Assasim – Final Report

Grégoire Beaudoire, Titouan Carette, Maverick Chardet, Maxime Faron, Jean-Yves Franceschi, Rémy Grünblatt, Antonin Lambilliotte, Fabrice Lebeau, Christophe Lucas, Harold Ndangang Yampa, Vincent Michielini, Victor Mollimard, Clément Sartori, Nicolas Vidal.

Supervisor: Silviu-Ioan Filip

`http://assasim.org/`

Ecole Normale Supérieure de Lyon

May 13, 2016

Abstract

This report presents the ASSASIM project, developed in the framework of the Integrated Project of the M1 of the Ecole Normale Supérieure de Lyon and supervised by Silviu-Ioan Filip. It explains our reflexions and our work, which we carried out from October 2015 to April 2016. For more practical details like installation or tutorials, please visit our webpage: `http://assasim.org/`.

Contents

1 Introduction

Agent-Based Modeling (ABM) is a class of modeling which is structured around autonomous entities (the *agents*) *e.g.* economical entities, citizens. Each of these agents has a defined behaviour which depends on their characteristics and the interactions between them.

There are a lot of existing ABM software products, but most of them are either abandoned, deprecated, inconvenient to use or specific to some applications (*e.g.* biology, economy). However, ABM can be used in many fields of science, and by users who may have little programming experience. With our project ASSASIM (AcceSSible Agent-based SIMulator), we aim at creating a software application which allows to define and simulate agent-based models, combining both expressiveness and accessibility.

The ASSASIM team is composed of 14 students from the École Normale Supérieure de Lyon : Grégoire Beaudoire, Titouan Carette, Maverick Chardet, Maxime Faron, Jean-Yves Franceschi, Rémy Grünblatt, Antonin Lambilliotte, Fabrice Lebeau, Christophe Lucas, Harold Ndangang Yampa, Vincent Michielini, Victor Mollimard, Clément Sartori and Nicolas Vidal.

1.1 Existing software

Our review of the existing ABM softwares was based on a study called “Agent-based Simulation Platforms: Review and Development Recommendations” (by Steven F. Railsback, Steven L. Lytinen and Stephen K. Jackson)[1]. The two softwares which were the closest to what we have achieved with ASSASIM were NetLogo[2] and AnyLogic[3]. The former is an open-source software, very accessible with a GUI, but it is very focused on simulations taking place in a 1D or 2D space, often a grid. With ASSASIM we wanted to be more general than that, to be able to address more problems. The latter is a paid software with closed source code. It is very general and has libraries to be able to use predefined elements and build a model in a short amount of time, and it allows 3D visualisation in real time when it makes sense. However, it seems to be complex to use, and few people can afford to buy the software and the training material.

Other ABM solutions mentioned are frameworks and libraries to allow one to code an ABM simulation, often in `Java` or in `Objective C`. While these are more general by nature, it might be too complex for a scientist who has very little knowledge of programming. With ASSASIM we wanted to create something that anyone could learn to use in a short amount of time.

Therefore we decided to make of ASSASIM a *general* and *accessible* simulator, so that it could meet the advantages of these already existing software products.

1.2 Paradigms

In order to be as general and accessible as possible, we decided to develop our tool based on the following model structure.

Evolution of the simulation The simulation is *discrete*: at each time step, the agents perform some actions depending on their previous state and the other agents.

Agents A model defines several types of agents, each one being characterised by its *attributes* and its *behaviour*. The set of attributes of an agent represents its current state, while its behaviour describes which actions the agent will perform during a time step. An agent can – through its behaviour – modify its own characteristics, and is the only one able to do so. Therefore a type of agent is like a *class* in oriented object programming, containing attributes and whose methods define its behaviour; thus we can introduce the notion of inheritance of agent types – an agent type B inheriting A has all the attributes of A and may use A's behaviour.

[1]At the time of the publication of this report, accessible at: `http://www.sci.brooklyn.cuny.edu/~sklar/teaching/s10/alife/papers/railsback-sim06.pdf`

[2]`https://ccl.northwestern.edu/netlogo/`

[3]`http://www.anylogic.com/`

Attributes An attribute can be an integer, a double, a string or a user-defined type. There are two types of attributes:

- private attributes: these attributes are only visible by the agent it belongs to;
- public attributes: these attributes can be accessed instantly by any other agent.

However all attributes are only modifiable by the agents they belong to, and no other, in order to avoid internal arbitrary decisions for conflict resolution (see remark at the end of this subsection).

Behaviours A behaviour expresses what will be the actions of an agent during a time step. It can, beyond elementary operations:

- modify the attributes of the agents it belongs to;
- access public attributes of the other agents;
- receive and send *interactions* from and to other agents.

Interactions An *interaction* is the only way for agents to communicate. There may be several types of interactions, each one carrying information: its type and some other attributes. For instance, an interaction $AskMoney(100)$ will carry the money request and the value of the request. Other possible uses of the interactions deal with the limitations of the attributes requests and modifications explained previously: it is possible *via* an interaction to simulate an order for an agent to modify an attribute of another agent, or to get the value of a private attribute.

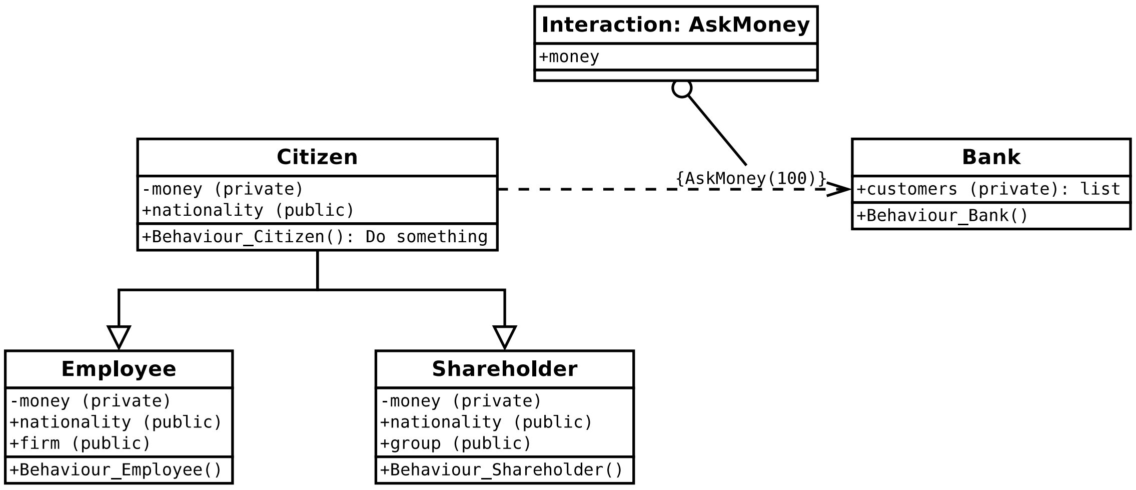


Figure 1: Illustration of the paradigms on a simple example

Instantiation and simulation Once the model is defined, the user has to provide an instantiation which corresponds to the initial state of the simulation: a simulation is the combination of a model and an instantiation.

We believe that these paradigms are a good basis to build an accessible software product: the object-oriented programming focus corresponds to intuitive notions (*e.g.* inheritance which is associated to the notion of categories) and is reduced to a minimum to be as simple to understand as possible. Moreover, the simplicity associated to the notions of behaviour and interaction allows to define any agent-based model, since the behaviours ensure the agents to do anything on themselves, and interactions can carry any kind of information, in order for the agents to react to the interactions they receive: these paradigms capture the essence of agent-based modelling.

Remark The choice of the self-management of the agents – only an agent can modify its own attributes – was a central subject and was strongly considered when we discussed about these paradigms. It is true that by assigning priorities to agents we could have introduced a concept of *orders*, which would have allowed some agents to directly modify others. However we decided to exclude this possibility for several reasons. The first of them is a practical one: we wanted to build a parallelised simulator, therefore an ordering mechanism would have been much more difficult to implement and would have probably introduced scheduling problems – which order should be executed on a particular machine? –, whereas this set of paradigms allows for natural parallelisation. Another more fundamental reason is the problem of conflicts between orders: if two agents send to another one contradictory orders with the same priority, then how should our simulator behave? Any solution to this problem would be arbitrary, either based on internal meaningless values or some specifically designed attributes, or based on randomness making our simulator fundamentally non-deterministic. None of these solutions are satisfying, whereas we believe the solution we chose is ideal: the concept of orders is easily simulated by the interactions, and the user has to define herself or himself how an agent should behave when it receives conflicting orders, by possibly using randomness if so desired. Using this method therefore makes our simulator clearer and more practical for the user since it is her or him who has to predict the conflicts and how the constructed model should behave in this case.

2 General objectives

After fixing the paradigms of our agent-based modelling base, we had to think about which features our tool should provide besides the simulation in order to be as accessible as possible. We chose to focus on the steps which are the most important for the user who would want to use an agent-based simulator: how he can define a model, and how he could analyse the results of the simulation. Thus we fixed the following main objectives for Assasim:

- Create a simple and expressive language allowing to describe an agent-based model following the paradigms we defined.
- Allow the advanced user to create models directly in a convenient `C++` syntax.
- Design a simulation library which can simulate any model with an acceptable efficiency – without having to be as efficient as the best existing software product, our simulation tool should be efficient enough to be usable even for complex and large models.
- Develop an ergonomic graphical user interface (GUI) giving to the user a practical way to build models, run a simulation and above all visualise and analyse the results of a simulation.

The final practical goal was to produce such a simulator capable of handling simple models like cellular automata, networks or basic economic models. If we had enough time, we planned some optional objectives which would have improved the quality of our tool:

- Develop and test several complex models.
- Implement the possibility for the user to dynamically change some simulation parameters.
- Create a graphical model creation tool allowing the user to define models using diagrams and predefined "black boxes" to make our simulator even more accessible.

Also for the sake of accessibility, we tried to make our program cross-platform, in particular we chose the libraries and tools used for the development (`Libtooling`, `boost`, `Qt`...) in order to enable a platform-independent implementation. However, we did not have time to do a proper support for Windows and MacOS.

3 Organisation

3.1 Structure of the program

Figure 2 shows all the interactions and the organisation between the components of Assasim. All interactions symbolised by arrows will be clarified in the sections detailing the work of each team, except for the transition between the graphical modelling tool and the language since this graphical tool was not developed due to lack of time.

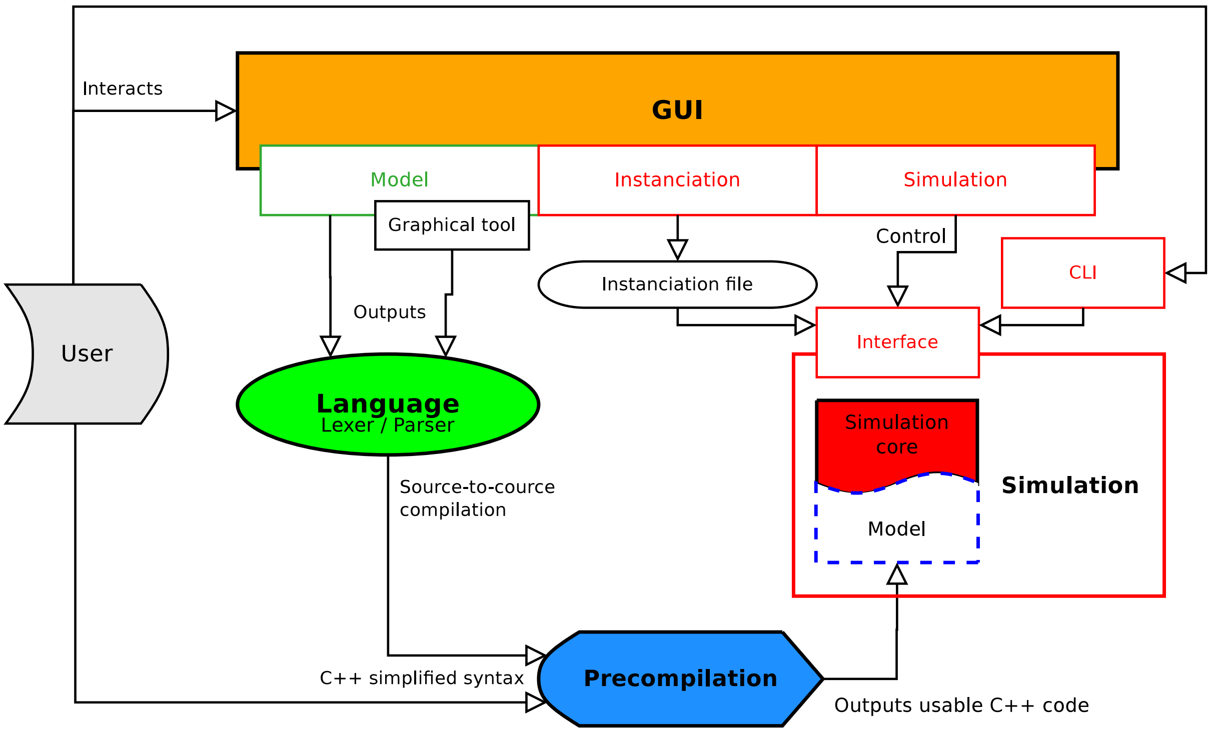


Figure 2: Architecture of AssaSim

3.2 Teams

The work on the project was split between three teams, each implementing one of the facets of the project:

- The *Simulation* team which worked on the simulation core, the preprocessing of a model and a CLI (Command Line Interface).
- The *Language* team which worked on creating a high level intuitive language to define agent-based models and implementing it with a lexer/parser.
- The *Interface* team which worked on designing an easy-to-use graphical interface that allows an ergonomic call (without command line) to the simulation core and the visualisation of its results.

Table 1 shows the repartition of the AssaSim team into the three sub-teams of our project.

3.3 General technical choices

3.3.1 Choice of the language

Most current agent-based simulators are coded in `Java` since it offers a large set of useful libraries for simulation and parallelisation. However we decided to code our project in `C++` with the latest standard – `C++14` – for several reasons.

The first one is practical: some of our teammates know and practice `C++` very well, whereas almost nobody in our team has good knowledge of `Java`; as we knew that this project required an important workload, we decided not to spend a huge amount of time on learning how to use `Java` along with the needed libraries. Also, `C++` is a lot more flexible than `Java`, allowing us to write both high and low level

Team *Interface*	Team *Language*	Team *Simulation*
Christophe Lucas (*leader*)	Titouan Carette (*leader*)	Rémy Grünblatt (*leader*)
Vincent Michielini	Antonin Lambilliotte	Jean-Yves Franceschi
Starting from Feb. 2016:	Nicolas Vidal	Fabrice Lebeau
Maxime Faron	Harold Ndangang Yampa	Maverick Chardet
Victor Mollimard		Grégoire Beaudoire
Clément Sartori		*Until Feb. 2016:*
		Maxime Faron
		Victor Mollimard
		Clément Sartori

Website manager: Rémy Grünblatt
Git manager and `C++` consultant: Maverick Chardet
Heads of the project: Jean-Yves Franceschi and Fabrice Lebeau

Table 1: Team repartition

code, and uses object-oriented programming (OOP) (like `Java`), which is a requirement for ASSASIM, since its paradigms are based on OOP.

Moreover, a lot of tools that we wanted to use and we are quite familiar with are specifically designed for `C` or `C++`, and not for `Java`, like most of the `MPI` implementations – for instance, `OpenMPI` or `IntelMPI` – or the GUI creation tool `Qt`. `C++` comes with lots of external free libraries, which helped us, for example, to easily parse `JSON` files; more generally, many `Java` libraries also have a `C++` equivalent.

We also planned to code in `C++` because we know that it would ensure ASSASIM achieves better overall performance than if written in `Java`, especially in the simulation part, where very low level programming is used in order to make use of some interesting features of the last `MPI` standard. Another advantage for the performance of our tool is that `C++` is natively compiled whereas `Java` is translated to bytecode, making it less suitable for applications where performance is extremely critical.

Finally, this choice also allows us to better distinguish ourselves from the other agent-based simulators, giving the user the opportunitiy, for instance, to code his own model in `C++`.

The choice of all the libraries used in this project are justified in the following sections.

3.3.2 Other choices

One of a the most important aspects of this project is accessibility, which can be achieved by opensourcing our tool and allowing other people to study it. Therefore we chose to protect our work using the LGPL v3 license, allow anyone to get, use and modify our code without having to state all the modifications this person made. We believe that is it the only way for ASSASIM to be used and improved after our project is completed.

Regarding the practical development choices, we used the version control system `Git`, hosted privately on the Aliens (Association pour la Libre Informatique à l'ENS de Lyon - a student organisation at the ENS de Lyon) repository[4]; the ready-to-use program is hosted on a public GitHub repository[5]. Finally, we chose the code conventions recommended by Google[6] as long as they are meaningful in our project, and we documented our code using the Doxygen[7] standard.

4 Language

We aimed to design a user-friendly programming language allowing those who are not advanced `C++` programmers to use ASSASIM. The high-level language is a mean to easily specify a simulation without using advanced or awkward `C++` syntax. The user only has to implement the provided structure and the code is then translated into proper `C++` code. Another feature of our language is the possibility to use `C++` code directly in the language to specify more complex behaviours.

[4] `http://git.aliens-lyon.fr/`
[5] `https://github.com/assasim/`
[6] `google.github.io/styleguide/cppguide.html`
[7] `www.doxygen.org/`

4.1 Structure

A simulation is defined by the agents and the messages they might exchange. The language allows to specify these elements (agents and messages) by implementing a general framework.

One must first declare all the messages. A message is specified by a name and a tuple which will be the default parameters of the message. Here is an example of a message section with the definition of one message POSITION which carries a couple of integers:

```
Messages
    integer, integer POSITION : (0,0)
```

Then it is the turn of the agents to be defined. The specification of an agent needs the implementation of three parts. The *States* part contains the declaration of the agent's attributes. All those attributes together specify the state of the agent. An attribute is defined by four items:

- Its visibility. An attribute can be either private, *i.e.* only the owner agent can access or modify it, or public, *i.e.* any agent can access it. If nothing is specified, an attribute is private by default.
- Its type. It can typically be an integer, a float or a boolean.
- Its name.
- Its default value.

The *Filter* part implements the way the agent modifies its own attributes depending on the received messages. The messages received at the last time step of the simulation are stored in a list. The agents can access the different elements of this list by using specific functions which can return the numbers of messages received, the list of the senders, and so on; depending on those parameters, the agents modifies its own attributes.

Finally the *Behavior* part implements how the agent acts in a given state during a time step. The actions an agent can perform are essentially modifying attributes, sending messages and creating other agents; the agent can also kill itself if needed.

It is possible to switch to `C++` with the tokens { and }, the code between them being considered as `C++` code during the translation.

4.2 Example

The following example is a toy prey-predator model:

```
Messages
    EAT : ()
    MIAM : ()

Wolf is Agent

    State
        integer stomach: 10
    Filter
        if number of MIAM > 0 then stomach=stomach+1 end
    Behavior
        if stomach= 0 then KILL
        else send EAT to any sheep
        stomach=stomach−1 end

Sheep is Agent

    State
        boolean dead: false
```

```
Filter
  if number of EAT $> 0$ then dead=true end
Behavior
  if dead then send MIAM to who sent EAT end
```

Here, there are two types of messages, EAT and MIAM; those messages are empty: this means they carry no additional information like some tuple of integers. There are also two kinds of agents, the *Wolf* and the *Sheep*. The *Wolf* has one attribute *stomach* which by default has value 10, thus the state of a wolf is entirely defined by the value of this attribute. The *Sheep* also has only one attribute which is the boolean *dead* which is by default $false$, representing the fact that all the sheeps are alive at the beginning of the simulation.

When a wolf receives at least one MIAM message in a time step then its stomach is filled. If a wolf has its stomach empty, with value 0, it starves so that the corresponding agent kills itself. At each time step, each alive wolf tries to eat a sheep chosen at random among all the *Sheep* agents, thus this interaction is represented by sending the EAT messages to the corresponding agents. The agents which received an EAT message switch their dead attribute to true. Then an agent which has been killed sends to the wolf which ate it the MIAM message before to kill itself.

The global dynamic of this model is very simple: the wolves will eat all the sheep, maybe some unlucky wolves will die during the process because of concurrency with other wolves, and then all the remaining wolves will starve because there are no sheep left to eat. This is not a very interesting model and this implementation is not the simplest way to represent it, but it shows what is the typical structure of a model declaration in our language.

4.3 Translation

To translate our language into proper `C++` code, we use an intermediate step which stores the data of the declaration in a tree – like an AST – which is then transformed into `C++` code.

The parsing uses the `C` tools `Flex` and `Bison` to read and parse the input file, and stores the main information in the data structure. During this step specific functions as those of the *Filter* part for example are replaced by `C++` functions which fit with the simplified syntax of Assasim (see next section). The *Filter* part is thus merged to the *Behavior* part to obtain the final code. Finally the code is compiled as a simulation directly coded in `C++`.

4.4 Conclusion

The aforementioned language is entirely specified and able to define many models easily, however the lexer and parser haven't been completed due to a lack of time, so for the moment it is not possible to actually define simulations using it.

5 Simulation

The simulation part consists of multiple sub-parts: the simulation core, the precompilation and the command line interface.

5.1 Simulation core

The goal of the simulation core is to distribute the calculations used to model agents and their behaviours. To speed up calculation, we defined an architecture centered around the use of many multi-core machines connected to the Internet.

5.1.1 Structure overview

The simulation core uses multiple entities interacting by message passing as well as direct memory access and remote direct memory access, following a Master/Slave paradigm. The slaves take care of the execution of the behaviours of the agents, while the masters serve as intermediates to facilitate inter node communication.

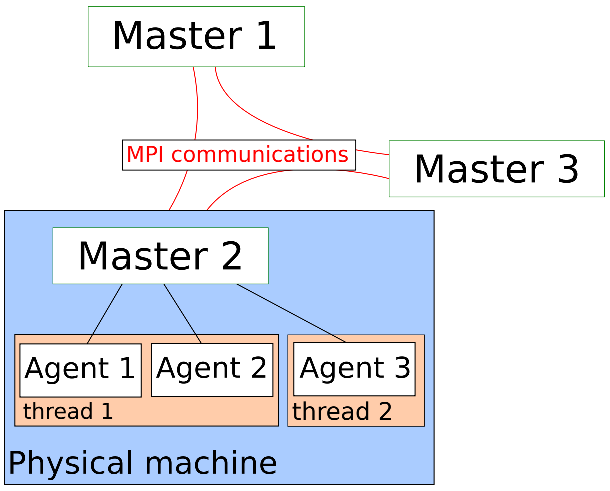


Figure 3: Organisation of the simulation core - General Overview

5.1.2 Simulation step organisation

As in the modelisation phase, the simulation core execution is divided into time steps, each of them being composed of multiple sub-steps:

1. Copy of the public attributes of the agents on the masters (intra node communication)
2. Migration of the agents across the physical machines
3. Exchange of the interactions between the masters (inter node communication)
4. Distribution of the interactions to the agents (intra node communication)
5. Execution of the behaviours of the agents

5.1.3 Different ways of accessing data

One master is present on each physical machine, managing multiple agent handlers (each of them executed in a thread), which in turn execute the behaviours of multiple agents. The inter node communication is done using message passing and the Message Passing Interface (`MPI`), and the intra node communication is done using shared memory.

To improve performances, the masters copy the public attributes of the agents at each simulation time step: this allows remote direct memory access (RDMA) from the other masters. RDMA allows to directly get or put data to a remote process without having him to manage the message receiving: one part (called the window) of the remote process memory acts like a local shared memory, in which you can write or read. It also simplifies the communication protocol: you do not have to know the number of message that will be sent or received beforehand, which means you do not have to send additional messages containing this information.

Moreover, some public attributes that are marked as critical in the language are replicated on all the masters at each time step: those attributes represent data that would be transferred anyway (for example, the position of all the agents), so they are transmitted and replicated preventively to allow easy access from the agents.

Figure 4 illustrates the different kinds of data exchange that are used in Assasim. The public attributes of the agent Bob and agent Alice are copied to their master's memory at each time step, as well as the critical attributes. The latter are also replicated on the other master: if the agent Smith wants to access one critical attribute of Bob or Alice, it directly reads it from its master memory (direct access). If it wants to access to a public attribute of Bob of Alice, it reads the remote memory of the master that manages them (Remote Direct Memory Access). And, if it wants to access a private attribute of agent Bob, it has to send an interaction to him using message passing, as the agent Bob can refuse to send this attributes in its behaviour.

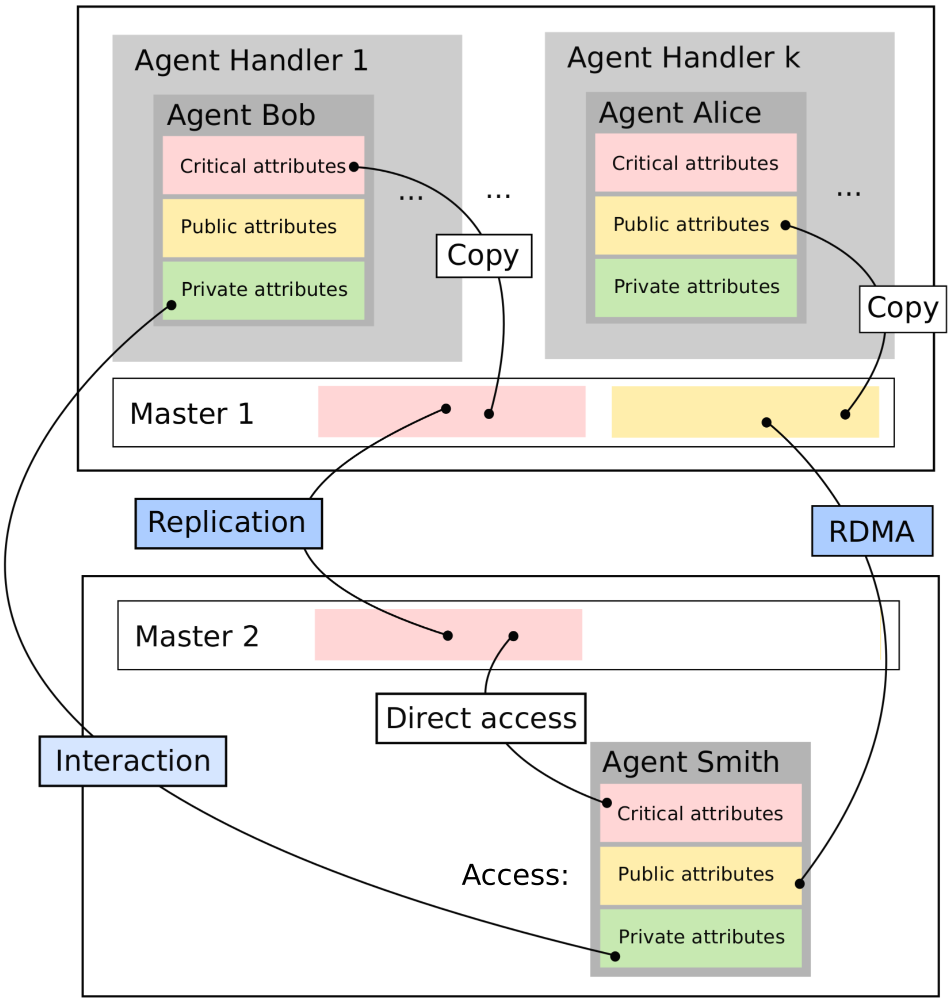


Figure 4: Organisation of the simulation core - Data exchange

5.1.4 Migration mechanism

The architecture also provides a way to migrate agents across the physical machines but no meaningful heuristic has been implemented. To do so, the agents attributes are dumped, then each concerned agent is killed and another agent is created on the destination master.

5.1.5 Choice of `MPI`

We chose to use `MPI` to parallelise our tool and as an interface for sending messages between the parallelised nodes. It has been pointed out by our expert reviewer that we could have used other, more practical tools like `ZeroMQ`; also, there exist some specifically designed languages for node communication like `Erlang`. However, `MPI` still appears as the best choice for us, for the following reasons.

First of all, when this project began, the only parallelisation tool that we practised significantly was `MPI`. So this choice allowed us to not spend any extra time learning other (very) different tools and to solely focus on the creation and development of our own tool.

Moreover, `MPI` offers other advantages in our case. It provides very efficient tools for message sending ((a)synchronous Sends or Receives) and even memory sharing, with RDMA which utility has been proved in the previous explanations: we could not have developed our tool without `MPI`'s characteristics, because attribute requests would have been for too difficult to handle with other tools like `ZeroMQ`.

MPI also combines advantages of both message handling tool like ZeroMQ and languages like Erlang, thus cancels their drawbacks: like Erlang, it is a powerful parallelisation tool which can connect multiple nodes from various origins (for example HPCs), and is even fully compatible with C++ – it is not the case of Erlang, which only provides message handling; and like ZeroMQ, it handles messages very well with even more features. The only drawback of MPI in our case is its error handling, for instance if a node is not accessible anymore, which is not flexible; however if an advanced user gets such problem, then he can also search why this node has failed since on single machines there should usually be no such problems.

Another significant advantage for the choice of MPI is the fact that, unlike ZeroMQ, it is a standard which has lots of various implementations: free ones like OpenMPI or MPICH, or commercial ones like IntelMPI. This way, an advanced user can choose which implementation is the most convenient for him, depending on his processor or his compiler.

5.2 Data export and control

5.2.1 Data format

To allow the simulation to communicate with the outside (user, GUI, ...) we decided to use the JSON[8] format. It is simple, compact, easily readable and allows to define structured information in a less verbose way than XML.

For critical data exchange (real-time data export) and for the information that the user is not supposed to modify, we used the Universal Binary JSON (UBJSON) format[9] because it is faster to write and read, and less space-consuming. An important remark is that an UBJSON structure has an unique JSON structure equivalent, and conversely. It is easy to go from one to another with a simple conversion program.

To handle JSON files, we used two libraries: Jeayeson[10] for usual JSON files, and UbjsonCpp[11] for UBJSON files. Both are easy-to-use C++ libraries making use of the latest C++ standard.

Model The model is exported by the precompilation program into a binary JSON file. It contains the list of all the types of agents and all the information about the properties of these types (name and C++ type). This standard model file allows an external program such as the GUI to know the model independently from the precompilation program.

Instance An instance is a definition of the state of the simulation at time 0. The simulation needs to load an instance before starting, to know how many agents of each type are alive at the beginning, and which values their properties have. Because this is defined by the user, it is a plain JSON file containing collections of agents. A collection of agents is a group of agents of the same type, with all of them having by default the same property values, but which can be overwritten for some of them. This allows to define a large number of agents in a very dense manner, as long as their properties are similar. One feature that could be added is to allow simple expressions on agent-specific values (like the ID or the values of the other properties) for default values of the properties.

Valuation At any step in the simulation, it is possible to export the current values of the attributes of the agents in a format similar to an Instance file. However, it is possible to add some aggregations of values, like the min/max/mean/... for a property among the agents of the same type, etc. It can be exported either to a plain JSON file (to be read by a human or used as a new Instance for restarting the simulation in this setup) or to a UBJSON file (for use by another program, like the GUI for instance).

5.2.2 Simulation control

Now that a time step has been defined and implemented, our simulator needs to provide methods which the user can use to control his own simulations. We present in the following the way the user can communicate orders to the simulation, and some interesting features of the control of the simulation.

[8]http://www.json.org/
[9]http://ubjson.org/
[10]https://github.com/jeaye/jeayeson
[11]https://github.com/WhiZTiM/UbjsonCpp

API The orders that are given to the simulation are received using the Boost.Interprocess library, which allows two separate `C++` programs to communicate. The received orders are stored in a queue, waiting to be treated by the masters. A list of possible orders is given in the documentation of our code.

An advantage of this choice is that the (very) advanced user is allowed to design its own interface to the simulation if he does not want to use our CLI, by simply respecting the defined communication convention with the simulation.

Parameters of the simulation Before initialising a specific simulation, the user can fix some technical parameters. Particularly it is possible to change the number of threads used by each `MPI` cluster, *i.e.* by each master. For the simulation to be as efficient as possible, the quantity number of threads per master $\times$ number of masters should be equal to the number of physical threads available for the program.

Data export The simulation is able to gather all attributes of all existing agents in the simulation and to export them in a file respecting the output format defined earlier. It is done by first grouping all the available information in each master; then these parts of `JSON` files are sent using `MPI` to the first master which merges them and outputs the result.

Dynamical changes of the simulation Between two time steps, it is possible to modify an attribute of any existing agent in the simulation, if it is sendable by `MPI`.

5.3 Precompilation

Even if we decided to develop a high-level and accessible language for defining a model, we wanted to allow more advanced users to directly code in `C++` (the choices made in the beginning, like OOP, and the fact that we decided to code in `C++` make it natural to develop a model in `C++`).

It soon became clear that we would have to add a precompilation step, since some parts of the code, like the behaviours or the instantiation of the agents, would have to depend on the model (i.e. we had to do some code generation). Moreover, even if developing in `C++` is intended for advanced users, there are parts of the code we could not ask the user to write, because it would be too tedious and error-prone. We also chose to add shortcuts for basic functions of the simulation (like handling messages) which makes defining the model easier.

A schema of the precompilation program can be found in Figure 5. The 3 main precompilation steps are:

Common step It parses the `C++` code given in input using `LibTooling`[12] (a `C++` interface for the `Clang` compiler) and retrieves the relevant information about the model. For instance, in `C++` an agent type is defined by writing a class which inherits from class `Agent`. At this point, we also check if the code entered is valid according to the restrictions we put on models (for example, public attributes of an agent should be sendable by `MPI`). In this step, the user should only define the *structure* of the model, i.e. the attributes of agents and messages.

Step 1 It generates the code in order to allow the user to define the `Behavior` function of the agent types with simplified syntax for sending and receiving messages, iterating over existing agents of a class and requesting a public attribute of an agent.

Step 2 It takes as input the complete definition of the model (including the behaviours written in simplified syntax) and outputs the final compilable code of the simulation: some code is generated (like the `MPI_Datatypes` of the attributes or the model instantiation from a `JSON` file) and some code is translated (simplified syntax). Finally the generated code and the code written by the user are merged with the simulation core (i.e. the code which is common to every model) to obtain the final code.

A short example of simplified syntax is given in Figure A.1. You can see, especially for sending messages, that it is much more convenient to code (or generate code) using this syntax.

Another advantage of having this precompilation step is to allow team Language to translate the high level code into some simplified `C++` code following the same standard. Moreover, errors are checked

[12] `http://clang.llvm.org/docs/LibTooling.html`

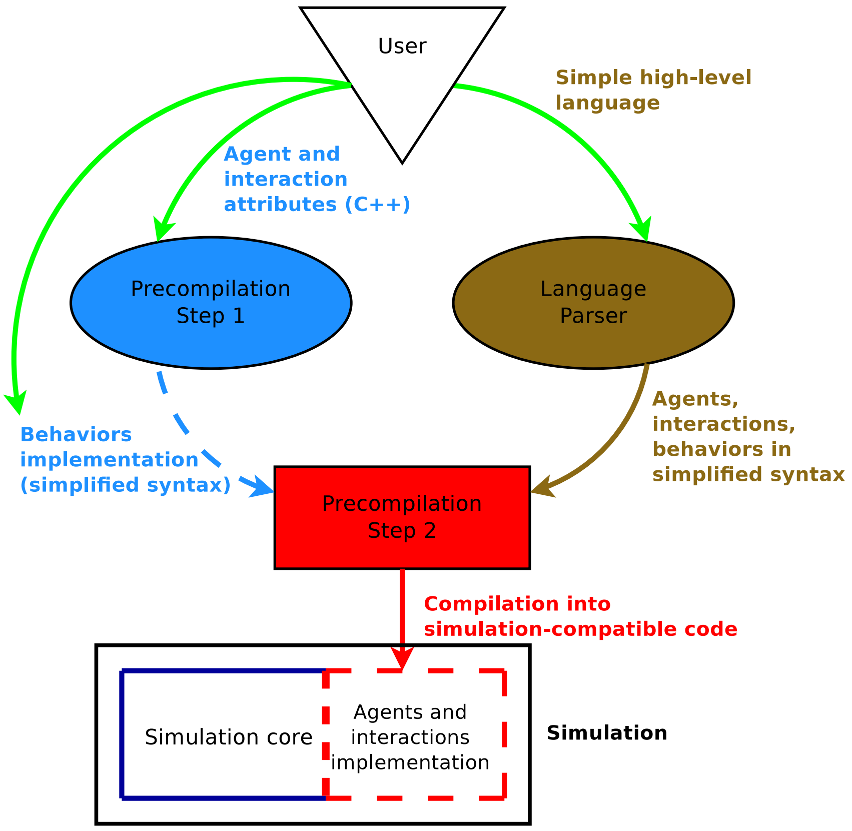

User
Simple high-level language
Agent and interaction attributes (C++)
Precompilation Step 1
Language Parser
Behaviors implementation (simplified syntax)
Agents, interactions, behaviors in simplified syntax
Precompilation Step 2
Compilation into simulation-compatible code
Simulation core
Agents and interactions implementation
Simulation

Figure 5: Schema of the precompilation program

after the translation into `C++` which allows to make the error checking step common between high-level language and `C++` development.

5.4 CLI

The work that was done in precompilation allowed, among other things, the simulation part of ASSASIM to be independent of the high-level language by giving an advanced user the ability to directly create models in `C++`. The purpose of the Command Line Interface we created is to provide a basic way of interacting with the simulation, without using the graphical interface.

We decided to use the only widespread CLI creator in `C`: `readline`[13] (knowing that there exist adaptations of this library on Windows). Our CLI is very simple, but still provides the following features:

- auto-completion of commands and paths;
- history of previously entered commands;
- control commands for the simulation, including:
  - initialisation and destruction of a simulation;
  - launching of a simulation on given duration;
  - data export;
  - changing of some parameters like the number of threads used on each `MPI` cluster.

Our CLI is also independent from the simulation and the precompilation steps – *i.e.* it is independent from the model and is compiled separately from the simulation – so that anyone can code its own interface to the simulation without having to cope with our CLI, using the communication standard to interact with the simulation.

[13]`https://cnswww.cns.cwru.edu/php/chet/readline/rltop.html`

5.5 Examples

There are some examples of `C++` models and corresponding `JSON` files in Appendix A.

6 Interface

The graphical user interface is meant to be an easier way for the user to interact with the simulation and view its results. It is compiled independently from the other parts of ASSASIM and must then be linked to them to work as an interface. It is able to communicate with the simulation using both the message queue defined in the simulation part – to control a simulation – and `JSON` and `UBJSON` files to create and instantiate simulations, and to visualise the data exported by a launched simulation.

Unfortunately, the interface is still in development and some features are not implemented while others are not finished. However its structure is completed and it just requires to link it to the simulation in order to be functional.

Tools

The used GUI library is `Qt` (version 5 and more), mainly because `Qt` is widely used with `C++` and some of us were already familiar with it. Since `Qt` does not offer an easy way to implement graph visualisation, we decided to use use `QCustomPlot`[14], a `Qt` `C++` widget for plotting graphs, alongside with `Qt`.

We also chose to use the statistical computing tool `R`[15] to manage data. Even if `R` is currently not necessary for the interface - the statistic functions implemented are currently far too basic - `R` is a very well-known tool and having thought the data structure with `R`, we can very easily add new statistical features to the interface.

Organisation

We decided to use a multiple document interface[16] (`mdi`) to organise our GUI. The `mdi` organisation consists of one main window that can contain several windows of different types. For our interface, we have three types of sub-windows: the model tool window which allows the visualisation of a model, the instantiation window which is used to specify the initial values of a model for a simulation, and the simulation window which displays some values during the run of the simulation.

6.1 Main Window

At the launching of the interface, a dialogue window (see `Qt` documentation for the technical details) asks what type of sub-window the user wishes to open.

The main window contains several menus:

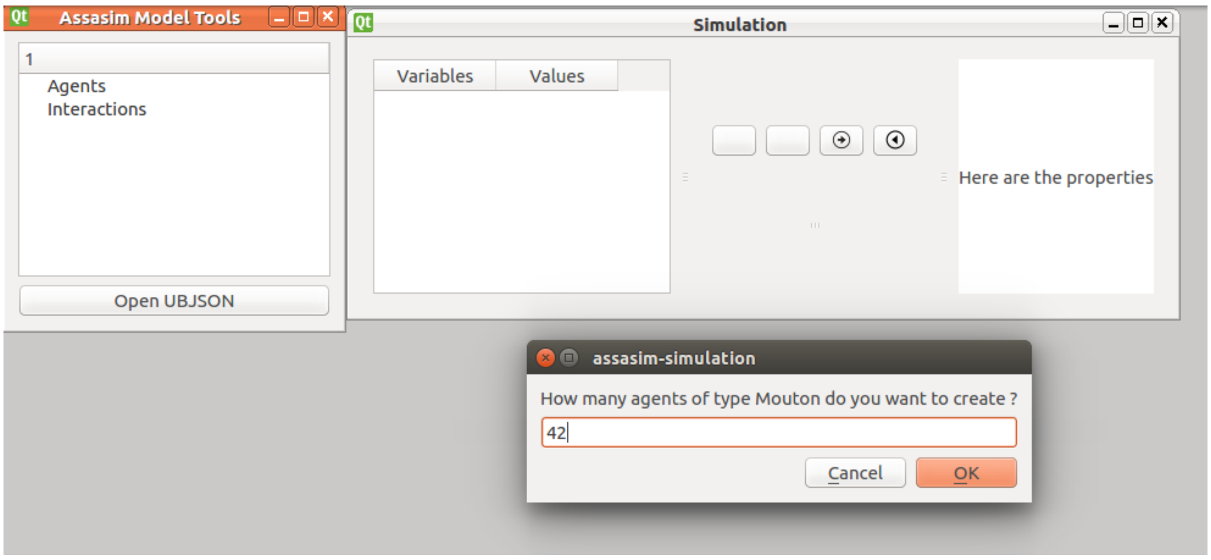
- The File menu which contains button to pop new sub-windows. There is also a button to quit properly the interface.
- The three other menus, Edition, Display and Help were prepared for modularity but are not yet usable (neither by the main window or one of the sub-windows).

6.2 Model

The model is exported by the precompilation process into a binary `JSON` file. It contains the agent types list and the attributes (and their types) of each agent type. The interface is able, *via* a button that opens a filesearch window, to load such a file, and transform it into a structure that is more easily handled by `Qt` (a `QStandardItemModel`). Then, this structure is loaded into a `QTreeView` that lets the user view it: this window uses the Model/View architecture[17].

As of now, the instantiation window is the same as the model window.

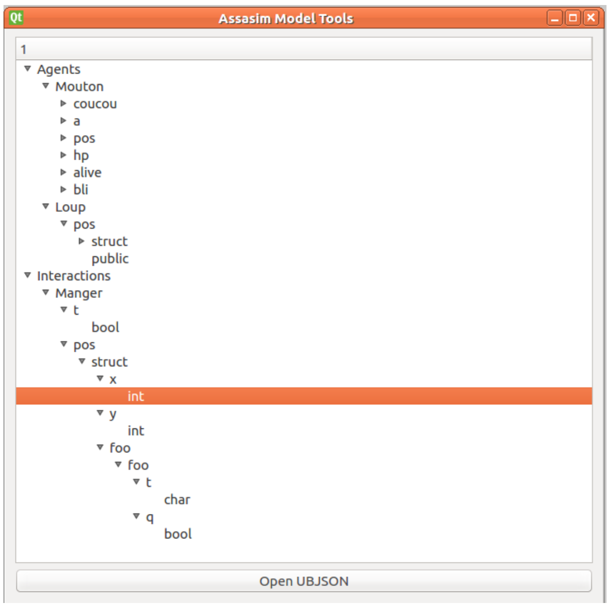
[14] `http://www.qcustomplot.com/`
[15] `https://www.r-project.org/`
[16] `https://en.wikipedia.org/wiki/Multiple_document_interface`
[17] `http://doc.qt.io/qt-4.8/model-view-programming.html`



How many agents of type Mouton do you want to create ?

42

Figure 6: Illustration of the main window



```
1
▾ Agents
  ▾ Mouton
    ▸ coucou
    ▸ a
    ▸ pos
    ▸ hp
    ▸ alive
    ▸ bli
  ▾ Loup
    ▾ pos
      ▸ struct
        public
▾ Interactions
  ▾ Manger
    ▾ t
        bool
    ▾ pos
      ▾ struct
        ▾ x
            int
        ▾ y
            int
        ▾ foo
          ▾ foo
            ▾ t
                char
            ▾ q
                bool
```

Figure 7: Visualisation of a model

6.3 Instantiation

The simulation needs to get instantiation files, which are also JSON files describing the agents at the beginning of the simulation (their number, name, states). During the analysis of the JSON model file in the Model window, instantiation popups appear, asking the user for information. For each type of agent, it asks:

- The number of default and custom agents.
- The default value for each attribute of the agent.
- The IDs and values for each attribute of each custom agent.

This data is then written in a JSON instantiation file.

Finally, this JSON file can be sent to the simulation, because a model-specific function for the simulation was generated during the precompilation.

6.4 Display

The display window can be separated into three parts:

- the right part lists all the agents of all types. The user must select one item in the list for its attributes to be shown in the left part of the window.
- the left part lists all the attributes of the agent or type selected in the right part.
- the central part is for the buttons allowing an interaction with the simulation (like "next step") and also displays the graphs.

The display window interact with the simulation through a message queue whose behaviour has been described in the simulation part. For instance, the button "next step" adds to the stack the messages `run 1` and `export_json`. The messages will then be read by the process running the simulation and the simulation will go one step forward and print the current state of the simulation in a JSON file accessible by the interface.

7 Conclusion and future work

We completed several of our objectives: the simulation is working very well and the simplified syntax in C++ is operational. There are still several parts to improve in the simulation like its installation process, its error handling or the migration of agents across masters for optimisation, but it can be considered as operational and even usable for at least simple models.

However there are still some important parts that could not be finalised: the theoretical language is ready, but the corresponding lexer and parser are still in development, while the essential parts of the interface (instantiation, visualisation) are functional, but the connexion between the interface itself and the simulation could not be finished on time. We also could not provide any official cross-platform support (although our tool in functional on all recent Linux distributions) or even provide a way to install Assasim quickly.

Although we have not fully achieved our initial target of providing an accessible system, since the theoretical aspects and a significant part of the technical implementation have been completed, Assasim is not far from reaching its original goal.

Finally, even when both the language and the interface are completed, there are still many things that can be done to further improve and validate our work. Concerning the simulation, we should study its performance in more depth by elaborating benchmarks, and test it on more complex models – which should be designed and implemented to provide a bank of examples –, to be sure that generality and performance have been reached. It is also possible to add features to both the simplified syntax and the high-level language, like the possibility for the user to manipulate agents that are distributed according a particular topology (a grid for instance, for spatial models). Once all of this is done, we could compare our simulator with other agent-based simulators to further exemplify its advantages and drawbacks. Needless to say, there are a lot of possible improvements for Assasim besides the ones we presented, but we believe these to be the more interesting and relevant ones for the initial philosophy behind our project.

Acknowledgements

We would like to thank our supervisor Silviu-Ioan Filip, who helped us create, develop and organise this project with his advice and decisions, as well as our expert, who reviewed our mid-term report and therefore helped clarify and deepen our goals.

Appendix A

Examples of JSON and C++ models

1 Simplified syntax example

```
if (Citizens[42].male)
```

$\downarrow$

```
if ((*((bool*)AskAttribute(2,42,0))))
```

Public attribute request

```
Send(Citizens[42],Request(money+100));
```

$\downarrow$

```
SendMessage(std::unique_ptr<Interaction>(new
Request(0,id_,0,42,0,this->money + 100)));
```

Sending a message

Figure A.1: Two examples of simplified syntax

2 C++ model example

In this section we present a simple example of a model defined in C++ using our tool.

Figure A.2 is an example of definitions of agents and messages, before the first step of the pre-compilation. `Interaction` and `Agent` are two "void" classes that are used to know if a defined class is an interaction or an agent. Therefore in this example `Contamination` is an empty interaction, and `Individual` is an agent that can be contaminated or dead, while its attribute `remaining_time` represents the number of time steps for it to heal if it is contaminated.

```
class Contamination : public Interaction {
};

class Individual : public Agent {
   public:
      bool contaminated;
            bool dead;
            int remaining_time;
};
```

Figure A.2: Example of definitions of agents and messages

After the first step of the precompilation, the user is able to define the behaviour of the agents he created, like the one presented in Figure A.3. This behaviour makes any `Individual` agent behave the following way, if it is not dead: if it is contaminated, then it chooses another agent to contaminate and sends to this agent a `Contamination` message, then it heals itself after some time; if the agent is not contaminated and it received a `Contamination` message, it becomes contaminated with a 10% chance. It is possible to introduce the random death of contaminated agents, by removing the comments. The second part of the behaviour allows the agent with ID 0 to look how many agents are contaminated and to print this number every 20 time steps, by looking at their public attribute `remaining_time`.

```
void Individual::Behavior() try {

        if (!dead) {
                if (contaminated) {
                        uint32_t to_be_infected = rand()%10000;
                        Send(Individuals[to_be_infected], Contamination());
                        remaining_time--;
                        if (remaining_time == 0) {
                                /*if (rand()%20 == 0)
                                        dead = true;
                                else*/
                                        contaminated = false;
                        }
                } else {
                        if (received_Contamination.size() > 0 && rand()%10 <= 1) {
                                contaminated = true;
                                remaining_time = 10;
                        }
                }
        }

        if (id_ == 0) {
                int count = 0;
                for (const auto &i : GetAgentsOfType(Individual_type)) {
                        if (Individuals[i].contaminated && !Individuals[i].dead) {
                                count++;
                        }
                }
                if (TimeStep()%20 == 0) {
                        std::cerr << count << std::endl;
                }
        }

} catch (const std::exception &e) {
        std::cerr << "[" << TimeStep() << "]" << " In agent Individual" << id_ << ": "
            << e.what() << std::endl;
} catch (...) {}
```

Figure A.3: Example of a behaviour

3 JSON files format examples

Figure A.4 is an example of possible `JSON` instantiation file for the previously defined model. It creates 10000 `Individual` agents with their given default values (`false` for contaminated, ...) except for agents with IDs 0 and 1, which are given customised initial values.

Figure A.5 shows a partial example of data export of our previous simulation after several time steps. It fully describes each agent of all types.

```
{
        "agent_types": [{
                "type": "Individual",
                "number": 10000,
                "default_values": {
                        "remaining_time": 0,
                        "dead": false,
                        "contaminated": false
                },
                "agents": [
                        {
                                "id": 0,
                                "attributes": {
                                        "remaining_time": 10,
                                        "dead": false,
                                        "contaminated": true
                                }
                        },
                        {
                                "id": 1,
                                "attributes": {
                                        "remaining_time": 10,
                                        "dead": false,
                                        "contaminated": true
                                }
                        }
                ]
        }]
}
```

Figure A.4: Example of instantiation JSON file

```
{
        "agents" : {
                "Individual" : [{
                                "id" : 98,
                                "attributes" : {
                                        "remaining_time" : 3,
                                        "contaminated" : true
                                }
                        }, {
                                "id" : 96,
                                "attributes" : {
                                        "remaining_time" : 3,
                                        "contaminated" : true
                                }
                        },
                    (...)
    }
}
```

Figure A.5: Extract of an example of data export JSON file