Scope") Variables
Exploring Composite Variable Values in the Debugger
Inspecting Composite Variables
Changing Variable Values During Debugging

Expression

type=3

Enable

OK Cancel
Stepping into Auto-Generated Code
Seeing Result of Expression Evaluation

Note that this doesn’t update immediate – may need to switch stack frames in the “Debug” method.
To see the update.