Reproducible simulation experiments with ML-Rules and SESSL

Supplementary Material

T. Warnke, T. Helms, and A.M. Uhrmacher

September 26, 2017

This is the appendix of the application note on the ML-Rules binding for SESSL. The binding brings together the modeling language ML-Rules and the experiment specification language SESSL. In this document, we describe the application example from the application note in more detail. In particular, we present the complete model of the Dictyostelium discoideum amoeba aggregation process in space and show some examples for additional experiments that can be conducted with the model.

1 The model in ML-Rules

In (Gilbert et al., 2015, pp. 195ff), an ML-Rules model of Dictyostelium discoideum amoeba aggregation was implemented and presented, see the complete implementation in Figure 1. It comprises intracellular non-spatial dynamics of the amoebas as well as their movement in space. Both phenomena are interrelated: The movement in space is directed towards high concentrations of intercellular cAMP, which is in turn secreted by the cell. As a result, the amoeba cells aggregate in space over time, which is a crucial step in the life cycle of Dictyostelium discoideum towards the formation of a multicellular slug.

2 Experimentation

The model initially contains a population of Dictyostelium discoideum cells that are uniformly distributed in a grid. For each cell, complex intracellular signalling processes as well as the spatial dynamics of the cell (i.e., moving from one grid cell to a neighboring one) are modeled. Thus, single simulation runs are computationally expensive, especially for larger grid size and more amoeba cells. Additionally, both intracellular and intercellular processes are subject to stochasticity. A typical simulation setup must therefore execute replications of each model configuration.
// PARAMETERS
xmax:2; // change the grid size here
ymax:2; // change the grid size here
d.dicy:0.002; kd.camp:2.4; nA:6.023e23; v:3.6720e-14;
k1:2.0; k2:0.9/nA/v/1e-6; k3:2.5; k4:1.5; k5:0.6; k6:0.8/nA/v/1e-6;
k7:1.0/nA/v/1e-6; k8:1.3/nA/v/1e-6; k9:0.3; k10:0.8/nA/v/1e-6;
k11:0.7; k12:4.9; k13:23.0; k14:4.5;
init.cAMP:1100; init.cAMPi:4100; init.ACA:7300; init.PKA:7100;
init.ERK2:2500; init.RegA:3000; init.CAR1:6000;

// FUNCTIONS
initialize :: num -> num -> sol;
initialize 0 y = [];
initialize x 0 = initialize(x - 1, ymax);
initialize x y = (init.cAMP) CAMPe(x,y)
    + 1 CELL(x,y)[
        (init.cAMP) CAMPi + (init.ACA) ACA + (init.PKA) PKA
    + (init.ERK2) ERK2 + (init.RegA) RegA + (init.CAR1) CAR1
    ] + initialize(x, y - 1);

// SPECIES DEFINITIONS
System();
CELL(num,num); CAMPe(num,num); CAMPi(); ACA(); PKA(); ERK2(); RegA(); CAR1();

// INITIAL SOLUTION
>>INIT[System[initialize(xmax, ymax)]];

// REACTION RULES
// intra-cellular dynamics
CAR1:c -> ACA + CAR1 @ k1*#c;
ACA:a + PKA:p -> PKA @ k2*#a*#p;
CAMPe(x,y); -> PKA + CAMPe(x,y); + initialize(x, y - 1);
PKA:p -> @ k4*#p;
CAR1:c -> ERK2 + CAR1 @ k5*#c;
PKA:p + ERK2:e -> PKA @ k6*#p*#e;
CELL(x,y)[s?]c -> CELL(x,y)[RegA + s?] @ k7*#c;
ERK2:e + RegA:r -> ERK2 @ k8*#e*#r;
ACA:a -> CAMPe + ACA @ k9*#a;
RegA:r + CAMPe(x,y) -> RegA @ k10*#r*#a;
CELL(x,y)[ACA:a + s?] -> CAMPe(x,y) + CELL(x,y)[ACA + s?] @ k11*#a;
CAMPe(x,y); -> @ k12*#a;
// countTwoAtts(...) is a predefined function, see documentation
System[CAMPe(x,y); + CELL(x,y)[c?] + r?] -> System[CELL(x,y)[CAR1 + c?] + CAMPe(x,y) + r?] @
k13*#a/1 + countTwoAtts(r?, 'Cell', x, y);
CAR1:c -> @ k14*#c;

// movement of cell to adjacent position depending on external cAMP amount
CELL(x,y)[s?] + CAMPe(x,y);:a1 + CAMPe(x,y);:a2 -> CELL(x,y)[s?] + CAMPe(x,y); + CAMPe(x,y);:
 @ if ((#a2>(#a1) && (#a1 > 0) && ((x1=x2) && (y1=y2)) && ((x1-x2==1)&&(x1-x2==-1)) &&
      ((y1-y2==1)&&(y1-y2==-1))) then k.dicy*(#a2/1/#a1) else 0;

// cAMP diffusion
CAMPe(x,y); -> CAMPe(x+1,y); if (y>ymax) then kd.camp*#a else 0;
CAMPe(x,y); -> CAMPe(x,y+1); if (x>xmax) && (y>ymax) then kd.camp*#a else 0;
CAMPe(x,y); -> CAMPe(x,y); if (x=xmax) then kd.camp*#a else 0;
CAMPe(x,y); -> CAMPe(x,y); if (y>ymax) then kd.camp*#a else 0;
CAMPe(x,y); -> CAMPe(x,y); if (x=xmax) && (y>y1) then kd.camp*#a else 0;
CAMPe(x,y); -> CAMPe(x,y); if (y=y1) then kd.camp*#a else 0;
CAMPe(x,y); -> CAMPe(x,y); if (x=xmax) && (y>y1) then kd.camp*#a else 0;
CAMPe(x,y); -> CAMPe(x,y); if (x=xmax) && (y=y1) then kd.camp*#a else 0;
CAMPe(x,y); -> CAMPe(x,y); if (x=xmax) && (y>ymax) then kd.camp*#a else 0;

Figure 1: Dictyostelium aggregation model in discrete space written in ML-Rules (Gilbert et al., 2015, pp. 14481ff).
import sessl._
import sessl.mlrules._
import sessl.ssj.LHCSampling

ox
execute {
  new Experiment with ParallelExecution with Observation with LHCSampling with
    CSVOutput {
  val grid = 1 to 10
  model = "Dicty.mlrr"
  simulator = HybridSimulator()
  replications = 9
  stopTime = 5000
  set("xmax" <~ grid.size, "ymax" <~ grid.size)
  parallelThreads = -1
  lhc(100, "kd_camp" <~ interval(1, 4), "kd_dicty" <~ interval(0, 0.02))
  observeAt(4000)
  for (x <- grid; y <- grid)
    observe(s"$x-$y" ~ s"System/CELL($x.0, $y.0)"
  }
  withExperimentResult(writeCSV)
}
}

Figure 2: SESSL specification of an experiment using LCH sampling and the
ML-Rules model of *Dictyostelium discoideum* to explore the parameters
kd_camp and kd_dicty.

Using the ML-Rules binding and SESSL, complex experiments can be de-
scribed to explore the parameter space of the model. For example, Figure 2
shows a complete SESSL experiment applying Latin Hypercube (LHC) Sampling
to explore the parameter space. LHC Sampling is especially relevant in the
early phase of model development, when modelers try to get an intuition for
the sensitivity of the model towards numerous parameters. In SESSL, a specific
number of points in the parameter space can be obtained via LHC sampling for
an arbitrary number of parameters in arbitrary intervals.

In the following, we describe the experiment specification in Figure 2 in
detail.

The *import* statements in the first three lines provide the experiment specification
with features from specific packages and bindings. Here, the SESSL core, the
complete ML-Rules binding, and the Latin Hypercube Sampling from the SSJ
binding (*L’Ecauyet et al.*, 2002) are imported.
new Experiment with ParallelExecution with Observation with LHCSampling with CSVOutput {

Line 6 defines the features of the experiment: The ML-Rules binding provides the Experiment itself as well as the Observation and ParallelExecution traits, whereas the LHCSampling lives in the SSJ binding. The trait CSVOutput is located in the SESSL core. This defines what configuration options are offered by the experiment.

model = "Dicty.mlrj"
simulator = HybridSimulator()
replications = 9
stopTime = 5000
set("xmax" <~ grid.size, "ymax" <~ grid.size)

For example, setting the model (l. 8), the simulation algorithm (l. 9), the number of replications (l. 10), and the simulation end time (l. 11) is supported by the ML-Rules binding. In line 12, parameter values of the model are set, which is also provided by the ML-Rules experiment. When starting a simulation, the values for xmax and ymax set here will be used to override the default values in the ML-Rules model file (see the model, ll. 2–3). The same value is used for both parameters, which leads to a quadratic grid.

val grid = 1 to 10

The actual grid size is defined in line 7 as a Scala range, which can be thought of as an interval. For example, in the above listing an interval grid ranging from 1 to 10 is defined, representing the coordinates along one dimension of the grid in the model. By changing the upper border of grid, the grid size to simulate can be changed in the SESSL specification.

parallelThreads = -1

The experiment employs three additional traits. The trait ParallelExecution provides the parallelThreads directive (l. 13). By configuring this to the value -1, the simulation will use all cores but one on the computer the experiment is executed on.

observeAt(5000)
for (x <- grid; y <- grid)
observe(s"x-y" ~ s"System/CELL($x.0, $y.0)")

observeAt(5000) for (x <- grid; y <- grid) observe(s"x-y" ~ s"System/CELL($x.0, $y.0)")

The trait Observation provides the configuration options observeAt (l. 15) to specify when to observe and observe (l. 17) to specific what to observe. Here, the time of observation is the simulation end time 5000. The observed variables are the counts of CELLS (i.e., the number of Dictyostelium discoideum cells). By enclosing the observe statement in a loop over both grid dimensions, again using the grid variable, all possible attribute configurations of the CELL species are counted individually. Syntactically, line 17 makes use of the Scala string
interpolator \texttt{s} that injects variable values into a string. For example, if \texttt{x} and \texttt{y} are both 1 in the first loop iteration, \texttt{s"$x-\$y"} will evaluate to “1-1”. This exemplifies how domain-specific statements provided by SESSL can be combined with programming techniques provided by Scala.

\begin{verbatim}
14 lhc(100, "kd_camp" <- interval(1, 4), "kd_dicty" <- interval(0, 0.02))
\end{verbatim}

In line 14, the \texttt{lhc} directive provided by the \texttt{LHCSampling} trait specifies the configuration for Latin Hypercube Sampling: 100 points in the parameter space are sampled, and the parameter space comprises the variable \texttt{kd_camp} and \texttt{kd_dicty} with the given interval borders. The parameter values that are sampled will override the ones set in the model in lines 7–8 and determine the movement of cells and cAMP in the model. Alternatively, a full factorial exploration can be applied easily by using line 15 instead of line 14.

\begin{verbatim}
20 withExperimentResult(writeCSV)
\end{verbatim}

Finally, line 20 specifies the output of simulation results. In particular, a predefined CSV file writer from the SESSL core is used to produce one file for each replication containing the numbers of amoeba cells in each grid cell. The files are placed in numbered folders that also contain a CSV file with the parameter values of the used configuration. By writing the results into a CSV file and being able to manipulate the data directly in SESSL with programming techniques provided by Scala, arbitrary output formats can be implemented. Thus, simulation results can be analyzed with various tools at the user’s discretion.

For example, the produced files can be read in and visualized using the R package \texttt{plot3D} with an R script, see Figure 3. Exemplary distributions of cells on the grid after 4000 simulation time units, produced by three configurations, are shown in Figure 4. Based on the SESSL description in Figure 2, assuming the experiments results to be in \texttt{/experiments/sessl/results}, the following command generates a PDF containing the plots from Figure 4:

\begin{verbatim}
createDistPDF("/experiments/sessl/results/", 0:99, 10, 2, 1)
\end{verbatim}

Based on the results, one might suggest that the accumulation depends on the configuration. To measure the accumulation, the standard deviation of the number of amoeba cells per grid cell could be applied, leading to the plot shown in Figure 5 for all configurations selected by Latin Hypercube Sampling. Small values for \texttt{kd_dicty} apparently result in less accumulation. The attached R script also contains the code to generate this plot.

Alternatively to a parameter space exploration, an optimization experiment could be applied with the ML-Rules binding for SESSL to find a configuration with a high accumulation of amoeba cells, see Figure 6. The Opt4J binding is used to apply a particle swarm optimization. Further, the standard deviation of the amoeba cell distribution can be calculated directly in the experiment description by writing the Scala code (ll. 36–39).
# get the column name "x-y" and return a vector (x,y)
cords <- function(name) {
  as.numeric(unlist(strsplit(substring(name,2),"[.]")))
}

# get the config from the config.csv for the header of the plot
header <- function(dir,config) {
  x <- as.matrix(read.csv(paste("./config-",config,"/config.csv",sep="")))
paste(x[1,1],"="x[1,2],",",x[2,1],"="x[2,2],sep=" ")
}

# draw nine plots (for each replication one)
plotReplications <- function(dir, config, gridSize, columns, rows) {
  par(mfrow=c(1,2),mar=c(5.1,4.1,4.1,4.1),oma = c(0, 0, 2, 0))
directory <- paste(dir, "config-",config,sep="")
files <- dir(directory, pattern="run-")
total <- gridSize*gridSize
for (i in 1:length(files)) {
  file <- paste(directory,"/",files[i],sep="")
  values <- as.matrix(read.csv(file,header=TRUE))
  x <- y <- 1:gridSize
  z <- matrix(ncol=gridSize,nrow=gridSize)
  for (j in 2:ncol(values)) {
    c <- coords(colnames(values)[j])
    z[c[1],c[2]] <- values[1,j]
  }
  colors <- c("FFFFFF",gray.colors(total-1,start=0.95,end=0))
  image2D(z,x,y,nacol='white',breaks=seq(-0.5,total-0.5,by=1),
  col=colors,colkey=list(plot = FALSE))
  colkey(add=TRUE,col=colors,at=seq(0,total,(total)/10),clim=c(0,100))
  mtext(header(dir,config), outer = TRUE, cex = 1.5)
}

# create a PDF including all plots for all configurations (one page per configuration)
createPDF <- function(dir, configs, gridSize, columns, rows) {
  pdf("plots.pdf",width=12,height=6)
lapply(configs,function(x) plotReplications(dir, x, gridSize, columns, rows))
dev.off()
}

Figure 3: R script creating plots to visualize the distribution of the amoeba cells.
3 Dependency Resolution

SESSL relies on Maven (https://maven.apache.org) to resolve dependencies between packages. Therefore, the information about the dependencies of a simulation experiment is contained in a `pom.xml` file that is usually placed in the same directory as the experiment specification. Based on the `pom.xml`, Maven locates and downloads all required binary artifacts. SESSL and ML-Rules artifacts are available from Maven Central (https://search.maven.org), which is Maven’s default artifact repository. As the artifacts in Maven repositories are persistent and platform-independent, Maven ensures reproducibility of simulation experiments across different computers, over time, and for diverse operating systems. A minimal `pom.xml` for the simulation experiments in this document is shown in Figure 7. A complete example is available at https://git.informatik.uni-rostock.de/mosi/sessl-quickstart.

References


Figure 4: Illustration of amoebas accumulation for three different configurations of the model and two replications for each configuration.
Figure 5: Standard deviations of amoeba cells per grid for 100 configurations selected by Latin Hypercube Sampling.
import sessl._
import sessl.optimization._
import sessl.opt4j._
import sessl.mlrules._

maximize { (params, objective) =>
  execute(new Experiment with ParallelExecution with Observation {
    val grid = 1 to 10
    model = "Dicty.mlrj"
    simulator = HybridSimulator()
    parallelThreads = -1
    replications = 9
    stopTime = 5000
    set("xmax" <~ grid.size, "ymax" <~ grid.size)
    set("kd_camp" <~ params("kd_camp"))
    set("kd_dicty" <~ params("kd_dicty"))
    observeAt(5000)
    for (x <- grid; y <- grid)
      observe(s"x-$y" ~ s"System/CELL($x.0, $y.0)"
    var stdDevs = List.empty[Double]
    withRunResult(result => {
      val counts = for (x <- grid; y <- grid) yield
        result.apply(s"x-$y").asInstanceOf[Double]
      stdDevs = stdDev(counts) :: stdDevs
    })
    withExperimentResult(_ => {
      objective <~ stdDevs.sum / stdDevs.size
    })
  }) using new Opt4JSetup {
    param("kd_camp", 1.0, 0.1, 4.0)
    param("kd_dicty", 0.0, 0.001, 0.02)
    optimizer = ParticleSwarmOptimization(iterations = 5, particles = 5)
    withOptimizationResults { results =>
      println("Overall results: " + results.head) //print results
    }
  })
}

def stdDev(counts: Seq[Double]): Double = {
  val mean = counts.sum / counts.size
  val variance = (for (c <- counts) yield (mean - c) * (mean - c)).sum /
    (counts.size - 1)
  math.sqrt(variance)
}

Figure 6: A SESSL experiment description for an optimization experiment.
Figure 7: A pom.xml for a simulation experiment with SESSL and its bindings for ML-Rules, SSJ, and Opt4J. The same version of all bindings must be used to avoid dependency conflicts (ll. 13, 18, 23). Note that the SESSL core and the ML-Rules package do not need to be declared as they are resolved as transitive dependencies of the declared artifacts.