OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models

Romain Reuillon[a,b], Mathieu Leclaire[a,b], Sebastien Rey[a,b]

[a] *Géographie-cités - UMR 8504*
13 rue du Four
75006 Paris
[b] *Institut des Systémes Complexes*
57-59 rue Lhomond
75005 Paris

Abstract

Complex-systems describe multiple levels of collective structure and organization. In such systems, the emergence of global behaviour from local interactions is generally studied through large scale experiments on numerical models. This analysis generates important computation loads which require the use of multi-core servers, clusters or grid computing. Dealing with such large scale executions is especially challenging for modellers who don't possess the theoretical and methodological skills required to take advantage of high performance computing environments. That's why we have designed a cloud approach for model experimentation. This approach has been implemented in OpenMOLE (Open MOdel Experiment) as a Domain Specific Language (DSL) that leverages the naturally parallel aspect of model experiments. The OpenMOLE DSL has been designed to explore user-supplied models. It delegates transparently their numerous executions to remote execution environment. From a user perspective, those environments are viewed as services providing computing power, therefore no technical detail is ever exposed. This paper presents the OpenMOLE DSL through the example of a toy model exploration and through the automated calibration of a real-world complex system model in the field of geography.

Keywords: Model exploration, Distributed computing, Workflow, Cloud computing, Complex-systems

Introduction

A complex-system can be defined as a "system comprised of a great number of heterogeneous entities, among which local interactions create multiple levels of collective structure and organization"[1]. The emergence of the collective structure from numerous local interactions is generally unpredictable analytically. Therefore scientists use simulation models as a medium to study complex-systems. Like the physical system they represent, the behaviour of

such models are unpredictable and counter intuitive. That's why large scale numerical experimentation is required in order to understand how patterns emerge from one scale to another.

Complex-system models are often multi-scale, stochastic and individual centred. Therefore, their execution is generally computationally intensive. Furthermore, the numerical experimentation on such models might imply million of executions [2, 3]. This huge computational load can only be carried out by high performance computing environments.

Dealing with such broad computational loads is brain consuming, technically tricky, error prone and far from the specific field of expertise of modellers. Hopefully, most model exploration algorithms expose a naturally parallel aspect: the large number of independent executions of the model is with no doubt the most computationally intensive part. In this paper we describe how we leveraged this natural parallelism to design a generic formalism for distributed experimentation on complex-system models. This formalism has been implemented in a platform called OpenMOLE (Open MOdeL Experiment)[1], which provides a convenient way to explore home-brewed models with quickly evolving implementations using advanced design of experiments. The contributions of OpenMOLE are twofold: it exposes a language for describing reusable design of experiments for simulation models and it provides an execution platform which distributes these experiments on high performance computing environments in a transparent manner.

This paper demonstrates the central concepts of the OpenMOLE formalism. This platform is mature and used daily to explore real-life complex-system models. However for the sake of the comprehension of this paper, OpenMOLE's concepts are illustrated here by the exploration of a toy complex-system model. The first section presents the goal of the platform. Then the test model is exposed. After that, this model is explored through several numerical experiments of increasing complexity. Finally, the last section describes a real case experiment on a multi-agent geographical model.

1. Distributed model exploration

1.1. The naturally parallel aspect of model experimentation

In the physical world, experimenting on complex systems (such as: human societies, neural networks, insect swarms..) is generally impossible, unethical or very costly. That's why scientists design numerical models in order to facilitate the study of such systems. The numerical modelling of a complex phenomena especially eases the experimentation required to understand how general patterns emerge from local interactions to global behaviour. The experiments are thus achieved in-silico, according to methods designed for this purpose.

Among the available methods for numerical experiment on models, one of the classics is the Design of Experiments (DoE) based on statistics [4]. DoE

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

have been widely studied to produce sensitivity analysis on deterministic and stochastic simulations [5]. Most DoE generate a set of samplings values for the input of the model. Each of them is evaluated through one or several model executions depending on the stochastic nature of the model. In DoE, each evaluation is independent and constitutes a naturally parallel aspect.

Apart from DoE other methods have been designed to automate the model calibration phase. Indeed, during the modelling process some parameters might lack an empirical value. They are thus fixed during a calibration phase. This task is time consuming, that's why some methods have been designed to automate it. They are generally based on an optimization algorithm (like in [6]) that aims at minimizing the difference between the model behaviour and an expected behaviour. Automated calibration algorithms repeatedly evaluate the model for various set of parameters in order to find a global minima. Evaluating several configurations in parallel speeds up the optimisation process, therefore automated calibration processes can also be considered as naturally parallel.

More recently, novel methods have been developed to explore fitness landscape, based on particle swarm optimization [7] and on genetic algorithms [8]. These methods are particularly well suited to map model dynamics. Swarm optimisations and genetic algorithms both consider a population of solutions. The evaluation of the fitness of the individuals of the population can be processed in a concurrent manner. Once again, that kind of method exposes a naturally parallel aspect.

Another example of this naturally parallel aspect of the model exploration methods arises from a recent work of ours. We have applied viability theory[9] to explore all possible dynamics of a model under a given set of constraints (this work is presented in [2]). In this experiment the model is evaluated millions of times. Most of these evaluations are independent from each others, exposing once again a naturally parallel aspect.

The list of methods in this section is not exhaustive, yet it shows that when dealing with model exploration many methods expose naturally parallel aspects. This natural parallelism concerns the numerous executions of the model which are required in order to understand its dynamics. This aspect is however rarely leveraged to enhance computation speed. It is even called embarrassingly parallel by some authors.

1.2. A generic platform for distributed model experimentation

Using distributed execution environment efficiently requires methodological and technical skills. People possessing those skills are generally not present in the community where model experimentation at large scale would be required. That's why we have designed a platform that completely hides the burden of distributed computing for model exploration.

The platform called OpenMOLE is based on a workflow formalism. This formalism is very suitable for representing parallel processes. Several platforms already propose workflow engines for scientific computing (for a review on scientific workflow platforms see [10]). The most popular free and open-source ones

are Kepler [11] [2], Taverna [12] [3], Triana [13] [4], Pegasus [5] [14] and P-Grade [15] [6]. Unlike those platforms, the design of OpenMOLE has been focused on automating distributed experiments on complex-systems simulation models. This requirement has led to a significantly different design.

The design of OpenMOLE has been driven by the practices of the complex system modellers. For instance, the quickly evolving implementations of complex-system models, in heterogeneous languages, and the need to experiment all along the modelling process made it crucial to easily embed continuously changing user software components in the platform. Other workflow platforms can generally use external calls to users-provided programs but they are not able to delegate them transparently to a remote execution environment. In OpenMOLE, we have made it easy to embed models based on diverse languages (such as all JVM languages (java, scala, groovy, clojure), NetLogo [7], native executables (C / C++ / Python / Fortran / Scilab / Octave...), etc) and to delegate their executions to remote execution environments.

To delegate the model executions, OpenMOLE enforces a cloud approach. Following the cloud concepts, it exposes remote execution environments as if they were services that provide computing power. It accesses them without exposing any technical specificities to the user. To do so, the workload is delegated transparently directly from the user computer to remote execution environments [16] following a zero-deployment approach: required software components are installed transparently and on-the-fly.

Finally, the necessity of carrying advanced design of experiments, such as genetic algorithms or iterative refinement of the exploration space, has pushed the design of the OpenMOLE toward a very flexible workflow formalism. This formalism provides processing structures which are not yet available in many other workflow platforms: cycles (loops), conditional branching, nested workflows and implicit representation of massively parallel workflows.

2. The OpenMOLE domain specific language

2.1. The toy model

In this paper, the key aspects of OpenMOLE are demonstrated through the exploration of a toy model in NetLogo. However, OpenMOLE handles models as black-boxes, therefore everything presented here is transposable to other models in other languages.

The Fire model, shown on 1, "simulates the spread of a fire through a forest. It shows that the fire's chance of reaching the right edge of the forest depends

[2] http://kepler-project.org
[3] http://www.taverna.org.uk
[4] http://www.trianacode.org
[5] http://pegasus.isi.edu
[6] http://portal.p-grade.hu
[7] http://ccl.northwestern.edu/netlogo

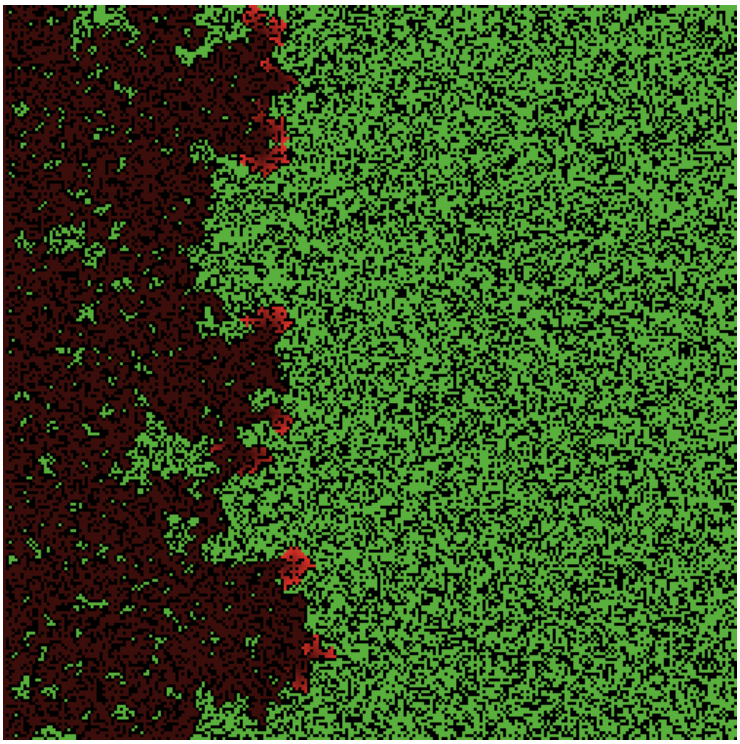

Figure 1: Visualisation of the fire model in NetLogo.

critically on the density of trees. This is an example of a common feature of complex systems, the presence of a non-linear threshold or critical parameter" [8].

This model is simpler that most of model we usually work with. However it is representative enough to show how it is possible to use OpenMOLE to explore a model.

Listing 1: Embed the Fire NetLogo model in OpenMOLE

```
1  // Declaration of the variables
2  val seed = Prototype[Int]("seed")
3  val density = Prototype[Double]("density")
4  val burned = Prototype[Double]("burned")
5
6  // Describe the model task
7  import org.openmole.plugin.task.netlogo5._
8
9  val fire =
10   NetLogo5Task(
```

[8] http://ccl.northwestern.edu/netlogo/models/Fire

```
11      "Fire",
12      "Fire.nlogo",
13      List("random-seed ${seed}", "setup", "while [any? turtles] [
            go]")
14    )
15
16  // Describe input and output of the task
17  fire addInput seed
18  fire addNetLogoInput (density, "density")
19  fire addNetLogoOutput ("burned-trees", burned)
20
21  // Describe values for inputs
22  fire addParameter (density, 59.0)
23  fire addParameter (seed, 42)
24
25  // Describe the display hook
26  import org.openmole.plugin.hook.display._
27
28  val display = ToStringHook()
29
30  // Describe the workflow
31  val execution = (fire hook display) toExecution
32  execution start
```

To explore models, OpenMOLE exposes a DSL (Domain Specific Language) for distributed model exploration. This DSL has been built on top of the Scala language [9]. The Scala syntax has been extended in order to specify tasks and transitions which are the central concepts in a workflow. The script of the Listing 1 exploit the OpenMOLE DSL [10] to execute the model a single time and displays the output density; it is directly executable in the console of OpenMOLE version 0.8.

The first part of the script declares variables. Those variables constitute the data-flow, which can be exchanged between tasks along the transitions. OpenMOLE variables are typed, which makes OpenMOLE worflows statically typed. As a result, formal verification of the data-flow is enforced before its execution. This formal verification ensures the correctness of the data-flow, which is a exhausting task when performed manually on large workflows. This very important feature is novel, when compared to other workflow systems.

The type system of OpenMOLE is very complete as it supports native types (int, double, long...), files, directories and all java and scala types. Furthermore, additional user-defined types can be provided as plugins in the form of Scala or Java classes.

In the Listing 1, 3 variables are considered:

- the seed, an integer to initialize the pseudo-random number generator of the model,
- the input value of the model representing the densities of trees,

[9] http://www.scala-lang.org/

[10] http://www.openmole.org/documentation/console/api/

- the output value of the model representing the number of burned trees.

In workflows, a task is an atomic execution component. Several tasks are linked with each-others by transitions to design the workflow topology. More specifically, in OpenMOLE tasks are all portable, reentrant, immutable software components which can be run concurrently. This means that tasks have been designed so they have no interfering side effects. Therefore they can be safely dispatched on several threads, processes or computers. As far as we know, this conception of portable tasks is also a unique feature of OpenMOLE among all the workflow platforms. For us, this feature is crucial, as it makes it possible to delegate the workload entirely transparently from a user perspective, enabling the cloud aspect of OpenMOLE.

The concept of task is reflected in OpenMOLE by an interface (Scala trait). It is polymorphic and extensible. Each implementation provides a new service. Even if OpenMOLE is implemented in Scala, tasks have been designed to embed many kind of software components through a non intrusive black box approach. For instance:

- JVM tasks run user code on top of the Java Virtual Machine, this code is provided as byte-code packaged in jars (including Java, Scala, Groovy...),
- the Netlogo tasks run NetLogo [11] simulations,
- the system execution task runs native binary code,
- the workflow task runs nested OpenMOLE workflows...

The Listing 1 shows how to declare a task in OpenMOLE. After the variable declaration part, a task that embed models written in NetLogo version 5 is instantiated. This task has been called the "NetLogo5Task". At creation time, this task requires: a name, the path of the file containing the Netlogo model and the NetLogo commands that run the model. The file "Fire.nlogo", which is provided to OpenMOLE, is an unmodified version of the fire model distributed with NetLogo.

When executed, a task accesses variables provided by the data-flow. Those variable, which are required for the task execution, are called "input data". When execution of a task has been completed, it produces data injected into the data-flow. This data is called "output data". The output data of a task can be used as input data in subsequent tasks in the workflow.

Lines 16 to 19 of the listing describe input and output data of the NetLogo task. This task requires a seed and a variable called density. The density input is mapped to the density variable of the model. The variable seed is used in the launching commands to initialize the pseudo-random number generator of NetLogo. Once executed the task brings out the value of the NetLogo variable "burned-trees" which is mapped to the burnedTrees variable of OpenMOLE. Lines 21 to 23 set default values for the seed and the density.

[11] http://ccl.northwestern.edu/netlogo

Tasks have been designed to be free of side effects: like pure functions, they compute output values (return values) from input values (arguments). It is a requirement to make it possible to execute them in a concurrent manner. Therefore, output operations (display, writing results in files or database...) are performed by other components called "Hooks". Hooks listen to the task executions and perform output operations once a task's execution has ended. In the Listing 1, the lines 25 to 28 instantiate a hook that displays every variable produced by the NetLogo task.

At the end of the the Listing 1, the workflow is described. This workflow contains a single task, a hook, and no transition. Its execution produces the output: "{burned=12448.0}".

2.2. Tracing the distribution law of the output

Fire is a stochastic model. The "burned-trees" output is therefore a random variable. To draw the probability distribution of this variable many independent executions of the model should be carried out. The OpenMOLE platform benefits from our previous work on the DistME platform [17] to automatically distribute stochastic simulation. Thus, it can distribute the replications of a stochastic model in a rigorous manner. Computing independent replications in the NetLogo platform means initializing the seed of the pseudo-random number generator with different values prior to each run.

Listing 2: Explore the distribution of "burned trees" output variable

```
1  /* ... Description of the fire task ... */
2
3  //Describe the exploration task
4  import org.openmole.plugin.domain.distribution._
5  import org.openmole.plugin.domain.modifier._
6
7  val replication =
8    ExplorationTask(
9      "replication",
10     Factor(seed, UniformIntDistribution() take 100000)
11   )
12
13 // Describe the hook to write into a file
14 import org.openmole.plugin.hook.file._
15
16 val write = AppendToFileHook("result.txt", "${burned}\n")
17
18 // Describe the execution environment
19 import org.openmole.plugin.environment.glite._
20
21 val complexSystemsVO = GliteEnvironment("vo.complex-systems.eu")
22
23 // Import the job grouping strategy
24 import org.openmole.plugin.grouping.batch._
25
26 // Describe the workflow
27 val wf =
28   replication -< (fire hook write on complexSystemsVO by 500)
```

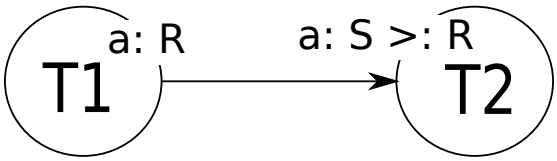

Figure 2: Single transition between two tasks.

```
29
30  val execution = wf toExecution
31  execution start
```

The worflow shown in Listing 2 distributes 100,000 replications of the fire model over the European grid EGI (European Grid Infrastructure). The first part of the script is identical to Listing 1, so it has been omitted.

An OpenMOLE workflow that replicates a model execution is generally composed of two tasks. The first task generates the seeds for the pseudo-random number generator and the second task uses the seed to execute the replications.

In a workflow tasks are linked with each-other by precedence constraints called "Transitions". Figure 2 represents a transition between two tasks: *T2* is executed once *T1* is completed. The typed variable values of the data-flow circulate along transitions. For instance in Figure 2, the task *T1* produces a variable value called a of type R. The execution of task *T2* depends on a variable a of type S. The type S should be the same type as R or a super-type of R, noted: $S >: R$.

In OpenMOLE, workflow level parallelism [18] is achieved by specifying several transitions from one task to several others. This formalism makes it possible to define independent tasks for which the executions can be can safely executed concurrently. This pattern is called "divergent transitions". This pattern is illustrated on Figure 3: in this workflow tasks *T2* and *T3* can be executed concurrently, however they both depend on a variable value produced by task *T1* that should be executed before *T2* and *T3*.

The dual of the divergent pattern is the "convergent transitions" pattern. In this pattern, several transitions lead to a single task. This formalism makes it possible to resynchronize execution streams which have previously been desynchronized by a divergent transitions pattern. In a convergent pattern, variables with the same name are aggregated in arrays. Each array is typed with the least common type (the more-specific common type) among all the super-types of the aggregated variables. On Figure 4, tasks *T1* and *T2* both produce a variable *a*. By consequence, the task *T3* receive a variable a of type array of R which is the least common super-type of S and U.

The three previously presented patterns (flat, divergent, convergent) are based on a single type of transition for which parallelism should be explicitly represented by workflow topology. In order to design massively parallel workflow executions in a concise manner, OpenMOLE proposes a notation to specify massive parallelism implicitly. This notation is called the "exploration transi-

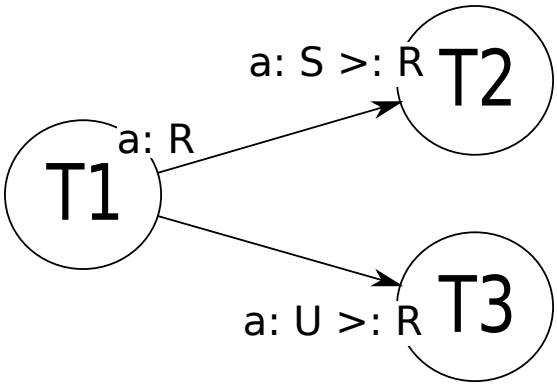

Figure 3: Divergent transitions pattern.

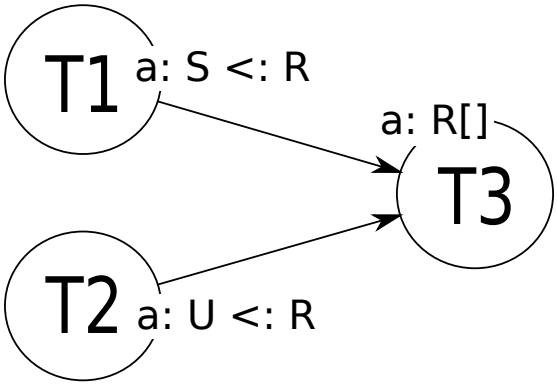

Figure 4: Convergent transitions pattern.

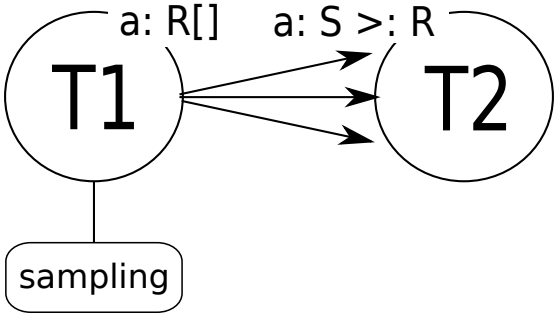

Figure 5: Exploration transition.

tion". It is exposed in Figure 5, in which two tasks are linked by an exploration transition. Task *T1* is a special task called "exploration task". It generates a set of values according to a strategy for generating samples. This strategy is called a "sampling". There are numerous sampling strategies implemented in OpenMOLE. Among other strategies, they provide common design of experiments such as full factorial sampling, latin hypercube sampling as well as the ability to read values in a file formatted with Comma Separated Values (CSV). Advanced users can even define their own samplings, thanks to the extensible design of OpenMOLE. In the workflow of the Figure 5, for each sample produced by the sampling of the task *T1* a new execution stream is created and an instance of task *T2* is scheduled for execution. When using distributed execution environments each execution of the task *T2* is run on a separated processing unit. The distribution of the many executions of a model generated by a design of experiments is a requirement to explore complex-system models. That's why, contrary to other workflow platforms, this kind of distribution has been made central in OpenMOLE.

In the OpenMOLE DSL transitions are represented by arrows. In Listing 2 the exploration task issues a sampling of 100,000 different seed values following a uniform distribution. Then, an exploration transition (noted $-<$) generates 100,000 independent executions streams, one for each seed. Each stream compute a replication of the model.

In the example exposed in the previous section, the result value was displayed on the standard output. In this workflow 100,000 values are generated and stored into a file. Lines 13 to 16 instantiate the hook that perform the storage operation. Each time a model execution is completed, the hook writes a new line containing the value of the variable "burnedTrees" in the file "result.txt".

This workflow computes 100,000 executions of the model. Even for fast models that would run in about 2s, the workflow would require 55 hours of computation on a single core to complete. To tackle the problem of the workload generated by model explorations OpenMOLE provides a seamless distribution of the computation load on remote computing resources. As written in [19]: "A lesson to be learned from Grids is that the abstractions that Grids expose to

the end user, to the deployers and to application developers are inappropriate and they need to be at a higher level." To solve this issue, OpenMOLE enforces a novel approach inspired from the cloud paradigm. As stated in the section 1.2, from the user point of view distributed execution environment are services providing computing power. From the internal OpenMOLE perspective, it autonomously discovers available resources and renders the management of data transfer, job execution and software installation transparent to the end user.

OpenMOLE enforces a Platform as a Service (PaaS) approach on top of environments that were not conceived to provide such a functionality (grids, clusters...). Apart from failure management, resource discovery and file transfers, one of the biggest problems is to hide the complexity of the applications deployment phase from the user. This has been solved using a zero-deployment approach that discovers and installs required execution components on-the-fly on the remote execution host, at execution time (more details about this aspect of OpenMOLE are exposed in the paper [16]). Zero-deployment is not provided by any other scientific workflow platform, however it is required in order to launch rapidly evolving home-brewed models of complex-systems.

Zero-deployment implies that a workflow is independent from any specific kind of execution environment. Its execution can scale from a laptop host up to multi-core severs, clusters and grids. Environments are switchable in a declarative manner as explained in [16]. For now, seven execution environments are implemented in OpenMOLE: local computer, multi-thread local computer, remote multi-core server, Portable Batch System (PBS) cluster, SLURM (Simple Linux Utility for Resource Management) cluster, Glite / European Middleware Initiative (EMI) and a versatile desktop grid based on a daemon distributed along with OpenMOLE. The support for other environments are being developed for cluster systems (Sun Grid Engine) and for the research (Stratus Lab [12]) and commercial IaaS (Infrastructure as a Service) providers.

The cloud aspect of OpenMOLE is taken advantage of in the example of Listing 2. The executions of the model are delegated to the virtual organization for complex-systems computation of the European grid EGI. Since execution environments are exposed as services by OpenMOLE, only access information should be provided. For EGI this information comprises only the name of the virtual organization.

As stated previously, massive parallelism is achieved by executing each model execution in an independent job on the execution environment. However for a fast model such as Fire, the overhead of several minutes implied by the grid submission system is not negligible. That's why grouping strategies have been implemented in OpenMOLE. They make it possible to run several model executions in a single grid job. For instance, in Listing 2, a grouping strategy is specified in the workflow description: the executions of the model are grouped by 500 and delegated to the grid.

[12] http://stratuslab.eu

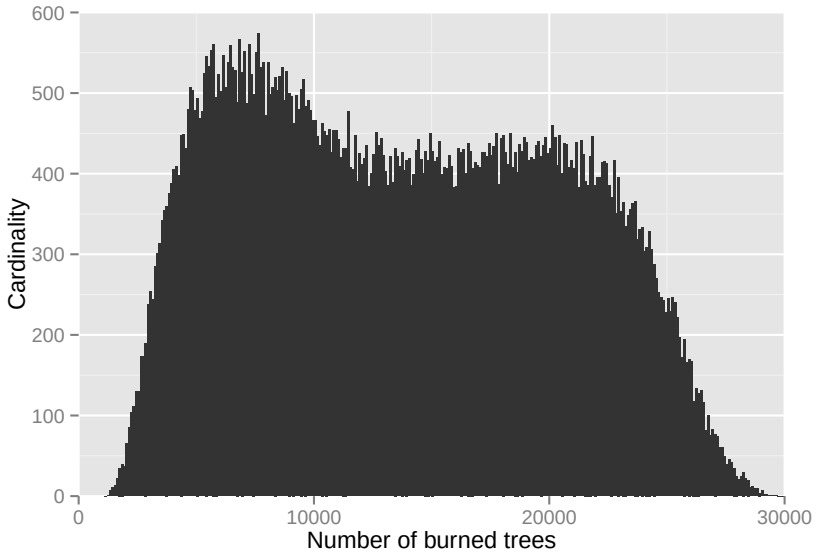

Figure 6: Distribution of the number of burned trees for a density of 59% of trees.

The workflow produces 100,000 independent samplings of the random variable burned trees. Figure 6 shows the distribution of those sampling. The law of the burned trees random variable looks multi-modal.

2.3. Compute statistics on the output

In the previous section we have shown how to compute the distribution of the variable. This one explains how to automate the computation of the median of this distribution.

In order to compute global indicators over all the executions of a model, the results of the model executions should first be gathered and stored in an array. While an exploration transition creates many execution streams for massively parallel executions, its dual, called "aggregation transition", gathers the results among many executions streams generated by an "exploration transition". For instance, in the workflow shown on Figure 7, task *T2* is executed once for each value generated by *T1*. After each execution of *T2*, an execution of *T3* is scheduled. The transition between *T3* and *T4* is an aggregation transition. This means that the transition is fired only when all executions of *T3* have been completed. If a variable named a of type T is produced by *T3*, *T4* should expect a variable a of type array of T containing all the values of a produced by the executions of *T3*. By default an aggregation transition is triggered when the execution streams initiated by the corresponding exploration transition are all terminated. However a stopping condition can also be triggered when a specified

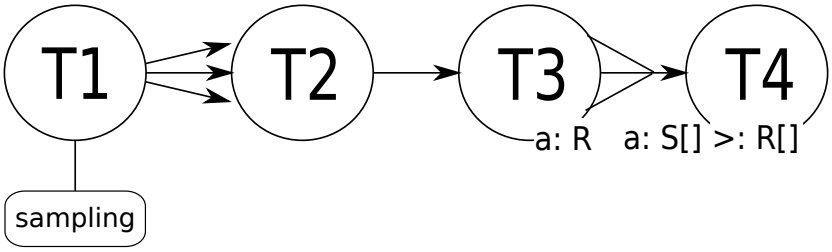

Figure 7: Aggregation transition.

state is reached. If this condition is satisfied, then all the execution streams for the exploration are killed and the aggregation transition is fired.

Listing 3: Compute the median of the distribution of the output variable burned trees

```
/* ... Description of the fire tasks ... */

// Describe the replication task
import org.openmole.plugin.domain.distribution._
import org.openmole.plugin.domain.modifier._

val replication =
  ExplorationTask(
    "replication",
    Factor(seed, UniformIntDistribution() take 100)
  )

// Declare variable to store the median
val burnedMed = Prototype[Double]("burnedMed")

// Describe the median task
import org.openmole.plugin.task.stat._

val median = MedianTask("median")
median addSequence (burnedTrees.toArray, burnedMed)

// Describe the display hook
import org.openmole.plugin.hook.display._

val display = ToStringHook()

// Describe the workflow
val wf =
  replication -< fire >- (median hook display)

val execution = wf toExecution
execution start
```

The code of Listing 3 uses the aggregation transition to compute the median among 100 independent replications of the model. First an exploration task is created to generate 100 different values for the seed. Each seed value is used

to compute an independent realisation of the output variable. All those values are then aggregated in an array by the aggregation transition (noted $> -$). Then the "median" task computes the median of this array. This result is then displayed by the hook: "{burnedMed=13696.5}".

2.4. Exploration of the model input

Understanding complex-system models generally requires an exploration of a wide space of parameters and to gather samples according to various design of experiments (for a review on design of experiments see [20]). Concerning the Fire model it might be interesting to explore the values taken by the median of the distribution of the burned trees depending on the density of trees. Listing 4 shows how to achieve it with OpenMOLE.

Listing 4: Explore the variation of the distribution depending on the density of trees

```
1  /* .. Definition of exploration, replication and median tasks ..
       */
2
3  import org.openmole.plugin.domain.collection._
4
5  val exploration =
6    ExplorationTask(
7      "exploration",
8      Factor(density, 0.0 to 100.0 by 1.0 toDomain)
9    )
10
11 // Describe the hook to write into a file
12 import org.openmole.plugin.hook.file._
13
14 val write =
15   AppendToCSVFileHook(
16     "distribution.txt",
17     density,
18     burnedMed
19   )
20
21 // Describe the execution environment
22 import org.openmole.plugin.environment.pbs._
23
24 val cluster = PBSEnvironment("rreuillo", "avakas.mcia.univ-
       bordeaux.fr")
25
26 import org.openmole.plugin.grouping.batch._
27
28 // Decribe the workflow
29 val replicationCaps = StrainerCapsule(replication)
30 val medianSlot = Slot(StrainerCapsule(median))
31
32 val wf =
33   exploration -< replicationCaps -< (fire on cluster by 100) >- (
         medianSlot hook write)
34
35 val densityTransmission = replicationCaps -- medianSlot
```

```
36
37  val execution = (wf + densityTransmission) toExecution
38  execution start
```

In this script an exploration task is instantiated. It explores the density from 0 to 100 by step of 1. For each value density value, 100 replications of the model are executed to compute the median. Thus the workflow execution launches in total 10,000 executions of the model. This computational load is distributed on a computing cluster (described lines from 21 to 24). In addition to the authentication, the only information required by OpenMOLE to take advantage of cluster is the user login and the address of the master node of the cluster.

In this workflow, the output file should contain the median number of burned trees given a density of trees. Therefore the density value should circulate all along the workflow to be written in the result file. In this script we use a functionality called strainer capsule to implicitly transmit this value from one task to another. Actually, in OpenMOLE workflows, tasks are not directly linked to each-other by transitions, they are first encapsulated in capsules. This convenience makes it possible to use a single task at several locations in the same workflow by encapsulating it in several capsules. In the previous examples the capsules were implicitly instantiated when the workflow was described. This workflow is a bit more complex, therefore explicit capsule instantiation is required. Line 29 encapsulates the "replication" task in a capsule. This capsule is a strainer capsule, meaning that the output of the previous capsules in the workflow are automatically transmitted through this capsule and to the next capsules. In the workflow, the capsule previous to the "replication" task (the one containing the exploration task) produces 1 output: the density. Thus, the density is transmitted through the strainer capsule to the model and to the "median" task.

Another new concept, the "Slot", is used in this listing. In order to describe iterative refinement of the space of parameters (see section 2.5), the OpenMOLE formalism provides seamless iterative transitions allowing cyclic workflows. To distinguish synchronization points (convergent transition pattern) from iterative transitions, the concept of "input slots" has been introduced: a task may possess multiple input slots and the execution of a task is triggered when all the incoming transitions belonging to the same input slot have been passed through. For instance in Figure 8, task *T1* owns two input slots. When transition *tr1* is fired, task *T1* is executed. Then when transition *tr2* is fired, task *T2* is executed. At the end of *T2*, if the *condition* is true, both transitions *tr3* and *tr4* are fired and two executions streams are generated to execute *T1* and *T3* concurrently.

As previously stated, the density should be transmitted to the median task in order to associate the density with the median in the result file. To transmit the density to the end of the workflow, a transition is used (line 35). This transition reaches the same slot as the aggregation transition in a convergent pattern, meaning that the array of values of burned trees (aggregated output results among the executions of the model) and the density (output of the replication

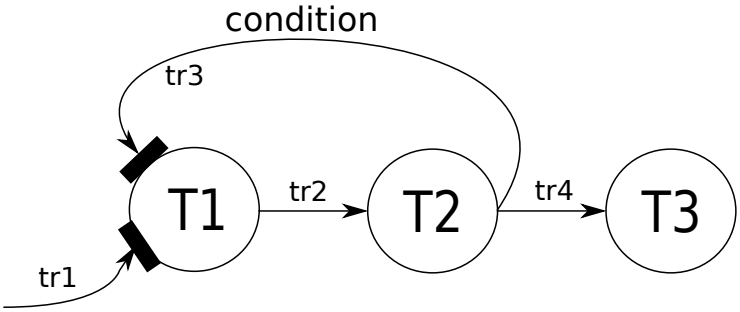

Figure 8: Iterative transition.

capsule) are both made available for the median task. The median task is placed in a strainer capsule, therefore the density is implicitly transmitted through this capsule and available to the hook.

The execution of the workflow produces a csv file, from which an extract is shown in Listing 5. A graphical representation of those results produces the curve on Figure 9. This curve shows the non linearity of the model. A fast transition occurs between densities of 55 and 60.

Listing 5: Result file of the exploration

```
1 density,burnedMed
2 70.0,43127.0
3 17.0,52.0
4 3.0,8.0
5 7.0,19.0
6 34.0,173.5
7 47.0,531.0
8 ...
```

2.5. Solving an inverse problem

The Fire model is a percolation model. Percolation models generally expose a threshold separating percolating dynamics from non-percolating dynamics. This section exposes, how to use OpenMOLE to estimate this threshold.

An inverse problem is a question posed on the outputs of a model that aims at understanding which values of the inputs lead to the production of some dynamics or some patterns. To illustrate how to solve that kind of problem with OpenMOLE, this section exposes how to frame the threshold of the density producing the brutal transition from a dynamics that don't percolate trough the entire space from those that do. In order to estimate the threshold, the workflow shown in Listing 7 (in the appendix section of this article) iteratively refines the input parameter space.

This workflow uses a slightly modified version of the Fire model. In addition to the number of burned trees, this version computes an additional boolean

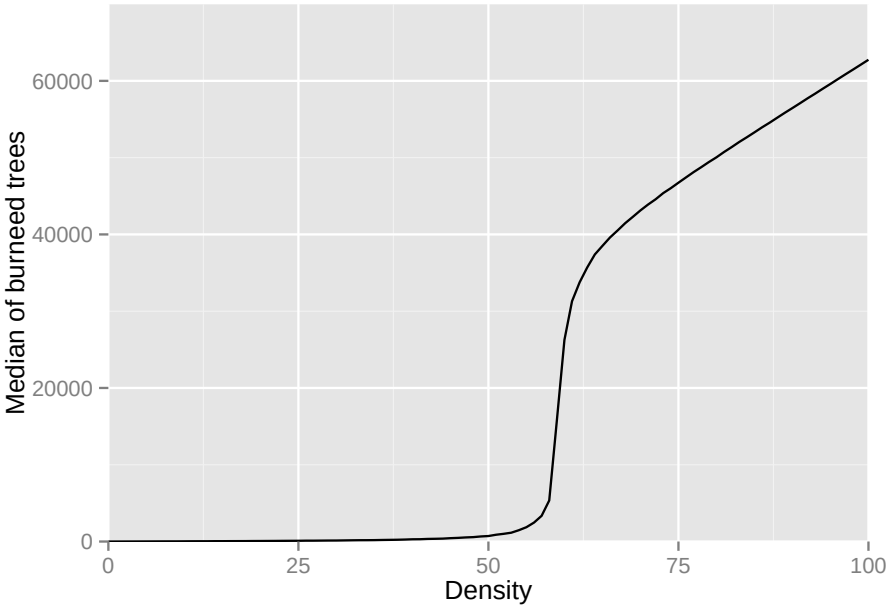

Figure 9: Effect of the density input on the median of the distribution of burned trees.

which is true if the fire has percolated through the entire forest (from left to right). The lines added to the original model are shown in Listing 6 in the appendix section.

To frame the threshold, the workflow first generates 100 density samples from 0 to 100. For each of those sample the model is replicated 1000 times. For a given density, we consider that the model percolates if at least 50% of the replications have percolated. Once the percolation criterion computed for the 100 densities, new bounds are computed for the next exploration cycle (given the densities that led to percolation and the ones that didn't). The condition set on the transition, line 101, makes it possible to repeat the exploration while the distance of the 2 bounds is greater than 0.1.

When executed, this worfklow displays the results: {down=59.0, up=60.0}, {down=59.23, up=59.28}. It computes bounds on the threshold by refining iteratively the search space. At the end, the last bounds should have been distant from 0.01, but they aren't. It means that the model is not precise enough to compute the threshold accurately. To gain in accuracy the simulation space should be enlarged.

3. The automatic calibration of a model of system of cities

OpenMOLE is mature and has already helped modellers in the production of significant scientific results in various application fields: food processing [21, 2], biology [22], bayesian networks [23], environmental sciences [24], geography [3]... This section describes a real case study: the automated calibration of a model in the domain of quantitative geography.

3.1. Geographical background

Geographical simulation models of systems of cities are based on the assumption that the micro-geographic interactions are likely to support the emergence of "stylized dynamics" on macro-geographical scales. These dynamics constitute one of the recurring characteristics of these complex systems [25].

The description of individual-centred dynamics, the non-linearity of the interactions and the importance of the historical context lead geographers to use agent based models (ABM) as support of reflection and experimentation [26, 27, 28]. For example, the SimPop models family [29] has been designed to study the dynamics of exchange between cities. In this family a simplified model, called SimPopLocal describes the emergence of a system of cities where an endogenous process of innovation and exchange generates growth.

3.2. The calibration process

In geography a successful modelling process aims at creating a set of mechanisms which can be validated against empirical data or theoretical knowledge with a high confidence interval [30]. This phase of calibration is generally conducted manually following a trial and error strategy, by introducing values for the parameters and verifying that the outputs of the model correspond to an

expected result. Such a manual calibration is very difficult and tiresome: multi-agent models comprise non-linear interactions, they contain parameters that don't have empirical equivalence and each increment in the process of the calibration can produce completely unexpected dynamics. The limitation of this manual process resulted in performing only a hundred simulations during the validation of a consolidated version of the model "SimPop2" [31]. Using this method one cannot reach a reasonable stage where the model could be considered as validated. Consequently, we decided to create an automated calibration procedure for SimPopLocal.

In order to calibrate SimPopLocal, we have considered the calibration as an objective function that should be reached. The goal was to find a set of parameters that minimizes the gap between the dynamic of the model and a dynamic that matches the geography theory. The reduction of this gap became a guide to feed one optimization algorithm. By doing so, we have answered the question: do there exist combinations of parameters that produce realistic dynamics for the growth of the system of cities?

3.3. The workflow

The calibration of the SimPopLocal model was automated by using a genetic algorithm [32]. These algorithms scan the search space following strategies inspired by natural processes to solve optimisation problems. Since we had multiple objectives we uses a popular multi-objective genetic algorithm called NSGA-II [33].

In its most effective implementation one execution of SimPopLocal lasts on average 3 seconds on a cutting edge processor. This model is stochastic, thus the evaluation of the adequacy of a single set of parameters requires the executions of 100 independent replications (300s) in order to compute the statistical indicators. To converge, a genetic algorithm evaluates the fitness function numerous times. Thus a 300 seconds fitness function is very computationally difficult. That's why we distributed NSGA-II using OpenMOLE.

Figure 10 represents the workflow we designed to distribute NSGA-II. First it generates genomes that are candidate solutions. Those candidate solutions are evaluated in a parallel manner. Finally, new candidate solutions are generated based on the best fit solutions already produced. The entire process is iterated until a convergence is reached (the algorithm doesn't improve the set of solution during a significant number of iterations).

The automatic calibration required almost 400 million executions of the model. It represents nearly 23 consecutive years of computing on single processing unit. These impressive figures illustrate well the difficulty of the calibration task in the absence of a priori knowledge of the behaviour of the model. OpenMOLE made it possible to apply, for the first time, an automatic calibration procedure on a multi-agent model intended to simulate the emergence of a system of urban settlements. It allowed a decisive advance in the validation of this model on two very important points: on the one hand, we proved that the SimPopLocal model is able to generate a satisfactorily evolution of an urban hierarchy; on the second hand, we estimated with a high degree of accuracy the

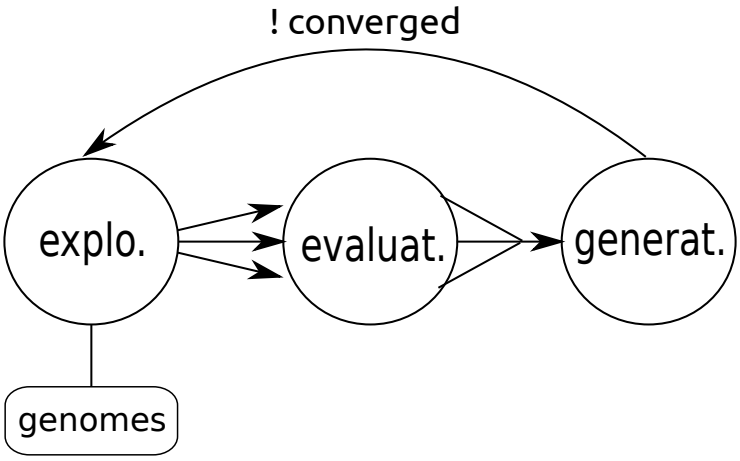

Figure 10: Genetic algorithm workflow.

values for parameters that are essential to this emergence, whereas these parameters remain generally unknown to historians and geographers specialists.

Conclusion

This paper shows the DSL proposed by OpenMOLE to perform large scale model exploration. This DSL is both a model independent way of describing reusable model experiment processes and a simple and efficient way of using high performance environments. Based on the cloud principle, the intensive computational work-load of the numerical experiment is transparently delegated to remote high performance computing environments. This paper demonstrates the adequacy of OpenMOLE for its purpose through a detailed study of a toy model in the complex-system field and by exhibiting a large scale experiment on a real use case for which OpenMOLE produced high-quality scientific results that wouldn't have been computable otherwise.

Acknowledgements

Acknowledgement for the funding received from the European Community's ERC project GeoDivercity. Results obtained in this paper were computed on the biomed and the vo.complex-system.eu virtual organization of the European Grid Infrastructure (http://www.egi.eu). We thank the European Grid Infrastructure and its supporting National Grid Initiatives (France-Grilles in particular) for providing the technical support and infrastructure.

References

[1] P. Bourgine, P. Chavalarias, E. Perrier, F. Amblard, F. Arlabosse, et al., French roadmap for complex systems 2008-2009 (2009).

[2] M. Sicard, N. Perrot, S. Mesmoudi, S. Martin, R. Reuillon, I. Alvarez, Development of a viability approach for reverse engineering in complex food processes: Application to a camembert cheese ripening process. adaptation of the viability theory, Journal of Food Engeneering (EFG) (2011) submitted.

[3] R. Reuillon, S. Rey, C. Schmitt, M. Leclaire, D. Pumain, Algorithmes évolutionnaires sur grille de calcul pour le calibrage de modéles géographiques, in: Conférence France-Grilles, 2012, pp. 12–16.

[4] R. Fisher, The design of experiments. (1971).

[5] J. Kleijnen, Design of experiments: overview, in: Simulation Conference, 2008. WSC 2008. Winter, IEEE, 2008, pp. 479–488.

[6] H. Madsen, Automatic calibration of a conceptual rainfall–runoff model using multiple objectives, Journal of hydrology 235 (2000) 276–288.

[7] D. R. Harp, V. V. Vesselinov, An agent-based approach to global uncertainty and sensitivity analysis, Computers and Geosciences 40 (2012) 19 – 27.

[8] J. Mouret, J. Clune, Uncovering phenotype-fitness maps using mole, connections 5 (2012) 10.

[9] J.-P. Aubin, Viability theory, Birkhauser Boston Inc., Cambridge, MA, USA, 1991.

[10] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-science: An overview of workflow system features and capabilities, Future Generation Computer Systems 25 (2009) 528 – 540.

[11] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the kepler system: Research articles, Concurr. Comput. : Pract. Exper. 18 (2006) 1039–1065.

[12] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, C. Goble, Taverna, reloaded, in: Proceedings of the 22nd international conference on Scientific and statistical database management, SSDBM'10, Springer-Verlag, Berlin, Heidelberg, 2010, p. 471–481.

[13] I. Taylor, M. Shields, I. Wang, A. Harrison, Visual Grid Workflow in Triana, Journal of Grid Computing 3 (2005) 153–169.

[14] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. hui Su, K. Vahi, M. Livny, Pegasus: Mapping scientific workflows onto the grid, 2004.

[15] Z. Farkas, P. Kacsuk, P-grade portal: A generic workflow system to support user communities, Future Gener. Comput. Syst. 27 (2011) 454–465.

[16] R. Reuillon, F. Chuffart, M. Leclaire, T. Faure, N. Dumoulin, D. Hill, Declarative task delegation in OpenMOLE, in: HPCS, 2010, pp. 55–62.

[17] R. Reuillon, M. K. Traore, J. Passerat-Palmbach, D. R. Hill, Parallel stochastic simulations with rigorous distribution of pseudo-random numbers with distme: Application to life science simulations, Concurr. Comput. : Pract. Exper. 24 (2012) 723–738.

[18] T. Glatard, J. Montagnat, D. Lingrand, X. Pennec, Flexible and efficient workflow deployment of Data-Intensive applications on grids with MOTEUR, Int. J. High Perform. Comput. Appl. 22 (2008) 347–360.

[19] S. Jha, A. Merzky, G. Fox, Using clouds to provide grids with higher levels of abstraction and explicit support for usage modes, Concurr. Comput. : Pract. Exper. 21 (2009) 1087–1108.

[20] J. P. C. Kleijnen, Design of experiments: overview, in: Proceedings of the 40th Conference on Winter Simulation, WSC '08, Winter Simulation Conference, 2008, p. 479–488.

[21] S. Mesmoudi, N. Perrot, R. Reuillon, P. Bourgine, E. Lutton, Optimal viable path search for a cheese ripening process using a multi-objective ea, in: ICEC 2010, International Conference on Evolutionary Computation, 2010. 24-26 oct, Valencia, Spain.

[22] I. Junier, R. K. Dale, C. Hou, F. Kepes, A. Dean, CTCF-mediated transcriptional regulation through cell type-specific chromosome organization in the beta-globin locus, Nucleic Acids Research 40 (2012) 7718–7727.

[23] A. Tonda, E. Lutton, R. Reuillon, G. Squillero, P. H. Wuillemin, Bayesian network structure learning from limited datasets through graph evolution, Genetic Programming (2012) 254–265.

[24] R. Lardy, A.-I. Graux, B. Bachelet, D. R. Hill, G. Bellocchi, Steady-state soil organic matter approximation model: application to the pasture simulation model, in: International Environmental Modelling and Software Society, 2012, pp. 769–776.

[25] D. Pumain, Une approche de la complexité en géographie, Géocarrefour: Revue de géographie de Lyon 78 (2003) 25–31.

[26] L. Sanders, Objets géographiques et simulation agent, entre thématique et méthodologie, Revue Internationale de Géomatique 17 (2007) 135–160.

[27] M. Batty, Fifty years of urban modelling : Macro-statics to macro-dynamics, in: The Dynamics of Complex Urban Systems, Springer, 2008, pp. 1–21.

[28] A. Heppenstall, A. Evans, M. Birkin, Genetic algorithm optimisation of an agent-based model for simulating a retail market, Environment and Planning B: Planning and Design 34 (2007) 1051–1070.

[29] D. Pumain, Une thérie géographique des villes, BSGLg 55 (2011) 5 – 15.

[30] R. Sargent, Verification and validation of simulation models, Winter Simulation Conference, Orlando, Florida, 2005, pp. 130–143.

[31] A. Bretagnolle, E. Daudet, D. Pumain, From theory to modelling : urban systems as complex systems, Cybergeo (2006) 1–17.

[32] J. Holland, Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence, MIT press, 1992.

[33] K. Deb, S. Agrawal, A. Pratap, T. Meyarivan, A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii, Lecture notes in computer science 1917 (2000) 849–858.

4. Appendix

Listing 6: Addition to Fire.nlogo code for percolation detection

```
globals [
  ;; same as Fire.nlogo
  percolate       ;; fire touch the right border
]

to setup
  ;; same as Fire.nlogo
  set percolate false
end

to ignite
  ;; same as Fire.nlogo

  if pxcor = max-pxcor [set percolate true]
end
```

Listing 7: Approximate the percolation threshold by iterative refining

```
// Declaration of the variables
val seed = Prototype[Int]("seed")
val density = Prototype[Double]("density")
val percolate = Prototype[Boolean]("percolate")
val down = Prototype[Double]("down")
val up = Prototype[Double]("up")

// Describe the model task
import org.openmole.plugin.task.netlogo5._

val fire =
  NetLogo5Task(
    "Fire",
```

```
14      "FirePercolation.nlogo",
15      List("random-seed ${seed}", "setup", "while [any? turtles] [
            go]")
16    )
17
18  // Describe input and output of the task
19  fire addInput seed
20  fire addNetLogoInput (density, "density")
21  fire addNetLogoOutput ("percolate", percolate)
22
23  //Describe the exploration task
24  import org.openmole.plugin.domain.distribution._
25  import org.openmole.plugin.domain.modifier._
26
27  val replication =
28    ExplorationTask(
29      "replication",
30      Factor(seed, new UniformIntDistribution take 1000)
31    )
32
33  import org.openmole.plugin.task.groovy._
34
35  val count =
36    GroovyTask(
37      "count",
38      "percolate = percolate.count(true) > 500"
39    )
40
41  count addInput percolate.toArray
42  count addOutput percolate
43
44  val newBounds =
45    GroovyTask(
46      "newBounds",
47      "results = [density, percolate].transpose().sort{it[0]}\n" +
48      "down = results.takeWhile{it[1] == false}.last()[0]\n" +
49      "up = results.reverse().takeWhile{it[1] == true}.last()[0]"
50    )
51
52  newBounds addInput percolate.toArray
53  newBounds addInput density.toArray
54  newBounds addOutput down
55  newBounds addOutput up
56
57  import org.openmole.plugin.domain.range._
58
59  val exploration =
60    ExplorationTask(
61      "exploration",
62      Factor(density, Range[Double]("${down}","${up}", "${(up -
            down) / 100}"))
63    )
64
65  exploration addInput down
66  exploration addInput up
67  exploration addOutput down
68  exploration addOutput up
```

```
69
70 exploration addParameter (down, 0.0)
71 exploration addParameter (up, 100.0)
72
73 // Describe the hook to write into a file
74 import org.openmole.plugin.hook.display._
75
76 val display = ToStringHook()
77
78 // Describe the execution environment
79 import org.openmole.plugin.environment.glite._
80
81 val complexSystemsVO = GliteEnvironment("vo.complex-systems.eu")
82
83 // Decribe the workflow
84 import org.openmole.plugin.grouping.batch._
85
86 val explorationCaps = Capsule(exploration)
87 val replicationCaps = StrainerCapsule(replication)
88 val countSlot = Slot(StrainerCapsule(count))
89 val newBoundsCaps = Capsule(newBounds)
90
91 val wf =
92   Slot(explorationCaps) -< replicationCaps -< (fire on
       complexSystemsVO by 500) >- countSlot >- (newBoundsCaps
       hook display)
93
94 val densityTransmission = replicationCaps -- countSlot
95
96 val iter = newBoundsCaps -- (Slot(explorationCaps), "(up - down)
    > 0.1")
97
98 val execution = (wf + densityTransmission + iter) toExecution
99 execution start
```