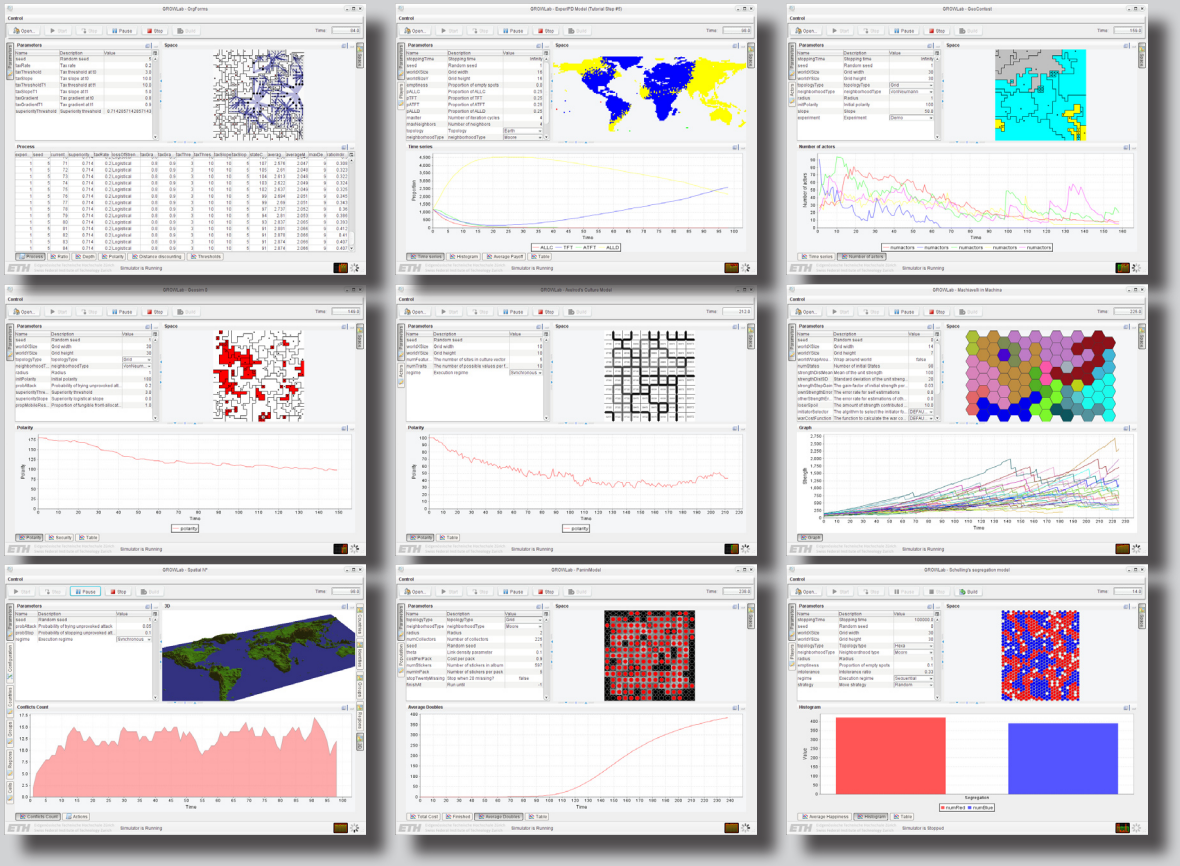
GROWLab

Modeler's guide



Center for Comparative and International Studies (CIS)
International Conflict Research (ICR)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction

Computer simulation has proved to be a vital adjunct to traditional approaches of understanding social phenomena in fully understanding the dynamics of individual and collective actions. These simulation tools are critical in the development and evaluation of policies that affect the way we want to shape our society. The GROWLab software toolbox proposes novel ideas to make agent-based simulations more effective and hopes to illustrate the importance and positive impacts of them in the understanding of conflict patterns.

Goal

GROWLab has been designed to facilitate the modeling, simulation, analysis, and validation of complex social processes, with a special focus on geographic conflict research. Its aim is to bring the development of agent-based simulations to the next level of complexity and realism.

The specific aim of this work is to develop a Java class library that follows the tradition of toolboxes for agent-based modelling, which has turned out to be a successful level of abstraction because of its inherent flexibility.

Features

- The seeding of the model with empirical facts (including geo-referenced data) to calibrate the environments and mechanism to the appropriate level of realism;
- The effective modelling of complex network and hierarchical relationships between model actors and the efficient scheduling of their interactions;
- The execution of large number of simulation runs on a grid made of many independent computers to test the sensitivity of the models;
- The statistical and visual analysis of the state of the system, as well as the unfolding of the processes over time.

Installation

GROWLab is a pure Java framework and should therefore run on any operating system (typically Windows, Mac OS X, and Linux), as long as a Java Virtual Machine (VM) exists for it. To compile your own model, the Java Development Kit (JDK) will be necessary. Two ways of installing GROWLab are provided, with the following advantages:

Using the installer

- Best for modeling that require a stable version of GROWLab.
- Allow to quickly deploy GROWLab on target machine. This is especially useful for making demonstrations as well as running batch runs.

From source

- Best for modelers who want to keep up with the latest improvements of GROWLab.
- Gives the possibility to modify the GROWLab source code.
- Allows to include your model into the main GROWLab code repository. This give a possibility to GROWLab developers to make sure the latest changes are compatible with your mode.

Installation using the installer

The following procedure describes how to install the GROWLab runtime environment by using the available installers:

Download

Download the appropriate installer for your operating system from:

http://www.icr.ethz.ch/research/growlab/download

and place the file to a temporary directory. You should now have one of the following installers on your machine:

growlab_windows_0_x_x.exe
growlab_macos_0_x_x.img
growlab_unix_0_x_x.sh

Install

1. Run the installer, typically by double-clicking on it
2. Follow the instructions: the installer will lead you through the installation process and install all the necessary files

The installer also provides the necessary mechanisms to easily remove GROWLab again from your system. You may install a new version of GROWLab on top of an older version. Your configuration will not be lost.

Installation from source

The following procedure describes how to install the GROWLab environment from our source code repository:

Prerequisites

1. An account for our Subversion repository is necessary at this point. Please contact us if you don't already have one.
2. A recent version of the Java SE Development Kit (JDK) installed on your system.
3. The Java 3D extension to the Java Virtual Machine.
4. IntelliJ IDEA Java Integrated Development Environment installed and the above Java installation configured (in *File→Settings→Project Settings→Project JDK*). Please contact us if you don't have a license.

Source code

You can retrieve the GROWLab source code using Subversion by invoking the following command:

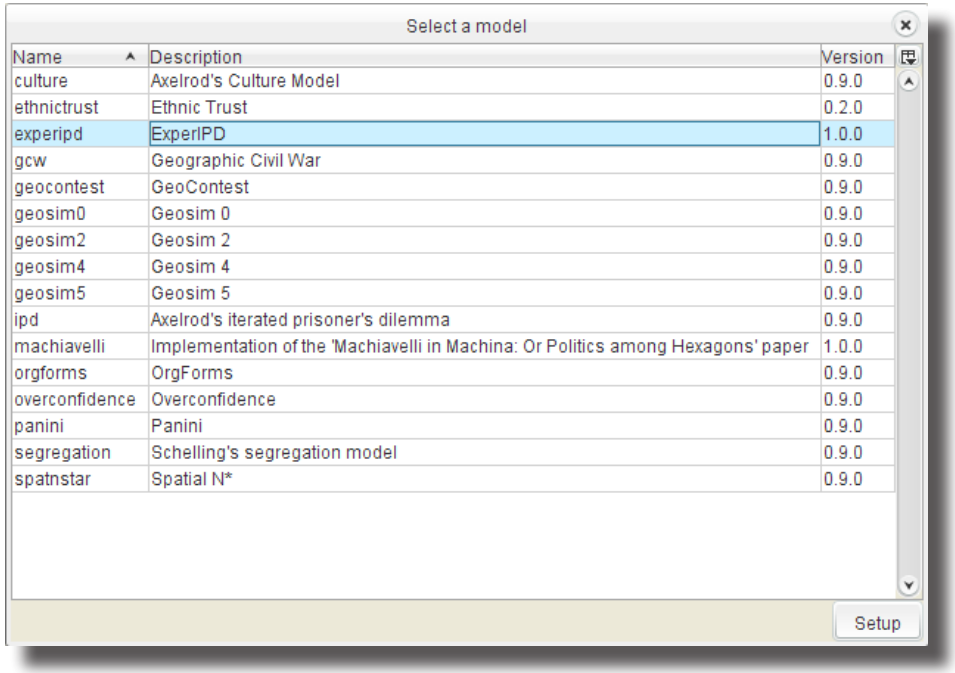
```
svn co http://cederman.ethz.ch/
    svn/java/ethz/trunk/growlab/
    growlab
```

An easier alternative is to check out the source code directly from IntelliJ IDEA:

1. Go to *Version Control→Checkout from Version Control→Subversion*
2. Add Repository Location: http://cederman.ethz.ch/svn/java/ethz/trunk/growlab/
3. Checkout http://cederman.ethz.ch/svn/java/ethz/trunk/growlab/
4. Select a destination directory where the local copy should be stored, e.g. C:\java\ethz\growlab

Running a simulation (1/4)

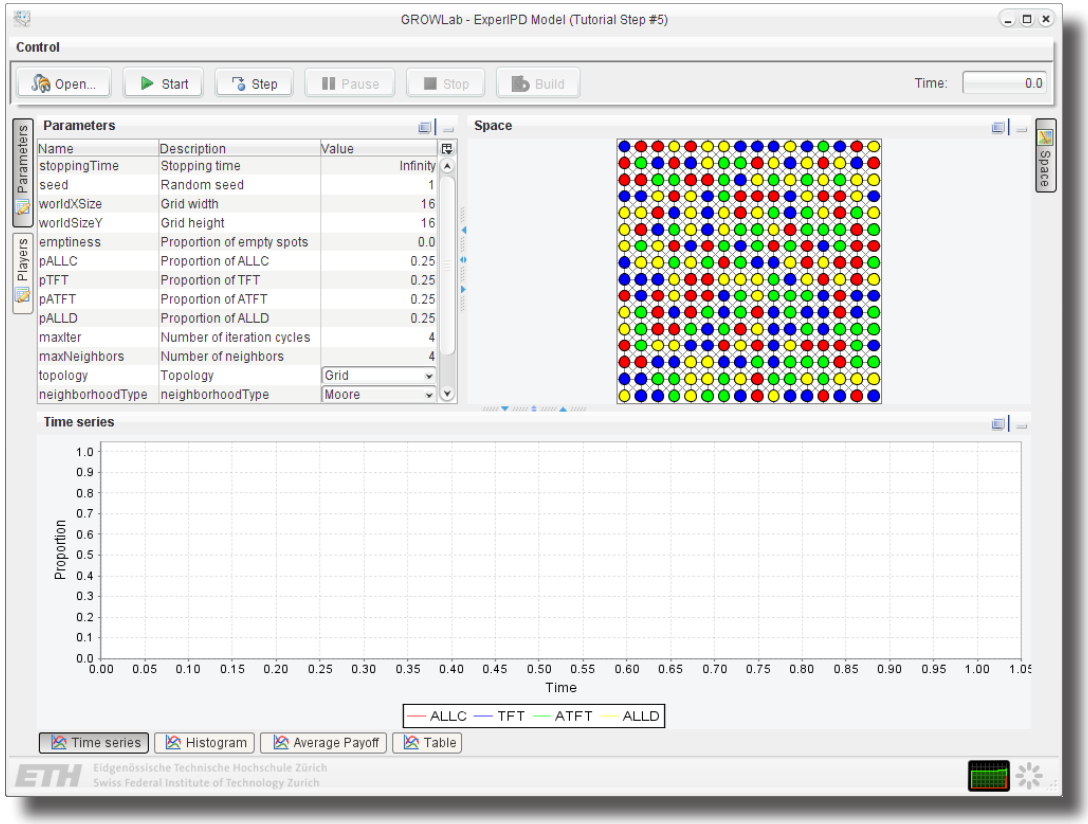
Running a simulation interactively can easily be achieved. If GROWLab was installed using the installer, run the GROWLab executable and you will be provided with a list of available models:



The list provides information about the model name, its description and its version number. Select the model you want to run and hit the Setup button.

Running a simulation (2/4)

Once the model has been set up, you will be presented with GROWLab main user interface:



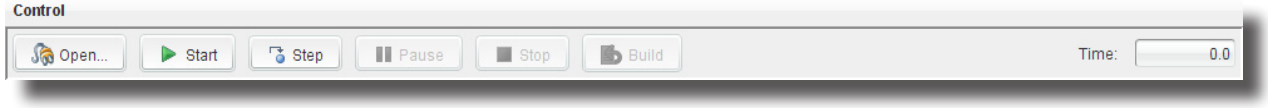
The user interface is organized in five areas used to structure the views and controls into a coherent layout:

1. Controls for the simulation (upper bar)
2. The input parameters and output variables (upper left dock)
3. Spatial views (upper right dock)
4. The dynamic views (lower dock)
5. Various status information about the current state (lower bar)

Each view is docked to a border and can be minimized and detached when desired.

Running a simulation (3/4)

The controls in the upper part of the user interface let you control the execution of the simulation:



Available controls

- *Open...*: will open the dialog with the list of available models, giving the possibility to start another simulation model.
- *Start*: execute the simulation model until it has reached the specified stoppingTime.
- *Step*: run the simulation for a single time unit, then pause the simulation.
- *Pause*: pause the simulation. The execution can be resume by hitting *Run* or *Step*.
- *Stop*: stop the simulation
- *Build*: rebuild a simulation using the currently specified parameters.

Possible states

The status bar at the bottom indicates the state the simulator is currently in. Possible values are:

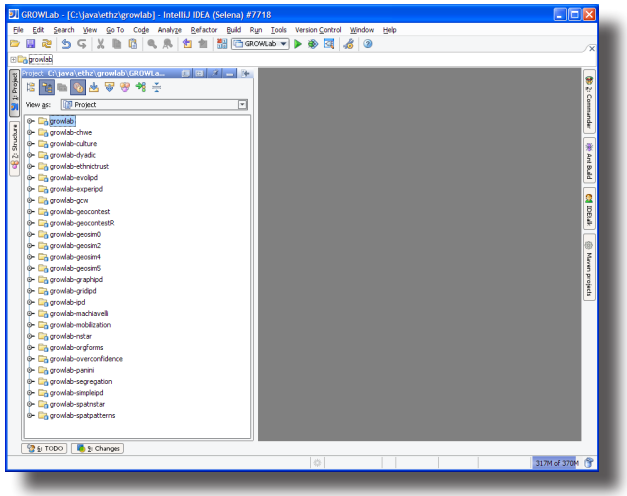
- Invalid: some parameters have changed and the simulation has to be rebuilt before it can be executed.
- Initialized: the simulation is ready to be started.
- Running: in this state, the simulator can only be stopped or paused.
- Paused: the execution is paused. You can use Start or Step to resume its execution.
- Stopped: the simulation has reached the end and has to be rebuilt to be run again.

Running a simulation (4/4)

GROWLab can also be run from IntelliJ IDEA (useful during the development of a model), as well as from the command line (typically to execute batch runs).

IntelliJ IDEA

1. Locate the predefined project file (GROWLab.ipr), which is provided at the root of the installation.
2. Open it with IntelliJ IDEA: *File→Open Project...*
3. You should now be ready for development:



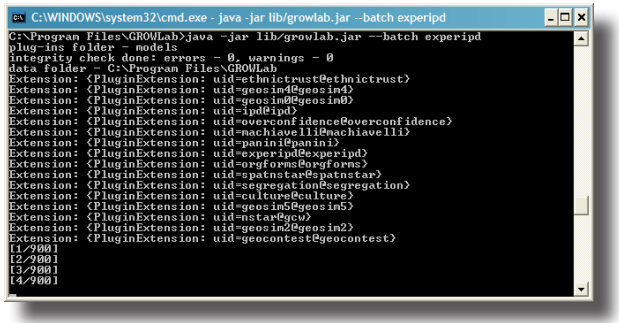
Command line

1. Go to the root of your GROWLab installation
2. Type the following command to start a simulation in interactive (graphical) mode:

   ```
   java -jar lib/growlab.jar experipd
   ```

   The last parameter is the name of the model. If ommited, the list of available models will be presented
3. To start a batch run, add the --batch parameter:

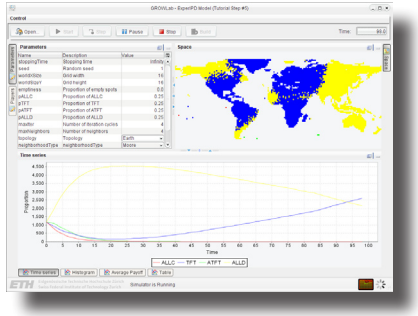
   ```
   java -jar lib/growlab.jar --batch
       experipd
   ```



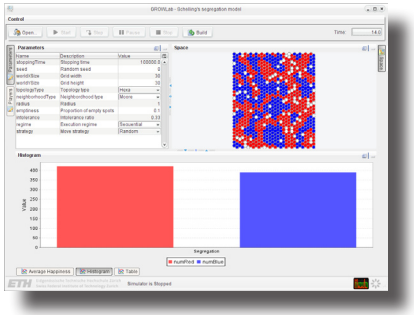
Example models

GROWLab comes with a fair number of example models. Do not hesitate to try them out and to study their source code. The models that are especially good at showing the best practices with GROWLab are:

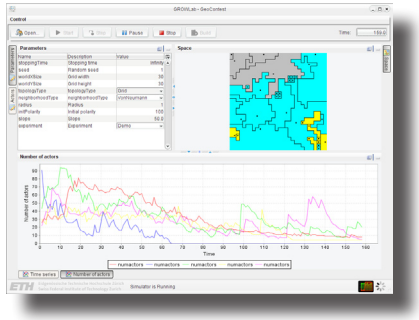
experipd: Our adaptation of the iterated prisoner's dilemma (popularized by Robert Axelrod). It is a repeated game in which two players may each "cooperate" with or "defect" (i.e., betray) the other player. In this game, the only concern of each individual player ("prisoner") is maximizing his/her own payoff, without any concern for the other player's payoff.



segregation: Famous model by Thomas Schelling showing that small preference for one's neighbors to be of the same color could lead to total segregation.



geocontest: GeoContest is a simplified version of the Geosim that allows for a variety of conquest strategies to be plugged into the model in order to examine their impact in a multipolar setting.

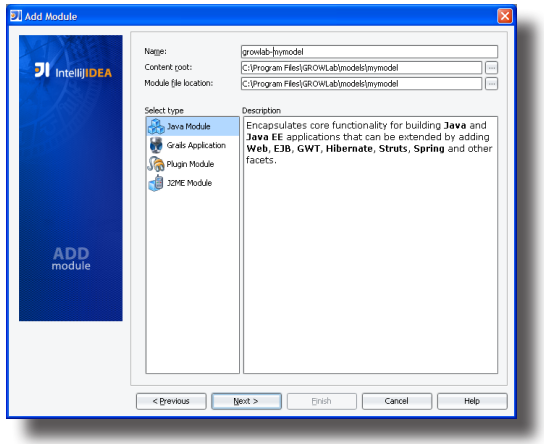


Building a model

On this page, we explain on to setup a new module within IntelliJ IDEA. However, the same configuration can easily be made using another IDE such as Eclipse or NetBeans.

Create a new module

1. Within IntelliJ IDEA, go to *File→New Module...*
2. Select *Create a module from scratch*.
3. Name it growlab-mymodel (where mymodel is the name of your model) and set its content root to the models/mymodel (relative to your installation root) directory.



4. Accept the default source directory (i.e. src).

Setting dependency

A dependency needs to be made between your model and the GROWLab module. To create this binding, do the following within IntelliJ IDEA:

1. Select your the module of your model (e.g. growlab-mymodel)
2. Right-click on it and choose *Module Settings* from the context menu.
3. Go to the *Dependencies* tab.
4. Select *Add...→Module dependency...*
5. Choose the growlab module and click Ok.
6. While you are at it, also change in the Paths tab so that the Compiler Output results in a directory named classes under your model root directory.

Laboratory

The basic building blocks to create a model is an implementation of the Lab and of the Model interfaces. The role of the Lab class is to setup the infrastructure for the proper execution of a Model object, which contains all the behavioral code.

This is easily achieved by extending the AbstractLab class, which requires only two methods to be implemented:

```
public class MyLab extends AbstractLab<MyModel> {
   public String getName() {
        return "MyModel";
    }

    public MyModel createModel() {
        return new MyModel(this);
    }
}
```

Note that we don't include the import statements required by Java, as they are automatically suggested by IntelliJ IDEA.

Method descriptions

getName(): Returns the model name. This is only used for presentation purpose.
createModel(): Returns an instance of the model which can be used to run a simulation.

Generics

GROWLab relies heavily on generics, an extension to the Java programming language that was introduced with Java 5. If you need to familiarize yourself with, read Generics in the Java Programming Language.

Input parameters

The Lab class is also meant to manage the input parameters to the models. Parameters can hold values for any Java type, including enum. Here is how three parameters to the Lab class (note that stoppingTime is provided by the AbstractLab class):

```
public class MyLab extends AbstractLab<MyModel> {
    IntegerParameter seed = new IntegerParameter(
        "seed", "Random seed", 0);
    DoubleParameter intol = new DoubleParameter(
        "intol", "Intolerance ratio", 0.33);

    public MyLab() {
        addParameters(stoppingTime, seed, intol);
    }
    ...
}
```

Common parameters

Typical parameter classes provided by GROWLab include:

- BooleanParameter
- IntegerParameter
- DoubleParameter
- StringParameter
- EnumParameter

Arguments

The arguments to the various parameter classes are usually:

1. The name of the parameter, by convention the same name as the Java variable.
2. A description of its meaning
3. A default value

Model

We can finally write the Model class by extending the AbstractModel class, which requires only two methods (and the constructor) to be implemented:

```
public class MyModel extends AbstractModel {
    MyLab lab;

    public MyModel(MyLab lab) {
        this.lab = lab;
    }

    public void buildModel(Simulator simulator) {
        System.out.println("Building using " + simulator);
    }

    public void step(Simulator simulator) {
        System.out.println("Time is: " + lab.currentTime.getValue(this));
    }
}
```

Method descriptions

buildModel(Simulator): Executed when a model run is initialized. Its purpose is typically to create the data structures and the agents.

step(Simulator): Called at every time step and contains the simulation behavior to be run repeatedly.

Life of a model

1. buildModel();
2. while(!stopped) {
   step()
   }
3. finish()

Bootstrapping

To be executed, either a XML descriptor file needs to be added to the project, or an entry point to the program needs to be specified. The creation of the XML descriptor is presented under the Packaging section and we will cover on this page only the latter. Like in any Java program, an entry point can be specified using the main(String[]) method:

```
public class MyLab extends AbstractLab<MyModel> {
    ...

    public static void main(String[] args) {
        SimulatorFactory.createSimulator(
            new MyLab(), args).setup();
    }
}
```

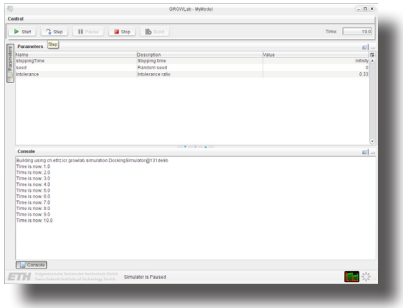
Execution

In IntelliJ IDEA, the simulation can easily be executed by right-clicking on the the main(String[]) method and by selecting *Run*.

With the Program parameters field left empty, the simulator will start in graphical mode. If “batch” is passed as parameter, then the model will be executed within the console.

Congratulations!

Your first model is now functional, and although it is not doing much it should nevertheless make clear the anatomy of a simulation model. Look at the output to make sure the system produces the expected behavior.



Random numbers

Random encapsulates the Colt library's random number generation. Initializing the random number generator and instantiating a new distribution works as follow:

```
public class MyModel extends AbstractModel {
    MyLab lab;
    UniformDistribution uniform;

    public MyModel(MyLab lab) {
        this.lab = lab;
    }

    public void buildModel(Simulator simulator) {
        uniform = new UniformDistribution(
            new MersenneTwister(lab.seed.getValue(this)));
    }

    ...
}
```

Common distributions

- UniformDistribution
- NormalDistribution
- LognormalDistribution
- BinomialDistribution
- PoissonDistribution

Data structures (1/2)

Layers, topologies, and configurations

In many agent-based models agents live in a two-dimensional grid world. These Object2DGrids (in RePast) have two major functions: They store the agents themselves, and they define relations (such as neighborhood) between agents.

Correspondingly, in GROWLab we introduce two interfaces capturing the two tasks: A layer is any collection of alike agents, and a topology is a set of relationships between them. In addition, in order to represent hierarchies of agents, we introduce the configuration interface. The following pages explain layers, topologies and configurations in detail.

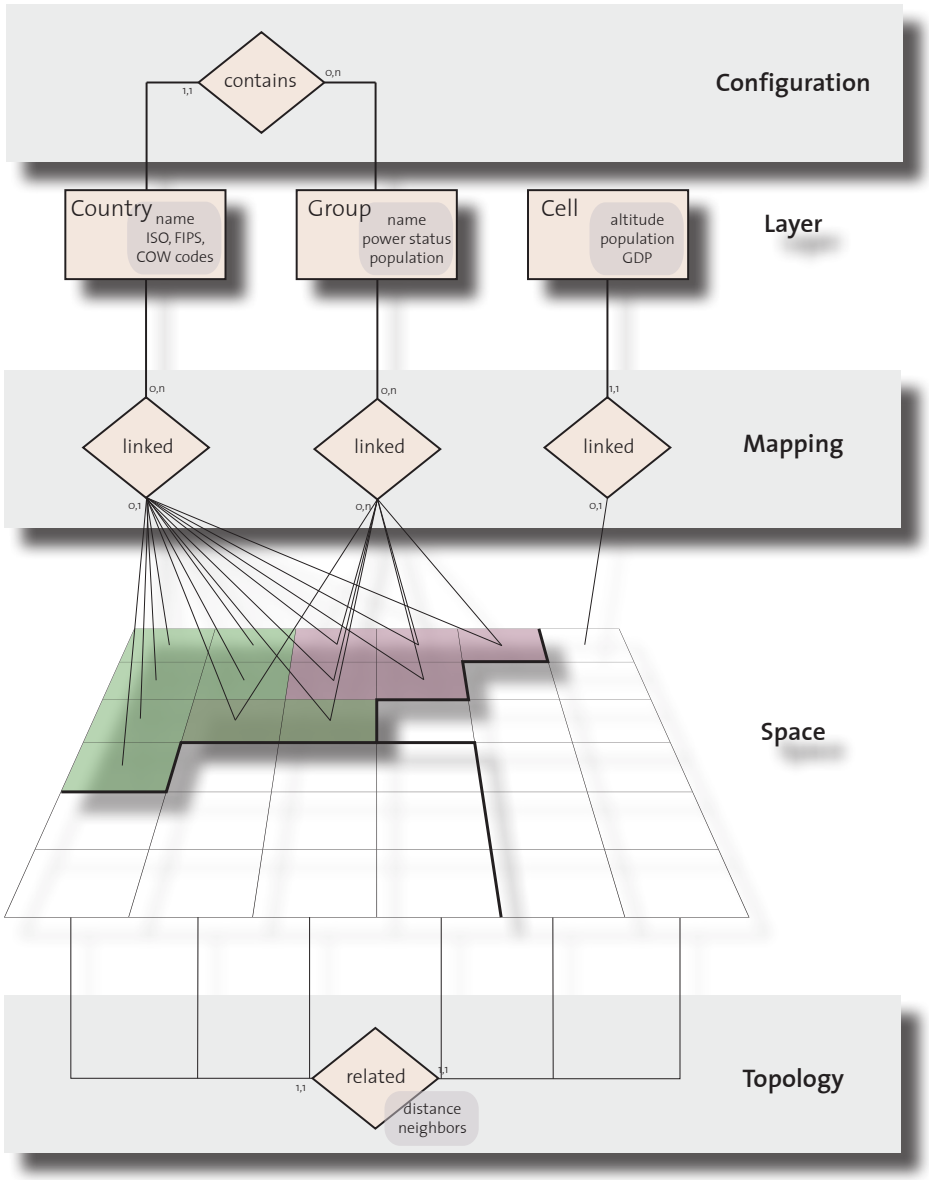
Spaces and mappings

In order to represent agents in a spatial environment, we distinguish between a space which is an empty set of locations, and a mapping which takes care of the assignments of agents to locations in this space.

This flexible design allows us to put agents at more than one position (e.g. states can occupy more than one province in a grid), or even to use one space for many different mappings.

For example, this is useful when representing the extent of states and ethnic groups in the same geographic space: Only one space object is required, whose locations are then linked in two mappings.

Data structures (2/2)



Layers (1/2)

A layer is a container for a set of alike and atomic agents. Layers offer general functionality to manage the agents contained in them, but can also be used to collect aggregate data about the entire population. A layer itself does not know about the neighborhood relations of its agents – instead, this is achieved by imposing one or more topologies on a layer, as it will be seen later.

Here is an example of how a layer can be used to store 100 agents to later iterate through them in a random order:

```
public class MyModel extends AbstractModel {
    MyLab sim;
    UniformDistribution uniform;
    MutableLayer<MyAgent> agents;
    ...

    public void buildModel(Simulator simulator) {
        uniform = new UniformDistribution(
            new MersenneTwister(lab.seed.getValue(this)));
        agents = new SimpleLayer<MyAgent>(MyAgent.class);
        for(int i = 0; i < 100; i++) {
            agents.add(new MyAgent());
        }
    }

    public void step(Simulator simulator) {
        for(MyAgent agent: agents.randomIterator(uniform)) {
            agent.doSomething();
        }
    }
}
```

Layers (2/2)

The Layer interface is extended by the MutableLayer interface, which defines the contract for a layer that can be changed. The main methods of both interfaces are summarized below:

Layer<E>

getElement(int): Returns the element at the specified position.

getNumElements(): Returns the number of elements contained in this layer.

getElement(UniformDistribution): Returns a random element.

orderedIterator(): Returns an iterator over the elements in their defined order.

randomIterator(UniformDistribution): Returns a random iterator over the elements contained in this layer.

contains(E): Returns true if this layer contains the specified element.

MutableLayer<E>

add(E): Appends the specified element to the end of this layer.

remove(E): Removes the specified element from the layer, if it is present.

clear(): Removes all of the elements from the layer.

shuffle(UniformDistribution): Randomize the order of the elements.

Topology

A topology is always defined on a layer of agents and defines a set of neighborhood relationships between them. In this sense, a topology is equivalent to a network. Based on the connectivity between agents, it can compute the neighborhood set of a given agents as well as their distance from each other.

Usually, topologies are not instantiated directly but are obtained through spaces and mappings, which extend the Topology interface.

Key methods

getNeighbors(E): Returns the neighbors of an element.

getDistance(E, E): Returns the distance between two elements.

Configuration (1/2)

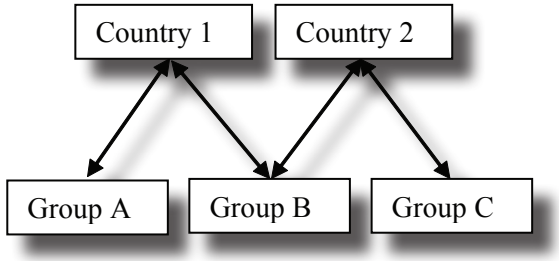
Whereas topologies can only exist between agents of the same kind, GROWLab offers the possibility to connect agents of different types to yield agent hierarchies.

This is done using configurations, which typically connect agents from two layers – the parent layer and the child layer, as we call it in GROWLab.

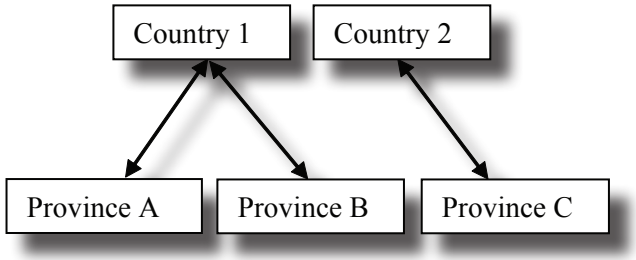
Configurations exist in different forms. The most general one is the many-to-many configuration, which allows the connection of a parent to many children, but also of a child to many parents.

A more restrictive configuration is the one-to-many type, relating one parent to many children, but permits at most one parent per child.

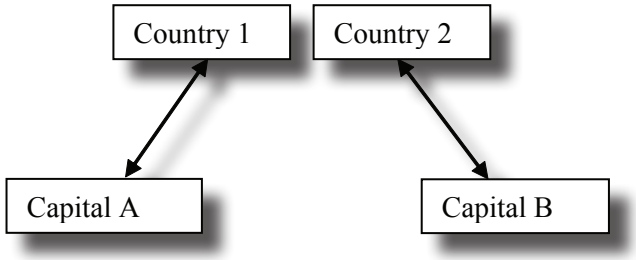
The one-to-one configuration adds the final constraint of only allowing exactly one child per parent.



A Configuration *where ethnic groups are linked to the states they live in.*



A OneToManyConfiguration *with countries linked to the provinces they occupy.*



A OneToOneConfiguration *with states linked to their capitals.*

Configuration (2/2)

The Configuration and MutableConfiguration class hierarchies are as follow, from the more general to the more specific:

Configuration<P, C>

getParents(C child): Returns the parents of a child element.

getChildren(P parent): Returns the children of a parent element.

MutableConfiguration<P, C>

link(P parent, C child): Creates a link between a parent and a child element.
unlink(P parent, C child): Removes a link between a parent and a child element.

OneToManyConfiguration<P, C>

getParent(C child): Returns the unique parent of child element

MutableOneToManyConfiguration<P, C>

- no additional method -

OneToOneConfiguration<P, C>

getChild(P parent): Returns the unique child of a parent element

MutableOneToOneConfiguration<P, C>

- no additional method -

Spaces (1/2)

GROWLab provides different types of spaces. On the one hand, it supports abstract spaces such as grids and hexagonal spaces. On the other hand, there is support for spaces with and explicit geographic reference, for example a GIS rastered space. Here, a location not only knows its x- and y-coordinates, but also its precise coordinates in latitude/longitude. Moreover, geographic spaces can compute the geodesic distance between locations.

Spaces implements both the Layer and Topology interfaces presented earlier. The easier way to instantiate a Space object is to use the following factory method:

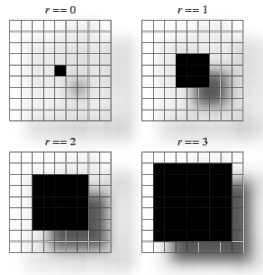
```
TwoDSpaceFactory.createSpace(int worldXSize, int worldYSize,
    Topology topology,
    Neighborhood neighborhood, int radius)
```

Topology choices

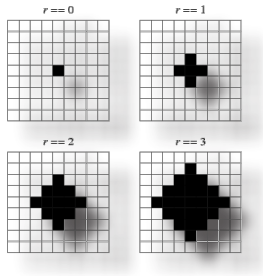
- Soup: every location is connected to every other location.
- Grid: a grid with no wrap-around.
- Torus: a grid with wrap-around.
- Jail: every location is isolated and has therefore no neighbor.
- Hexa: an hexagonal tiling of the space.
- Earth: a space representing the world.

Neighborhood choices

- Moore



- VonNeumann



Spaces (2/2)

Here is an example of how the various parameters of the TwoDSpace.createSpace() method can be obtained using input parameters:

```
public class MyLab extends AbstractLab<MyModel> {
    TwoDSpaceFactory.Parameters spaceParameters =
        new TwoDSpaceFactory.Parameters();
    ...

    public MyLab() {
        spaceParameters.worldSizeX.setDefaultValue(16);
        spaceParameters.worldSizeY.setDefaultValue(16);
        spaceParameters.topology.setDefaultValue(
            TwoDSpaceFactory.Topology.Grid);
        spaceParameters.neighborhood.setDefaultValue(
            TwoDSpaceFactory.Neighborhood.Moore);
        spaceParameters.radius.setDefaultValue(2);

        addParameters(stoppingTime, seed, intolerance);
        addParameters(spaceParameters);
    }
    ...
}
```

Mapping (1/2)

The mapping package follows the same principles as the configuration package, but links locations in space to elements of a layer. Here is a typical example of how to initialize a mapping:

```
public class MyModel extends AbstractModel {
    ...
    TwoDSpace<TwoDLocation> space;
    MutableOneToOneMapping<MyAgent, TwoDLocation> mapping;
    ...

    public void buildModel(Simulator simulator) {
        ...
        space = TwoDSpaceFactory.createSpace(
            lab.worldSizeX.getValue(this),
            lab.worldSizeX.getValue(this),
            lab.topology.getValue(this),
            lab.neighborhood.getValue(this),
            lab.radius.getValue(this));
        mapping =
            new SimpleOneToOneMapping<MyAgent, TwoDLocation>(
                agents, space);

        for (MyAgent agent: agents.orderedIterator()) {
            mapping.link(agent, mapping.getVacantLocation(uniform));
        }
    }
}
```

Mapping (2/2)

The Configuration and MutableConfiguration class hierarchies are as follow, from the more general to the more specific. Please note that the Mapping interface extends the Topology interface, and therefore provide its functionality as well.

Mapping<E, L>

getElements(L location): Returns the elements at a given location.

getLocations(E element): Returns the locations of a given element.

getVacantLocations(): Returns all the vacant locations.

getVacantLocation(UniformDistribution): Returns a random vacant location.

MutableMapping<E, L>

link(E e, L l): Creates a link between a parent and a child element.

unlink(E e, L l): Removes a link between a parent and a child element.

OneToManyMapping<E, L>

getElement(L location): Returns the single element at the given location.

MutableOneToManyMapping<E, L>

- no additional method -

OneToOneMapping<E, L>

getLocation(final E e): Returns the single location of an element

MutableOneToOneMapping<E, L>

- no additional method -

Activation regimes

GROWLab provides a scheme that allows the developer to set the execution order for the agent actions. The execution order determines whether the behaviors of the agents in the model are performed synchronously or asynchronously. So, for example, let us say that all of the agents have the following set of rules:

1. Play with the agent neighbors
2. Move to a new spot.

If the execution order is set to Synchronous, then the agents will perform their behaviors synchronously. Each agent will pick a random neighbor. Every agent will play with its neighbors. Once every agent has done this, then every agent will move to a new spot. So, in this case, all agents perform their rules in synchronization (synchronously). Everyone is playing and moving at the same time. If the execution order is set to Sequential or Democratic, then an agent will play with its neighbors and move to a new spot. Then another agent will also perform this set of actions. This continues until all agents have performed the full set of actions, and then a new iteration with the first agent again. In this case, the agents perform their rules asynchronously, since an agent (B) selected later in the iteration may actually choose to play an agent (A) that had been selected earlier in the iteration, and has recently moved into B's neighborhood.

Specifying the execution order simply involves creating the appropriate Processor object.

Processes and intentions

A process encapsulates an atomic action to be performed. The action is not directly performed, but is expressed in term of intention. An intention gives the possibility to check for conflict before being carried out. Here is an example process that will move the agent to a random vacant location:

```
public class MoveProcess implements Process<MyAgent> {
    MyModel model;

    public MoveProcess(MyModel model) {
        this.model = model;
    }

    public Intention getIntention(Simulator simulator, MyAgent
element) {
        TwoDLocation l = model.mapping.getVacantLocation(
            model.uniform);
        return model.mapping.getLinkIntention(element, l);
    }
}
```

Processors

You can then create the appropriate processor to carry out the provided process(es):

```
public class MyModel extends AbstractModel {
   Processor<MyAgent> processor;
   UniformDistribution uniform;
   ...

   public void buildModel(Simulator simulator) {
        ....
        processor = ProcessorFactory.createProcessor(
            lab.regime.getValue(this), uniform, new MoveProcess(this));
    }

   public void step(Simulator simulator) {
       processor.process(simulator, mapping.getLayer());
   }
}
```

Type of Processors

- Synchronous
- Sequential
- Democratic

Conflict resolutions

In the synchronous activation regime, a conflict resolution algorithm will automatically discard intentions that conflict with one another (e.g. if two agents want to move to the same location).

Collecting data

The way to extract statistics from a GROWLab model is done with the help of so-called “collector” classes. This mechanism is very flexible and can be used both for visual and batch models. Collectors are standardized data collection facilities storing the data in the format required for the analysis. For example, in a batch run one will typically use file-based collectors which simply output the assembled data to a file. For visual simulations, GROWLab offers collectors which prepare the data for display in a chart. Similarly, we provide a collector outputting a sequence of image files which can then be assembled to an animation of the simulation. Collectors are registered in the simulator executing the simulation. It takes care of activating the collector after each tick or at the end of a run.

Collectors get their data from variables within the simulation. However, as for the parameters introduced earlier, variables are implemented using the wrapper classes offered by GROWLab. Beyond the storage of a value these classes add meta-information about the variables such as their name and description. Additionally, variables can also be computed on the fly.

Output variables

Computation of derived output variables can easily be encapsulated in anonymous class to be created in the simulation class. Here is an example that will output the total number of neighbors in the mapping:

```
public class MyLab extends AbstractLab<MyModel> {
    IntegerVariable numNeighbors = new IntegerVariable<MyModel>(
        Integer.class, "numNeighbors") {
        public Integer getValue(MyModel model) {
            int neighbors = 0;
            for(MyAgent agent: model.agents.orderedIterator()) {
                neighbors += model.mapping.getNumNeighbors(agent);
            }
            return neighbors;
        }
    };

    ...
}
```

Collectors

Collectors are usually setup by overriding the setup(Simulator) method of the simulation class. Collectors should be registered to the simulator, depending whether data collection should be made when the simulation start, end, as well at every time tick. Here is an example to record 4 variables and parameters into a file named mymodel.txt:

```
public class MyLab extends AbstractLab<MyModel> {
    ...

    public void setup(Simulator simulator) {
        FileCollector collector = new FileCollector(
            new File("mymodel.txt"),
            seed, currentTime, regime, numNeighbors);
        simulator.setStartCollectors(collector);
        simulator.setStepCollectors(collector);
        simulator.setFinishCollectors(collector);
    }

    ...
}
```

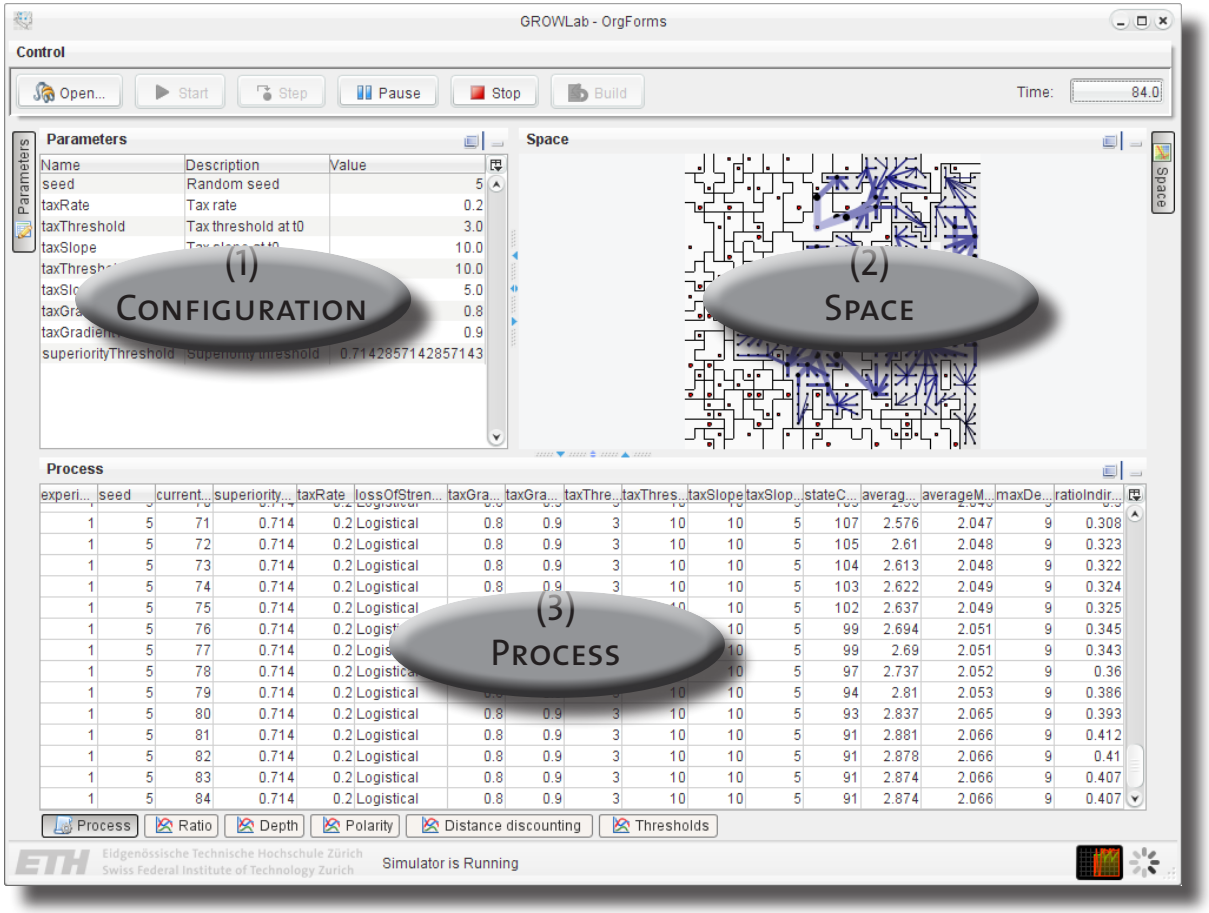
Common collectors

- FileCollector
- ConsoleCollector
- ImageCollector
- TableModelCollector
- XYSeriesCollectionCollector

Creating views

When a graphical simulator is created, the buildUI(VisualSimulator) method of your Lab class will be called. You can therefore override this method to add views to display some aspect of your model, for examples, tables, spatial representations, and charts.

While GROWLab provide some common views that can easily be customized, it is also possible to create plain Swing components, giving you full flexibility and control over every display and interaction aspects.
The views can be docked into one of the following area:



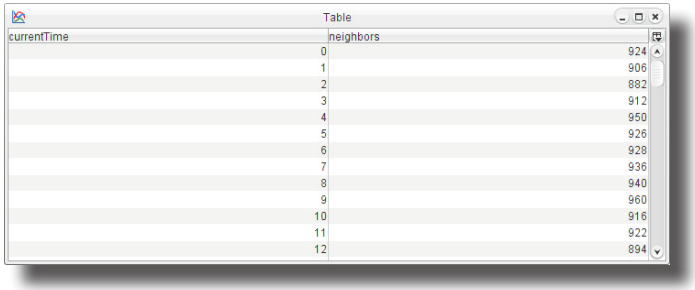
Tabular views

Instead of collecting data into a file, the information can directly be displayed into GROWLab. You can adapt the following code to display any number of variables and parameters, of any type, into a table.

```
public class MyLab extends AbstractLab<MyModel> {
    ...

    public void buildUI(MyModel model, VisualSimulator simulator) {
        TableModelCollector tableCollector = new TableModelCollector(
            currentTime, numNeighbors);
        simulator.setStartCollectors(tableCollector);
        simulator.setStepCollectors(tableCollector);
        simulator.addProcess("Table",
                new JScrollPane(ViewFactory.createTable(tableCollector)),
                IconFactory.getTableEditIcon());
    }
    ...
}
```

The example above will produce the following table when run:



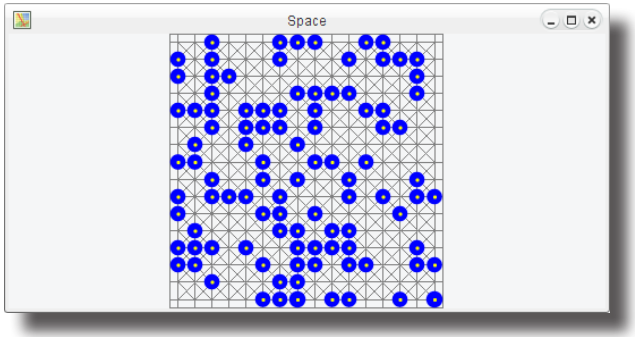
Spatial views

Creating a spatial view involves first creating the view itself and then adding some renderers to it to parametrize how the agents should be displayed:

```
public class MyLab extends AbstractLab<MyModel> {
    ...

    public void buildUI(MyModel model, VisualSimulator simulator) {
        TwoDSpaceView spaceView =
            ViewFactory.createTwoDSpaceView(model.space, simulator);
        OneToOneMappingViewRenderer<MyAgent> lvr =
            new OneToOneMappingViewRenderer<MyAgent>(
                "Identity", model.mapping, spaceView);
        spaceView.addRenderer(
           new NeighborhoodViewRenderer(spaceView));
        lvr.addRenderer(new MyAgentRenderer());
        spaceView.addRenderer(lvr);
        simulator.addSpace("Space", new AspectJPanel(spaceView),
            IconFactory.getMapIcon());
    }
    ...
}
```

The code above will produce the following display:



Charting

GROWLab provides ways to create various different charts, which can be adjusted to some particular needs thanks to the underlying JFreeChart library. Here is a an example of how to create a time series chart:

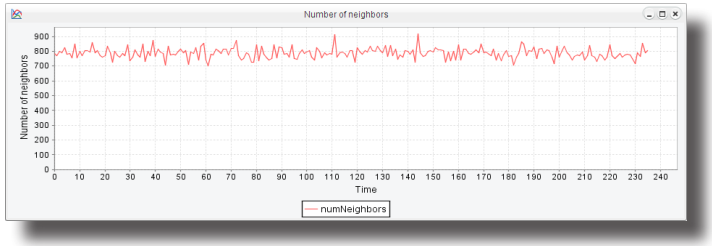
```
public class MyLab extends AbstractLab<MyModel> {
    ...

    public void buildUI(MyModel model, VisualSimulator simulator) {
        TableModelCollector tableCollector =
            new TableModelCollector(currentTime, numNeighbors);
        XYSeriesCollectionCollector timeSeries =
            new XYSeriesCollectionCollector(
                "Number of neighbors", numNeighbors);
        simulator.setStartCollectors(tableCollector, timeSeries);
        simulator.setStepCollectors(tableCollector, timeSeries);

        simulator.addProcess("Number of neighbors",
                new ExtendedChartPanel(
                    ViewFactory.createTimeSeriesChart(timeSeries)));
    }

    ...
}
```

And here is the result once the simulation has been run for a little while:

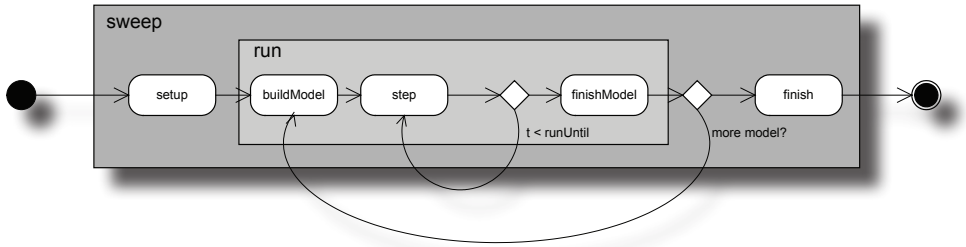


Simulators

A simulator has the ability to run a simulation. Both the visual simulator and the batch simulators provides all the functions of the Simulator interface, while the VisualSimulator has additional methods for docking components to the user interface. The easiest way to create an entry point to the simulator is to create it using the following factory method, which is optimally placed into the simulation class:

```
public static void main(String[] args) {
    SimulatorFactory.createSimulator(new MyLab(), args).setup();
}
```

This will setup the simulation, which will then have the following life cycle:



Simulator

- setup()
- start()
- step()
- stop()
- setStartCollectors(Collector...)
- setStepCollectors(Collector...)
- setFinishCollectors(Collector...)

VisualSimulator

- addSpace(String title, JComponent space, ImageIcon icon)
- addConfiguration(String title, JComponent configuration, ImageIcon icon)
- addProcess(String title, JComponent process)

Parameter sweep

With ease, GROWLab let you schedule multiple simulation runs to be executed. This is especially useful to test the sensitivity of a model, as well as to search for optimal parameter values.

To define the runs to be performed, simply loop through all the combinations of values you want to test and use the addRun(ParametersSetters) method to add them to the laboratory. Here is an example to define 500 runs to be performed using a combination of two parameters, for a duration of 1000 ticks:

```
public class MyLab extends AbstractLab<MyModel> {
    IntegerParameter seed = new IntegerParameter(
        "seed", "Random seed", 0);
    DoubleParameter intol = new DoubleParameter(
        "intol", "Intolerance ratio", 0.33);

    public MyLab() {
        addParameters(stoppingTime, seed, intol);

        stoppingTime.setDefaultValue(1000.0);
        for (double intol = 0.0; intol <= 1; intol += 0.1) {
            for (int seed = 1; seed <= 50; seed++) {
                SimpleParametersSetters run =
                    new SimpleParametersSetters();
                run.addParameter(this.seed, seed);
                run.addParameter(this.intol, intol);
                addRun(run);
            }
        }
    }
```

Running batch simulations

GROWLab include support for performing batch simulation runs. This is useful when you want to run a simulation without the extra burden of graphical displays or when you have a large set of parameters you wish to run your model against.

Note that GROWLab batch simulator will take advantage of SMP systems and multicore microprocessors by running multiple simulation runs in parallel.

Using IntelliJ IDEA

1. Go to *Select Run/Debug Configuration→Edit configurations*
2. Select the model you want to run in batch mode
3. Add batch to the Program parameters and hit *Ok*.

From the command line

To start a batch run from the command line (typically when you deployed a model on a remote machine), add the --batch parameter:

```
java -jar lib/growlab.jar --batch
    mymodel
```

GeoModel template model (1/2)

Based on the template model GeoModel, geo-coded real-world data can be integrated in the modeling process. This template can be extended by inheriting the built-in functionality and by adding some custom behaviors and mechanisms or complement it with additional layers of data.

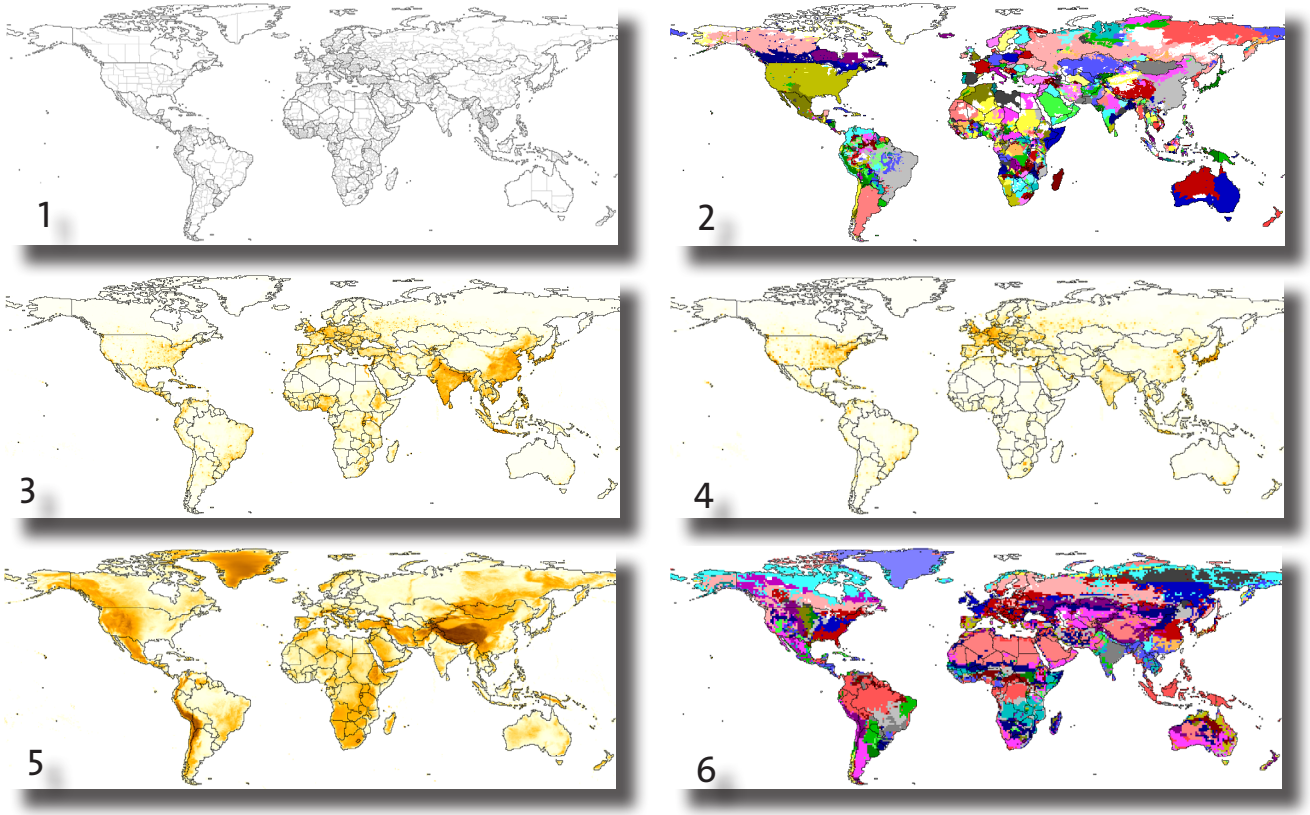
GeoModel's default space is a rasterized representation of the entire globe, using the WGS84 projection. The raster can be used in two different resolutions: 15 arc-minutes (~30km), and 30 arc-minutes (~60km). All the geographic data is based on this space.

In addition, we provide disaggregated data for every cell in the system for population (downsampled from the Gridded Population of the World v. 3 provided by CIESIN (2005)) and elevation (downsampled from GTOPO30 (2007)), as well as local GDP estimates, compiled by the G-Econ project (at a 1-degree resolution) from Nordhaus (2006). They relieve the modeler from the tedious task of having to collect and merge complicated datasets and thus provide a prototyping environment for geographic agent-based models.

To GeoModel template model does not include any behavior. However, behaviors can be added by extending the AbstractGeoModel and by providing your implementation of the Cell, Capital, Country, Identity, Group, and Region interfaces.

GeoModel template model (2/2)

As illustrated below, some of the information contained in the GeoModel template includes, from left to right, (1) country border and administrative divisions, (2) ethnic groups across countries, (3) population density, (4) spatial GDP figures, (5) elevation data, and (6) vegetation type. For each country, we provide their borders as of 1964 and 1994, and also try to reconcile their ISO, FIPS and COW codes through customized mapping. To check adjacency of countries, the Minimum Distance data from Gleditsch and Ward is also included to query for neighboring countries that are separated by water. At this point, all ethnic groups are directly based on the GREG definitions. For each ethnic group in a country, there is also information about the "ethnic group in power" (EGIP) coding by Cederman and Girardin.



Packaging

It may be useful to be able to package your model for the following reasons:

- To easily allow other people to run your model
- To deploy your model on a remote machine so that it can be executed in batch mode.
- To archive a version of your model so that the results can be reproduced.

To accomplish all of these, GROWLab provide a mean of packaging your model in the form of a plugin that can dynamically be found at run time. This is achieved by packaging the model classes with a XML descriptor into a Zip file. This Zip file can then be placed into the models directory of the GROWLab installation, which is typically structured as follow:

```
GROWLab/
    javadoc/
    jre/
    lib/
    models/
        mymodel-1.0.0.zip
        ...
```

Creating descriptors

So that GROWLab automatically detect your model, a XML descriptor called plugin.xml needs to be present at the root of your module. Here is an example content, with the values to be adjusted highlighted:

```
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 1.0" "http://jpf.
sourceforge.net/plugin_1_0.dtd">
<plugin id="mymodel" version="1.0.0" class="ch.ethz.icr.grolab.models.
mymodel.MyLab">
    <requires>
        <import plugin-id="ch.ethz.icr.growlab.core"/>
    </requires>
    <runtime>
        <library id="mymodel" path="classes/" type="code">
            <export prefix="*"/>
        </library>
    </runtime>

    <extension plugin-id="ch.ethz.icr.growlab.core" point-id="Model"
id="mymodel">
        <parameter id="class" value="ch.ethz.icr.growlab.models.mymodel.
MyLab"/>
        <parameter id="name" value="mymodel"/>
        <parameter id="icon" value="icon12x12.png"/>
        <parameter id="description" value="My Model"/>
    </extension>
</plugin>
```

Bundling a simulation model

Model classes and optionally source code can be bundled together to facilitate their redistribution. This is done through a simple Zip file that can be placed into the models directory of GROWLab. GROWLab will automatically detect it upon startup.

The naming convention for the Zip file name is mymodel-x.x.x.zip where mymodel is the name of the model and x.x.x is the version of the model.

The root of the Zip file should contain the plugin.xml XML descriptor file at the root. The classes should be placed in the classes directory and the source code optionaly in the src directory.

A typical organization of the Zip file can be structured as follow:

```
mymodel-1.0.0.zip
    plugin.xml
    growlab-mymodel.iml
    classes/ch/ethz/icr/growlab/models/mymodel
        MyLab.class
        MyModel.class
    src/ch/ethz/icr/growlab/models/mymodel
        MyLab.java
        MyModel.java
```

Credits

GROWLab would not exist without the contribution of a lot of people. Thanks to all of them for their continuous support!

Lead developers

- Luc Girardin
- Nils Weidmann

Contributors

- Lars-Erik Cederman
- Lutz Krebs
- Lucas Serba Silva
- Jan Alsenz