Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Masterarbeit

**Lukas Grundmann**

## Globale Sensitivitäts- und Unsicherheitsanalyse mit MARS

*Fakultät Technik und Informatik*
*Studiendepartment Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Lukas Grundmann

# Globale Sensitivitäts- und Unsicherheitsanalyse mit MARS

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Clemen
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 12.1.2018

**Lukas Grundmann**

**Thema der Arbeit**
Globale Sensitivitäts- und Unsicherheitsanalyse mit MARS

**Stichworte**
agentenbasierte Modellierung, Sensitivitätsanalyse, MARS, Automatisierung

**Kurzzusammenfassung**
Diese Arbeit beschreibt eine alternative Implementierung des Basic Immune Simulators (BIS) mittels des Multi-Agent Simulation and Research (MARS) Frameworks. Hauptsächlich aufgrund seiner fachlichen Logik ist die Implementierung umfangreich, komplex und daher schwer wartbar. Eine Modelreduzierung durch eine globale Sensitivitätsanalyse (GSA) anstrebend, fokussiert sich die Arbeit primär auf die Durchführung von GSAs mit MARS. Das primäre Ergebnis ist ein Batchsimulationssystem (BSS). Die Arbeit erklärt die Verwendung von MARS BIS sowie des BSS und evaluiert beider Implementierungsstände. Während generell GSAs mit MARS möglich sind, macht eine GSA von MARS BIS der Zeit keinen Sinn.

**Lukas Grundmann**

**Title of the paper**
Global sensitivity and uncertainty analysis with MARS

**Keywords**
agent-based modelling, sensitivity analysis, MARS, automatisation

**Abstract**
This thesis introduces another implementation of the Basic Immune Simulator (BIS) using the Multi-Agent Simulation and Research (MARS) framework. Mainly due its functional logic the implementation is extensive, complex and thus expensive to maintain. Targeting a model reduction through global sensitivity analysis (GSA), this thesis primarily focus on GSA with MARS. The primary result is a batch evaluation system. Finally the thesis gives an usage examples of the new toolset and evaluates its and MARS BIS' current state. The evaluation shows, that MARS generally is capable to do GSAs. But it does not make sense to apply a GSA on MARS BIS, yet.

# Contents

# 1. Introduction

## 1.1. Motivation

Sensitivity analysis estimates the influence of certain model input factors on their output values. It can empirically identify factors, which are non-influential. Input factors can enable or disable certain model features (Saltelli et al. 2008). Thus sensitivity analysis can evaluate their influence on the simulation outputs, on which the answers of a research question base. In conclusion it is possible to reduce the model complexity by removing model parts with low importance (for a certain research question). Beside model reduction sensitivity analysis is also useful for other applications, including model calibration, uncertainty analysis or dominant controls analysis (Pianosi et al. 2016).

Multi-agent based models can map single individuals and their interactions quite directly. Thus this method is on the one hand useful to create representations of complex systems. On the other hand, it often raises the question, which aspects of the individuals in the real system to model, if the role of certain aspects in the mapped system is not fully understood. Each new modelled aspect leads to a more complex model and eventually the maintaining of a model functional and technical exceeds the available developer resources. Multi-agent based simulations have a high demand of computional resources depending on the amount individuals.

Advised software design and usage of a proper simulation system tackles the problem technically. The Multi-Agent Research & Simulation (MARS) group[1] at the HAW Hamburg develops the equally named system (Hüning et al. 2016). Model libraries hide several technical aspects of agent based simulation and support the creation of modular models. MARS' cloud service's allow to separate the management of input data and simulation result analysis from the model itself. Further MARS aims to scale well with the available computer resources.

[1] www.mars-group.org; The author of this thesis is member of the group.

The immune system is essential for the human body, since it is able to identify a great variety of danger's sources and to fight them. Even more, the complex system is able to learn and to give a quicker and stronger response, if a similar situation occurs (Neumann 2008). Beside the classical research methods of biologist on a complete living systems or parts of it, computer simulations allow to watch all variables of interest in a system on ethically uncritical resources. Many aspects are still subject of research. For example there are even multple theories, how the immune system keeps the information about former attacks (Neumann 2008). The required amount of agents easily can be high even with strict focus on small body parts, because a human body hosts billions of immune cells (Neumann 2008). Further the diversity of the immune cells is high.

The author developed an implementation of the Basic Immune Simulator by Folcik, An, and Orosz (2007) using MARS. While both implementations basically share the functional model logic, the MARS implementation's architecture differs. The technical design was done from scratch and integrates several features and paradigms from MARS. With the implementation almost finished the question arises, how to go ownward with the results.

One problem of both implementations is clearly their extent. The original implementation has around ten classes, which each has thousands lines of code. Meanwhile the MARS implementation has hundred of classes, which are smaller than the classes of the original implementation. Overall, there is no significant difference regarding the amount of code. Improvements of MARS have the potential to technically reduce the source code size. Another approach is to evaluate more closely, if really all of the many classes are required for achieving the simulation outcomes, which Folcik, An, and Orosz (2007) observed (and influenced their conclusions).

## 1.2. Goals

A primary goal of this thesis is to build a framework based on MARS for sensitivity analysis. The thesis demonstrates, how well the resulting system works. It should identify potential areas of improvements.

Another important target of the thesis is to document the design of the alternate Basic Immune Simulator. Further the objective is to evaluate the implementation state of the MARS Basic Immune Simulator. An ultimate goal is to reach a state, which actually allows to apply sensitivity analysis methods on the latter.

## 1.3. Structure

The second chapter describes the summarizes knowledge for the following chapters. Its contents comes mostly from the works of others. The subsections introduce the immune system in general, agent based models of the immune system, the original Basic Immune Simulator, as well as the MARS simulation system. Finally the greatest part of the second chapter summarizes several aspects of sensitivity analysis. The topic order is somehow chronological. Great parts of the MARS based Basic Immune Simulator exist before the author started to study the sensitivity analysis methods.

The third chapter describes the design of the MARS Basic Immune Simulator implementation. After an overview, several subsections explain design decisions and public interfaces of the components. The overall goal is, that the third chapter is a guide for another developer to get along with the project structure and the used concepts.

The fourth chapter starts with requirements, which must be met to support sensitivity analysis (as described by chapter two) with MARS. It suggests a workflow, which merges the common sensitivity analysis process with the typical model evaluation procedure using MARS. The fourth chapter continues with a more detailed design description of tools, which the author developed to support the suggested workflow.

The fifth chapter mainly is a tutorial, which shows the usage of the previously described tools. It additionally shows, how well the presented systems work. First a section outlines the used environment. The next section shows, how to do evaluations of the Basic Immune Simulator implementation. The final subsections describe the sensitivity analysis with an example model, and the results.

The sixth chapter discusses the conclusions from the previous chapters in relation to the introduction and the goals it defined. Finally the last chapter contains the conclusion and outlook.

# 2. Related work

## 2.1. Introduction of the immune system

The decentralized human immune system consists of trillions of cells and antibodies (Schütt and Broeker 2009). There are dozen of different specialized cell types. Depending on its type a cell either belongs to the innate or the adaptive immune system. While the innate immune system is able to detect and fight almost every danger since birth the adaptive immune system trains itself over a life span getting more powerful (Neumann 2008).

The adaptive subsystem is able to fight intruders like virus by respecting their foreign genetic structures (antigens) (Neumann 2008). The adaptive immune systems regulates the strength of the immune response to specified antigens. Also it controls the locality of a specified response. The adaptive immune response is efficient. Antigen matched antibodies are produced. Cells are activated and reproduced, which are specialized to detect and destroy other cells infected by a specified antigen (Neumann 2008).

To enable the adaptive immune system the innate system must propagate the antigen information. This is primary done by dendritic cells. After finding antigen structures, they present them over their receiptors and move to the lymph nodes of the body, where many adaptive immune system cells reside (Schütt and Broeker 2009). There a dendritic cell activates T cells, which match a represented antigen. T cells are developed in the thymus and undergo a selection, which ensures that for many possible antigen structures matching T cells exist (Schütt and Broeker 2009).

After activation, cytotoxic T cells search and destroy infected other cells. Meanwhile T helper cells activate matching B cells, which then produce the antibodies. Also they emit signal molecules, which stimulate the reproduction of all immune system cells. Further some T helper cells move to the infection sites. There they attract immune system cells (adaptive and innate) by production of signal molecules (Neumann 2008).

Even after a immune response successfully removed intruders from the body, specialized and long living T and B cells remain in the body (Schütt and Broeker 2009). They can be activated by any innate immune system cells presenting the specified antigen. In addition, a low antibody production is kept up. This way the adaptive immune response starts much quicker if intruders with the same antigen. Thus the immune system gets stronger with every infection (Schütt and Broeker 2009).

## 2.2. Agent based modeling of the immune system

This subsection gives an overview of multiple models of the immune system, which all have in common, that agents map the immune system cells. In some models, antibodies and antigens are also agents (M. Bernaschi and Castiglione (2001), Rapin et al. (2010)). In contrast, other models just represent the amount of antigens and antibodies (Folcik, An, and Orosz (2007), Song et al. (2012)). Models also differ regarding their scope. While some concentrate on the adaptive immune system (M. Bernaschi and Castiglione (2001), Rapin et al. (2010)), others also represent the cells of the innate immune system (Folcik, An, and Orosz (2007)}, Song et al. (2012)). In addition, models exist, which model specific body parts and their special aspects (Wendelsdorf et al. 2012).

The related work can be further classified into their approach of antigen representation. One group of models does not map the molecular structure of antigens, matching antibodies and cell receptors (Folcik, An, and Orosz (2007), Song et al. (2012), Wendelsdorf et al. (2012)). On the contrary, other models represent the molecular structures by bit strings (M. Bernaschi and Castiglione 2001). In this case the number of different bits specify the propability of a match between antigen and antibody (or receptor). One model uses algorithms and data structures of bioinformatics to represent molecules (Rapin et al. 2010). The Latter allows the integration of informations from biological databases.

All models represent most physical and chemical processes simplified. For example, the agents representing body cells often move randomly (M. Bernaschi and Castiglione (2001), Wendelsdorf et al. (2012)). However, influence of the movement by signals also exists (Folcik, An, and Orosz (2007), Song et al. (2012)). Many models map the room through grid structures. There are two and three dimensional spatial representations. Some models split the room into apart subareas (Folcik, An, and Orosz (2007), Wendelsdorf et al. (2012)). Most models represent

the time using simulation steps. Continuous time mapping (e.g. using event driven simulation (Tay and Jhavar 2005)) is rare.

Some simulators provide a graphical user interface (GUI) for parameter input or visualization of model aspects (Folcik, An, and Orosz (2007), Sarpe and Jacob (2013)). Simulators without a GUI often utilize proprietary declaration languages for parameterisation and textual result output (Wendelsdorf et al. (2012), M. Bernaschi and Castiglione (2001)). Many models only need several global parameters as input. The model of Rapin, Lund, and Castiglione (2011) additionally integrates molecular information about antigens and cell receptor from biological databases.

The huge agent amounts (related to the human cell numbers) are a big challenge for agent based simulations. Therefore all related works only simulate few cubic millimetre of the body. Even so the explosive grow of the agent numbers in some simulation scenarios is a prominent technical challenge for computer science (Folcik, An, and Orosz 2007). Thus some researchers concentrate on distributing their immune system simulations over many compute nodes (Wendelsdorf et al. (2012), Andrew Emerson (2007), M. Bernaschi and Castiglione (2001)).

Many fold domains provide requirements regarding information management, visualisation and distribution to the computer science. Thus it makes sense to base a simulator on a generic agent based modeling and simulation framework. This way a fold domain can profit from techniques, which were originally developed for another domain. Some existing agent based immune system simulators are based on frameworks (Folcik, An, and Orosz 2007, Song et al. (2012)).

## 2.3. Basic Immune Simulator

Folcik, An, and Orosz (2007) developed an agent based immune simulator to research the relation between the innate and the adaptive immune system. One important aspect has been the analysis of the the dendritic cell's role (Folcik, An, and Orosz 2007).

The environment of the model is divided into three spatial zones (Folcik, An, and Orosz 2007). One zone represents a part of infected tissue and another one maps a piece of the lymph system. The last zone is the travel area for agents between the other two zones. A two dimensional grid represents each zone. Each grid cell can hold an infinite number of agents (Folcik, An,

and Orosz 2007). Agents leaving a border reappear on the opposite border. Discrete simulation steps map the time. The model doesn't specify a quantitative spatial and temporal mapping.

The agents emit signals and check signal values in their neighbourhood (Folcik, An, and Orosz 2007). For that each grid cell stores the amount of each signal. The signals diffuse over the grid with each simulation step (Folcik, An, and Orosz 2007). Antigens and antibodies are mapped as signals.

The agents represent immune system cells and tissue cells (Folcik, An, and Orosz 2007). Later are infection target and reproduction hosts for a virus. Each agent's logic is based on a state machine. The agent logic's input consists of the signal values and agents in the agent's current and neighbored spatial grid cells (Folcik, An, and Orosz 2007). After state change, interaction with other agents and change of signal values an agent can move into a neighbored grid cell. For that the agent chooses the target grid cell randomly or by the signal values. Agents can change to another spatial zone over portal agents. Those control and manage the agent's zone change and exchange signals between the zones (Folcik, An, and Orosz 2007).

Folcik et al implemented their model in JAVA using the RepastJ framework (Folcik, An, and Orosz 2007). They published their state machines with textual and graphical description and the binaries and sources of the implementation's 2007 version.

## 2.4. MARS framework and cloud

### 2.4.1. Overview from the user's perspective

The Multi Agent Research and Simulation (MARS) framework offers a toolset for agent based simulations. It includes libraries, which support the development of models. Further a cloud based system manages models, input data, simulations and their results. Resources are divided into projects and the system provides a simple, user based access handling. At the time of writing, browser driven, graphical user interfaces exists for resource management, visual analytics of results and live visualization of spatial simulations.

Models written with MARS are divided into agents and layers. Each agent represent an active individual of the modelled system and is the instance of a specific type (e.g. elephant, immune system cell ...). An agent's type basically implements the rules of its behavior, whose effects depends on the individual agent's parameters (including the position in spatial models). The

agent's environment is divided into layers. Each layer represents a domain specific aspect of the mapped environment (e.g. temperature, concetration of hormone ...). A layer can be passive or active (like the agents).

The MARS library provides basic layers, basic agents and additional, commonly needed components. At the moment of writing these layer base types exist:

- Time series layers, which allow agents to query time lines (e.g. global temperature development of the last 20 years).
- Obstacle layers provide spatial boundaries to the agents.
- Potential field layers support agents in finding certain spatial hotspots.
- Geo Information System (GIS) layers provide geospatial data to the agents.

The MARS libraries provide classes to develop agents based on an sensor, reasoning, action pattern. An agent's sensors observe its environment. Their results lead to certain decisions by the reasoning, which finally conclude into actions. The latter can be changes of the agent's state, its environment or interactions with other agents. Classes exist for geospatial, grid-based or continuous position management. Additional helper classes support the movement of the agents. Finally base agents help the model developer with the initial registration at the position management (for spatial agents) and at the execution engine.

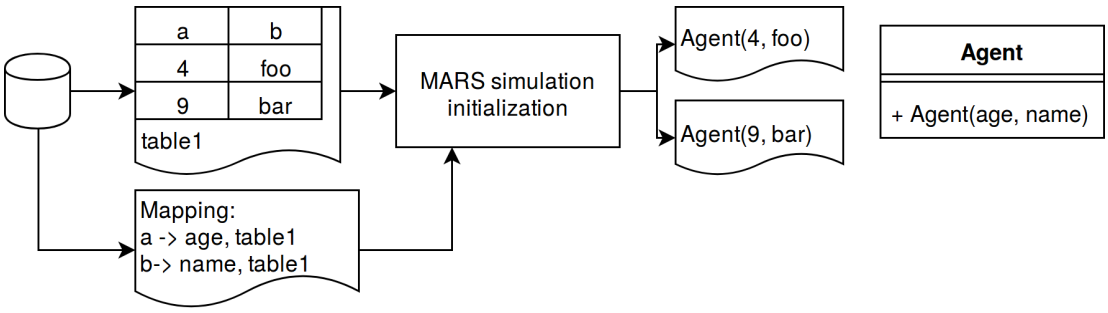


Figure 2.1.: Simple example of MARS' table based agent initialization.

MARS can initialize agents based on data tables, which the user uploaded before the simulation. A scenario description provides the mapping of table columns on agent parameters. Each row corresponds to a certain agent. Figure 2.1 shows an example. The *Agent* type takes the parameters *age* and *name*. Further the *Mapping* binds column *a* of *table1* on the parameter *age* and *b* on *name* (of *table1*). Based on this information the rows of *table1* conclude in two

agent instances. It is possible to map the columns from different tables on an agent parameter. Also constant values can be assigned. In the example *7 -> age* would result in both agents initialized with an *age* of 7. Further data on layer and column on time series layer mappings can be defined.

A simulation happens in discrete, constant steps called ticks. Each tick relates to a certain amount of time, which the model developer defines. In each simulation step, MARS triggers all agents exactly once (e.g. for starting the proposed sensing, reasoning, action pattern). In case of the proposed pattern the agent then senses, reasons and acts. MARS does not guarantee an order of execution. But it ensures, that all agents have finished with one simulation step, before it proceeds with the next step. Active layers can provide three methods. One is executed before an actual simulation step. Another one MARS calls together with the agent evaluation. Finally the third runs after the actual simulation step. Interactions between agents base on synchronous method calls.

At the time of writing, MARS supports two ways of recording agent information at the end of each tick. In the legacy mode an agent has to implement a certain interface and implements its method. The latter provides certain agent data as a dictionary, which the MARS system will store together with metainformation (e.g. tick) into a database. The modernized recording separates the output definitions from the model (Dalski, Hüning, and Clemen 2017). A result output configuration specifies agent attributes, whose values MARS stores after tick.

### 2.4.2. Typical simulation workflow using MARS

While previous subsection focus on the model development with MARS, this subsection outlines the processes and documents around a model evaluation (simulation). The diagram fig. 2.2 shows the (user) workflow leading to a simulation, the involved documents and their relation. The dotted arrows visualize references between documents, while the straigh lines show input / output relations between processes and documents.

Actually MARS stores all of the documents (except the ones initially uploaded by the user) in databases. The diagram does not show this relations for clarity.

At the beginning the user uploads *Input files* and the *Model* binary. The following *Import* processes the contents, stores it into databases using internal formats and creates a *Metadata* resource for every uploaded file. Basically each *Metadata* resource contains user defined

metainformation and the internal references to the databases. Further *Metadata* for tables provide column informations. The *Import* process scans a *Model* binary and identifies the agents and layers it provides. Further it analyzes, which input parameters each agent requires and what attributes it provides for output. Finally the related *Metadata* document stores the results of this model reflection.

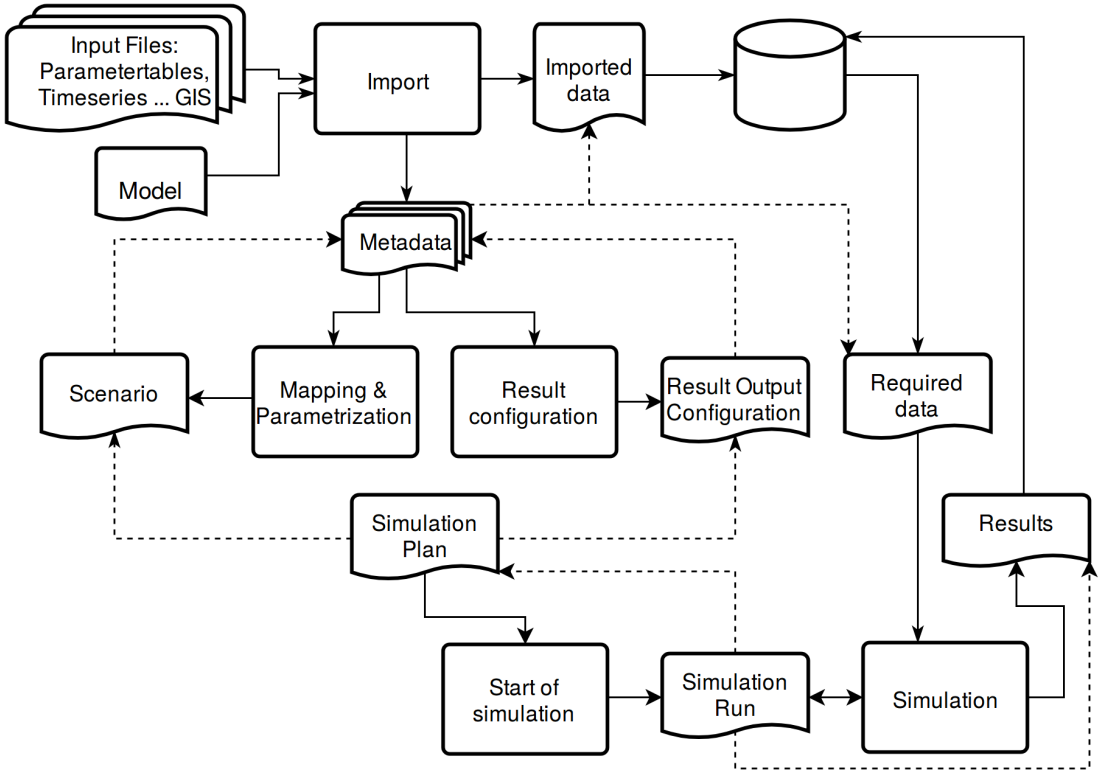


Figure 2.2.: Overview about processes and documents involved in a simulation with MARS.

While the *Mapping & Parametrization* process the user selects a previously uploaded *Model* and maps data on its agent's parameters and layers (see sec. 2.4.1). Further not mapping related parameters are defined, e.g. the simulation duration. A *Scenario* stores the mapping and the parameters. MARS does several verification checks on every *Scenario*. Incomplete (e.g. missing mapping) or invalid (e.g. not allowed file type) *Scenario*s can not be used for following processes. Meanwhile the user can create *Result Output Configuration*s for simulations not using the legacy output (see sec. 2.4.1).

As soon a *Scenario* and *Result Output Configuration* (for the same model) are complete, the

user can combine them in a *Simulation Plan* (process not shown explicitly in fig. 2.2). Based on the latter, a *simulation can be started*. This will create a *Simulation Run* document, which combines all necessary information to start the *Simulation*. Further it stores the state of the *Simulation* process. Finally it references the *Results*, which the *Simulation* generates.

### 2.4.3. Distributed simulations

A MARS simulation can be distributed. In this case layers and agents are statical spread over the calculation nodes. Synchronous method calls map the interactions between agents and layers or other agents. MARS archives that by using proxies (Hüning 2016). Those are named *Agent Shadows* in the case of agents. The proxies marshal method calls into network packages, which MARS then passes to the calculation node hosting the actual agent or layer. After getting the response, the proxy (*Agent Shadow*) deserializes it. While MARS creates layer proxies automatically at simulation start, the *MARS AgentShadowingService* produces *Agent Shadows* on demand by a global unique identifier. At the time of writing the simulation distribution features are experimental. The MARS Cloud does not offer an user interface or services to manage the required configurations. Though this might change in the future. Thus it is important to design a model distributable, unless this feature is not required. Especially method calls, which might go over proxies, must be defined carefully. Parameters and responses must be serializable. For example instead of the technical reference to an *Agent Shadow*, the global identifier of its agent must be passed.

### 2.4.4. Introduction of MARS Cloud's services

Subsection sec. 2.4.2 outlined the single processes involved in a simulation. This subsection gives an overview about the MARS Cloud services, which manage the processes. All services are designed stateless and can have multiple instances. All instances of one service type access those service's persistent data directly. But the MARS Cloud architecture requires an instance of one service type to access another service's datasets only over the Application Programming Interface (API). Most service's APIs base on HTTP, while two services offer GRPC[1] interfaces. Most HTTP APIs are described as Swagger[2] definitions, which (as MARS convention) the

[1] https://grpc.io
[2] https://swagger.io

service's repositories provide in the directory *interfaces*. The GRPC definitions are in the *proto* directories of the related service's repositories.

Table 2.1 gives an overview about the elementary services, their tasks, managed resources and project[3] names. Several services, especially helper services for certain file formats, are not listed. The *api service*s are reverse proxies, which handle client requests with applying authentication and authorization.

Table 2.1.: MARS service overview (*1: mars-result-mongo-query-service)

| Service | Task | Resources | Project |
|---|---|---|---|
| File | import management | - | mars-file-svc |
| Metadata | metadata management | Metadata | mars-metadata-svc |
| Scenario | scenario management | Scenarios | scenario-svc |
| Result Config. | result cfg. management | Result Ouput Con. | resultcfg-svc |
| SimRunner | simulation management | Simulation-Plans, -Runs | sim-runner-svc |
| SimMonitor | sim. progress observation | - | mars-sim-mon-svc |
| User | user management | Users | mars-user-svc |
| Project | project management | Projects | mars-project-svc |
| TeachingAPI | api service, authentication | - | mars-teachingapi-svc |
| ResultQuery | api service, result access | - | (*1) |

### 2.4.5. Deployment of the MARS services and simulations

All services and the simulations are deployed as containers (process based virtual machines). At the time of writing Kubernetes[4] orchestrates those containers. Also each simulation runs in its own container. The MARS *SimRunner* service builds an initial file system image for each *Simulation Plan* instance. Basically the service packs the simulation runtime environment together with the model binaries into the image. Then it pushes the image to a global registry. Kubernetes locates the container for a *Simulation Run* on a suitable calculation node. The latter pulls the image and creates a container based on it. The MARS simulation runtime in the

[3]Repository is the located at https://gitlab.informatik.haw-hamburg.de/mars/REPO_NAME
[4]https://kubernetes.io/

container fetches the required input data by direct communication with the MARS services and databases. Kubernetes balances the load on multiple instances of the same service.

## 2.5. (Global) Sensitivity Analysis

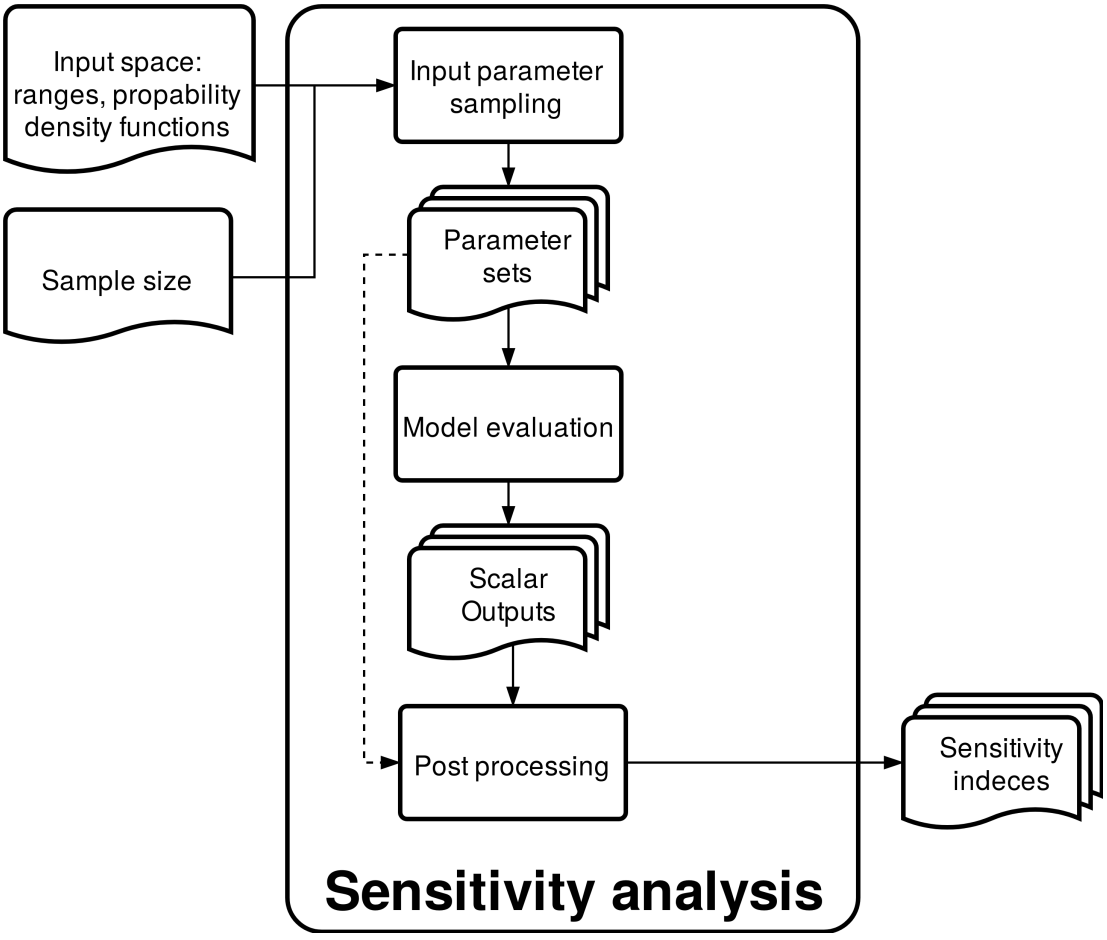


Figure 2.3.: Typical sensitivity analysis workflow

The goal of sensitivity analysis is to determine the impact of input factor changes on output changes of a model's simulation (Pianosi et al. 2016). Basically all sensitivity analysis approaches run simulations multiple times with different sets of model parameters, which are sampled from the input space. Each scalar parameter specifies the concrete shape of a certain

input factor. It can just define a model constant or structurally configure the model (Saltelli et al. 2008). Each simulation results in one or more scalar output values. Further the sensitivity analysis evaluates differences in each output's values in relation to the inputs. The results are often sensitivity indices, where each index represents the influence of an input on the variability on the output in the given setup. Graphic fig. 2.3 gives an overview about the typical workflow. Meanwhile the next subsections will cover certain aspects of sensitivity analysis.

### 2.5.1. Uncertainty analysis and its relation to sensitivity analysis

Uncertainty analysis evaluates the uncertainness of a model and its simulation results. Meanwhile a sensitivity analysis can estimate, how much each of an models input factors contributes to the output's uncertainty (Saltelli et al. 2008). Some researchers also define sensitivity analysis this way (Saltelli et al. 2004). Thus a sensitivity analysis can follow an uncertainty analysis, e.g. to determine input factors to be treated with care.

Another connection between the two analyses are the often similar methods. Also uncertainty analysis can base on multiple model evaluations using different input samples. Since sensitivity analysis explores the input space, it requires a more complicated sampling strategy (Helton et al. 2006). But a uncertainity analysis can reuse these more sophisticated samples and the related model evaluation. Since the final additionally costs (e.g. for different post processing methods) are rather small, analyses often cover uncertainty and factor sensitivity (Pianosi et al. 2016). Helton et al. (2006) presents an overview about sampling based uncertainty analysis methods and refers to analyses of uncertainty, which does not originate from the input factors. Meanwhile the remainder of this thesis focus on sensitivity analysis with MARS.

### 2.5.2. Types of sensitivity analysis

A local sensitivity analysis only evaluates the effect, which a single factor has on its own (Saltelli et al. 2008). Interactions between multiple parameters are ignored and are not part of the analysis' results. Meanwhile the global sensitivity analysis also estimates the influence of a factor through interactions with other factors (Pianosi et al. 2016). It can happen that factors are influential only in combination (Nossent, Elsen, and Bauwens 2011).

Beside doing a quantitative analysis with calculating the sensitivity indices it is also possible to do a qualitative analysis. This is mostly done by using visualization (Pianosi et al. 2016). One

powerful diagram type for it, are scatter plots. Saltelli et al. (2008) shows how certain shapes of the points gives a hint about the influence of a parameter on the output. A qualitative analysis can be helpful to make design decisions for a following quantitative analysis.

### 2.5.3. Input factors and the output

Before doing a sensitivity analysis the input factors must be defined. Since every evaluated factor is a dimension of the input space (Saltelli et al. 2008), the count of input factors is a driver of the minimum sample size. Thus the analysis should concentrate on the input factors of interest and fix the others. If prior sensitivity analysis clearly shows, that certain input factors have an negligible influence, they can be fixed in further experiments (Saltelli et al. 2008).

The sensitivity analysis methods require the input parameters to be numeric values. Some methods further require them to be from a metric space (Pianosi et al. 2016). In most cases the sampling generates normalized values between 0 and 1 (Saltelli et al. 2008). For the transformation of them into the shape expected by the model, the researcher has to specify the possible ranges for each input factor. It should be considered, that different ranges can result in different sensitivity analysis results. Thus the latter should be taken with care, if the ranges are poorly known (Pianosi et al. 2016). If the actual model input is more complex, the researcher must express certain aspects of the input data (e.g. amount and distribution of spatial entities) with numeric parameters. The same is the case, if an input factor is the activation of model parts. One other possible approach is to provide sets of complex input data / model implementations and let a numeric parameter decide, which element of the set to use for a simulation run (Pianosi et al. 2016). Further several sensitivity analysis methods rely on independent input factors (Saltelli et al. 2008). Finally some methods incorporate the input distributions and thus their performance depends on the knowledge about those (Pianosi et al. 2016).

Similar to the numeric parameters generated by common sampling methods, the post processing methods expect scalar model outputs (Pianosi et al. 2016). If the output is more complex than that, extensions to the evaluated model must convert it to a scalar, which represents output aspects of interest. Multiple outputs can be evaluated separately (using the same input samples and repeating the post processing) or in combination (Minunno et al. 2013).

### 2.5.4. Goals of sensitivity analysis

Following three types of findings are most common (Pianosi et al. 2016)(Saltelli et al. 2008):

- Ranking of the input factors based on their influence on a certain output's values.
- Screening to identify input factors with no significant influence on output's diversity.
- Mapping determines areas in the input parameter space, which result in output values of interest. (e.g. values over a certain boundary)

The purpose is an important point in the decision, which analysis methods to apply.

### 2.5.5. Overview of sensitivity analysis methods

One major classification of the available methods is their general sampling approach (Pianosi et al. 2016):

- Especially methods related to local sensitivity analysis vary only one input factor at a time to analyze its influence on the output. Meanwhile all other factors are fixed.
- Most of the global sensitivity analysis methods change all input factors while sampling the parameter sets for the simulations.

Since the explored input space with only one varying factor has only one dimension, few samples are enough to explore it properly (enough for sensitivity analysis). But the disadvantage is that it does not give insights about the interactions between the input factors.

#### 2.5.5.1. One factor change per sample based methods

The outputs without and with perturbation of an input factor can be visualized together. A more quantitative approach is to calculate the partial derivate of the output function and use it as sensitive index for the input factor (Pianosi et al. 2016). Aggregating these individual indices for multiple parameters allows to evaluate interactions (Morris 1991) (and do a global sensitivity analysis). Methods based on this approach differ regarding the selection of the fixed points, determination of the variation magnitudes (of the input factors) and type of aggregation (Pianosi et al. 2016). Some methods do not aggregate finite differences but transformations (e.g. squares) of them.

Morris (1991) provided the most established (Pianosi et al. 2016) method based on evaluating multiple perturbartions of single factors and aggregating the results. It is also often called Elementary Effect Test (Saltelli et al. 2008) and calculates the sensitivity index for a factor *i* as the mean of a specified amount of elementary effects (*EE*). The latter are finite differences $EE = \frac{g(x_1,...,x_i+\Delta,...,x_M)-g(x_1,...,x_i,...,x_M)}{\Delta} c_i$, where $g$ is the model evaluation function, $x$ a sample from the input space, $M$ the factor count, $\Delta$ the perturbation and $c_i$ a scaling factor to make inputs with different units of measurement comparable (Pianosi et al. 2016). The standard deviation of the elementary effects gives a hint about the strength of interactions involving the factor (Saltelli et al. 2008). Meanwhile the original sampling approach by Morris (1991) rasterizes the input space into an uniform grid. It then explores the space by building a trajectory for each elementary effect, which each has $M$ (factor count) + 1 points and random start within the grid (Pianosi et al. 2016). Two sequent points differ in one factor (dimension) at a time by $\Delta$ (the perturbation), so that each trajectory allows the evaluation of one EE per factor (Pianosi et al. 2016). Several variants of the sampling strategy exist, which all have in common that for $r$ elementary effects $r(M + 1)$ input samples need to be evaluated to cover all input factors (Pianosi et al. 2016).

#### 2.5.5.2. Multiple factor changes between sequent samples based methods

Correlation and regression analysis methods derive the information about sensitivity from statistical analysis of input and output datasets generated by Monte Carlo simulation and many of them rely on assumptions of linearity and / or monotonicity between inputs and outputs (Pianosi et al. 2016). Since such assumptions can often not be made for a complex, agent based model, this class of sensitivity analysis methods is no further covered here.

Regional sensitivity analysis (also called Monte Carlo filtering) mainly identifies parts of the input space, which result to output values above / below a certain threshold or (do not) match some pattern. Thus this analysis approach allows *mapping* (as defined in sec. 2.5.4) (Pianosi et al. 2016). The pattern or threshold criterion divides the input values into two groups based on their output. The *bevarioural* set contains the input samples, whose output value conform to the expected pattern (or is above the threshold) (Spear and Hornberger 1980). Meanwhile the *non behavioural* set contains the other input samples. Besides a visual comparison of the both sets for information on factor mapping, the divergence between the set's distributions can be used as sensitivity index (Pianosi et al. 2016). The Kolmogorov-Smirnov statistic is one possible method to calculate the divergence *d*: $d_{m,n} = sup_x|S_n(x) - S_m(x)|$, where $S_n$ and

$S_m$ are the sample (empirical cumulative) distribution functions for $n$ behaviours and $m$ non behaviours (Spear and Hornberger 1980). The index may be used for ranking of parameter's, but not for screening. The reason is, that input factors influencing the output only through interactions may have the same distribution functions for both sets (Pianosi et al. 2016). Thus a low divergence alone is not sufficient to determine an unimportant input factor. Meanhwile an advantage of regional sensitivity analysis is, that it does support any type of ouput (Pianosi et al. 2016). It even is an exception to the rule, that the model evalutation outputs must be scalar. But still a criterion to divide the outputs must be defined and verified. As soon there is no clear distinction possible between "good" and "bad" behaviour, this becomes problematic.

Meanwhile another class of analysis methods takes the output variance as estimator of a input factor's sensitivity. The output values are calculated by Monte Carlo experiments with optimised sampling. Usually variance decomposition estimates two types of indices. Following formula calculates the first order index $S_i$ for a factor $i$ (Saltelli et al. 2008):

$$S_i = \frac{V_{X_i}(E_{X_{\sim i}}(Y|X_i))}{V(Y)}$$

where $X_i$ is the factor, $X_{\sim i}$ are all other factors and Y are the output values. Further the expectation operator $E_{X \sim i}$ is the mean of the outputs with a fixed $X_i$ (Saltelli et al. 2010). Finally $V$ are the variances. While a first order index indicates the direct influence of a factor (ignoring its interactions), the total effect index includes the interaction effects. The latter $S_{Ti}$ is calculated by the formula (Saltelli et al. 2008):

$$S_{Ti} = \frac{E_{X \sim i}(V_{X_i}(Y|X_{\sim i})}{V(Y)} = 1 - \frac{V_{X_{\sim i}}(E_{X_i}(Y|X_{\sim i}))}{V(Y)}$$

Total effect indices are especially useful for screening, since a low value is a sufficient condition for a factor, which has no influence (Pianosi et al. 2016). It is possible to calculate sensitivity indices of higher order to further evaluate the interactions between certain factors (Borgonovo 2007). Different sampling strategies are available (e.g. discussed by Saltelli et al. (2010)), which goal is to approximate the variance with few samples as possible. Though variance based methods are expensive (compared to the previously introduced methods) regarding the required sample count. Beside the computational costs the implicit assumption, that the variance fully captures uncertainty, may be problematic regarding multi-modal or higly-skewed output distributions (Pianosi et al. 2016).

Thus a further group of methods calculates sensitivity indices based on the probability density functions of the output (Pianosi et al. 2016). The latter represents the output distribution's shape better than its variance (for a certain input parameter set). The general form of a density-based sensitivity index (for factor *i*) is $S_i = stat_{x_i} divergence[f_y, f_{y|x_i}(\cdot|x_i)]$, where $f_y$ is the output's probability density function based on varying all input factors, while $f_{y|x_i}$ is the output's probability density function with one input factor fixed (Pianosi et al. 2016). The *divergence* measure can be for example the area between two probability density functions with *stat*istic being the mean (Borgonovo 2007). Since it is complicated to calculate probability density functions based on empirical results, some methods use cumulative distribution functions (e.g. PAWN by Pianosi and Wagener (2015)). Density-based sensitivity analysis can be restricted to a subrange of the output and often does not require a tailored sampling strategy (e.g. Pianosi and Wagener (2015) just uses random samples) (Pianosi et al. 2016). The required count of samples has the same magnitude as for variance-based methods.

#### 2.5.5.3. Summary with comparison

Table tbl. 2.2 gives a summary about the groups of sensitivity analysis (SA, except correlation & regression) methods, which cover interactions (e.g. do global SA). *Number of factors* is a suggestion by Saltelli et al. (2008) and they introduce a grouping method (not covered here) for a higher number of input factors while screening. *r* (typically around 4 - 10, see Saltelli et al. (2008)) is the number of elementary effects (see sec. 2.5.5.1), *k* is the number of input factors and *N* can basically be set to any value. The next section discuss *N* further. While Pianosi et al. (2016) gives a broad overview about methods, Saltelli et al. (2008) explains several methods with more background and practical examples.

Table 2.2.: Overview about the previously introduced class of sensitivity methods (based on Saltelli et al. (2008) and Pianosi et al. (2016)).

| Aspect | Elementary Effect Test | Regional SA | Variance / Density based |
|---|---|---|---|
| Samples from | levels | distributions | distributions |
| Number of model runs | r * (k + 1) | k * N, N > 100 | (k + 2) * N; N, N > 1000 |
| Number of factors | 20 - 100 | < 20 | < 20 |
| Ranking | yes | yes | yes |
| Screening | yes | no | yes |

| Aspect | Elementary Effect Test | Regional SA | Variance / Density based |
|---|---|---|---|
| Mapping | no | yes | no |

#### 2.5.5.4. Selection of method(s), sample size and number of model evaluations

The decision for a method depends primarily on the goal (sec. 2.5.4) of the sensitivity analysis. Further the available computational resources (=> doable amount of model evaluations) are an argument. Finally certain model properties (e.g. non linearity, skew of the output) expulse the application of specific methods or let them perform not well enough (Pianosi et al. 2016). A common approach is to start with a Elementary Effect Test (sec. 2.5.5.1) to exclude clearly non-influental factors from further analysis with more advanced methods (Saltelli et al. 2008). Meanwhile the sampling approach mainly depends on the selected analysis method. While several alternatives exist for Elementary Effect Tests (or local sensitivity analysis methods), many variant-based methods depend on tailored sampling (tightly coupled with post processing) (Pianosi et al. 2016). Further Regional Sensitivity Analysis and and density-based methods basically allow any (quasi-)random sampling. Most commonly used are Latin-Hypercube and Sobol' quasi-random sampling (Pianosi et al. 2016). Some sampling methods are better suited for certain model properties. But it is likely, that other settings (e.g. input variability or the output definition) influence the analysis' results more (Pianosi et al. 2016).

The main driver for the required amount of samples is the chosen sensitivity analysis method and for several methods the number of input factors (see sec. 2.5.5.3). Meanwhile the exact number of input samples depends on the actual problem and the required convergence level of the sensitivity indices (Pianosi et al. 2016). Sensitivity indices for some input factors might converge quicker (measured in required sample amount) than others. While the sensitivity indices for non-influental factors often converge quickly, much more samples are needed for influential factors (Nossent, Elsen, and Bauwens 2011).

Finally additionally model evaluations might be necessary for stochastically (e.g. agent based) models, whose output for certain inputs is not deterministic. An unstable numeric scalar output (of a model evaluation) tampers with the result of the sensitivity analysis, which empirically evaluates the effects of input changes. Thus if it is not possible to make the outcome of a stochastically model deterministic (e.g. by fixing the seed values of it's random number

generators), another solution is the calculation of the mean output from multiple evaluations of the same input sample (Prestes García and Rodríguez-Patón 2016).

### 2.5.6. Frameworks and libraries

Prestes García and Rodríguez-Patón (2016) describe a sensitivity analysis framework based on the Repast Symphony (North et al. 2013), which is a toolset for agent based modeling (thus similar to MARS regarding its use case). Beside Repast Symphony the sensitivity analysis framework bases on the programming language R[5]. Basically Prestes García and Rodríguez-Patón (2016) provide an integration broker, which runs inside the same process as the controlled Repast Symphony instance. The broker communicates with an engine, which runs in the R process. On top of the engine Prestes García and Rodríguez-Patón (2016) build R application programming interfaces (APIs) to control *Repast Symphony* simulations. Further Prestes García and Rodríguez-Patón (2016) implemented a set of sampling and postprocessing methods to do elementary effects and variance based sensitivity analyses. Finally their highest abstraction layer allow to specify a sensitivity analysis basically in one method call, whose parameters specify the number of samples and other sampling parameters (e.g. grid cell size for elementary effect based methods). The user passes a definition of the input parameters (being subject of the analysis) and their boundaries. High order functions translate between the input parameter's values and the model's input and define the scalar output. Prestes García and Rodríguez-Patón (2016) apply their framework on multiple models, including a *Repast Symphony* example, which is an agent based implementation of the popular predator / prey model.

Wu, Mortveit, and Gupta (2017) introduces a distributed system to support the validation of network based models. Beside other (validation) model analysis methods they use the system for sensitivity analyses. Similar to MARS the system stores models, metainformation about them (e.g. their parameters) and evaluation results into databases. The user provides wrapper code along with the models, which translates input data from Wu, Mortveit, and Gupta (2017) system's format into the form required by the model. Same applies to the results. A difference to Prestes García and Rodríguez-Patón (2016) is, that the model wrapper also handles the execution of a model evaluation. Thus it would theoretically be possible to combine Wu, Mortveit, and Gupta (2017)'s system with MARS models. But Wu, Mortveit, and Gupta (2017) does not provide enough technical information about their implementation for further work

[5] `https://www.r-project.org/`

in this direction. In addition, MARS already provides data integration and result management on its own.

# 3. Extendend basic immune system model implementation with MARS

## 3.1. Architecture

Primary architecture goal for the alternate implementation of the Basic Immune Simulator was to create a extensible and reusable solution. Thus many generalized and configurable components are the base for the implementation. Also MARS layer and agent structure concepts are used. This section introduces the resulting model design.

### 3.1.1. Parameter generation

Folcik, An, and Orosz (2007) calculate the initial agent distributions within the RepastJ model at the start of every simulation. But the MARS implementation separates this step from the MARS model binary to allow data integration (e.g. of antigen information) in the future. Thus an external tool (*Parameter Preparer*) generates multiple data tables. These tables are imported by MARS. Further MARS' agent initialization creates the agents based on the data (sec. 2.4.1).

*Parameter Preparer's* input file is a map based structure. The class diagram fig. 3.1 provides an overview of it. The *MetaGroups* at the highest level are user defined categories (e.g. agents, areas and antigens). Each MetaGroup has a *name* and further maps multiple *AgentGroup* objects by their *name*s. An *AgentGroup* objects can have any number of user defined attributes. Each attribute corresponds to an agent parameter with the two exceptions *_type* and *_number*. While *_number* specifies the amount of data rows (agent number) to be generated, *_type* defines the target data table. The attribute values can be constants or *GeneratorReference* objects. In latter case *type* is the name of a generator known by *Parameter Preparer*, which will calculate (multiple) attribute values based on optional, named *parameters*.

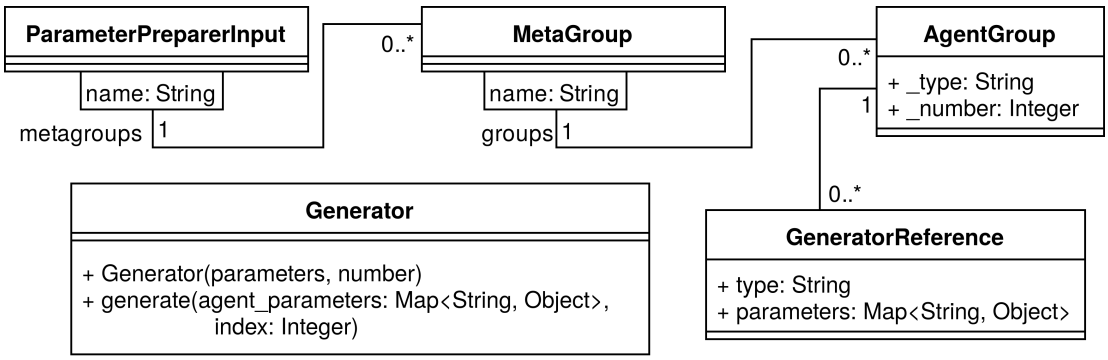


Figure 3.1.: Basic structure of *Parameter Preparers* input and the Generator base class.

While execution *Parameter Preparer* creates a *Generator* object for every *GeneratorReference*. It passes *parameters* (from *GeneratorReference*) and the *number* of values to be calculated as constructor arguments. Then *Parameter Preparer* calls the *generate* method of every *Generator* related to a *AgentGroup* *_number* of times. The argument *agent_parameters* are the constant attribute values of the *AgentGroup* and previously generated agent parameters. Meanwhile *index* is the ordinal number (between *0* and *_number*) of the *generate* call.

### 3.1.2. Components of the model binary

All components shown in figure fig. 3.2 except *CommonAgentTools* provide a layer, which each implements an interface used by the other layers and the agents.

The remainder of the section describes each layer more detailed:

- *CommonAgentTools* is a collection of service classes used by the agents or to manage them. All classes are so abstract, that they might be of interest to other domains and are staged for a upstream process into the MARS framework. For example the component contains helper classes to realize state machines and timer units.
- *AreaLayer*: At the time of writing no MARS' spatial management component exist, which can be distributed over multiple calculation nodes. Further the spatial interfaces provided by the MARS framework lack the concept of subareas, which are needed to realize the zones (see section sec. 2.3). Thus each agent type and zone combination uses its own instance of a MARS spatial management component. Each instance is wrapped by an agent hosted by *AreaLayer*. The layer maps zone names and agent types on the

global identifiers of the agents and creates the adapter agents on demand. For later purpose it also manages the dimension of each zone / area.

- *AntigenInformationLayer* isolates the domain specific representation of molecular structures, which are the base of individual antigens, antibodies and matching cell receptors. The layer maps identification strings on molecular structure objects. Those offer a simple interface, which calculates the binding probability between two molecular structures.

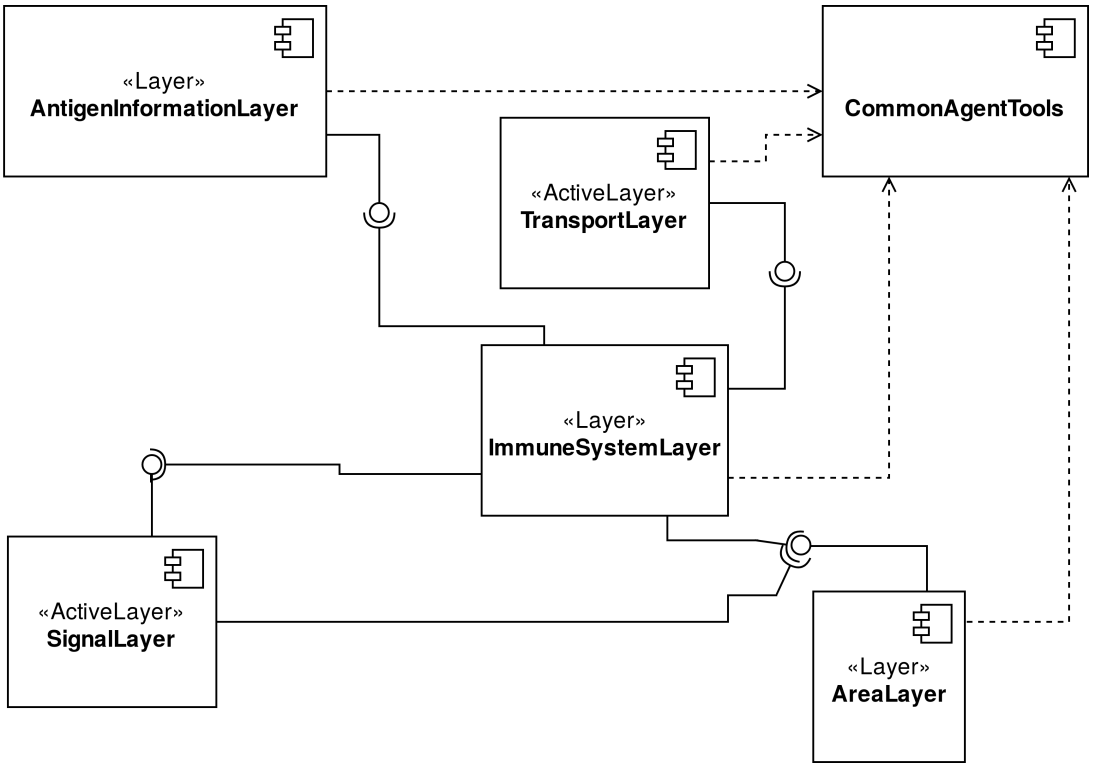


Figure 3.2.: Components of the Basic Immune Simulator implementation with MARS.

- The *SignalLayer* component is responsible for the management for the signal values. The active layer collects all changes of each signal value for each grid cell while a simulation tick. After the tick it then applies the changes and calculates the diffusion. Of course the layer's interface also allows agents to retrieve the current signal values.
- As a part of the portal (see section sec. 2.3) implementation the *TransportLayer* allows output portal agents to register information objects paired with their global identifier. Input portals register transported agents by their global identifier and an exit filter. The filter matches portal information objects with specific attribute values. After each ticket

the logic of the *TransportLayer* distributes all registered agents on matching output portals. Finally in the following tick the output portals can query their attached agents and process them further.

- The *ImmuneSystemLayer* finally contains most of the agents, which are further described in the next subsection. The layer also contains the input data based initialization of the agents using the *MARS AgentManager*. Further the layer manages initial parameter sets used for reproduction of agents. Each parameter set has a specific name (e.g. inflammatory TH-Cell). In case of reproduction an agent fetches a parameter set and modifies it depending on its own state (e.g. position).

### 3.1.3. AntigenInformationLayer: Basic support to map antigen specifity

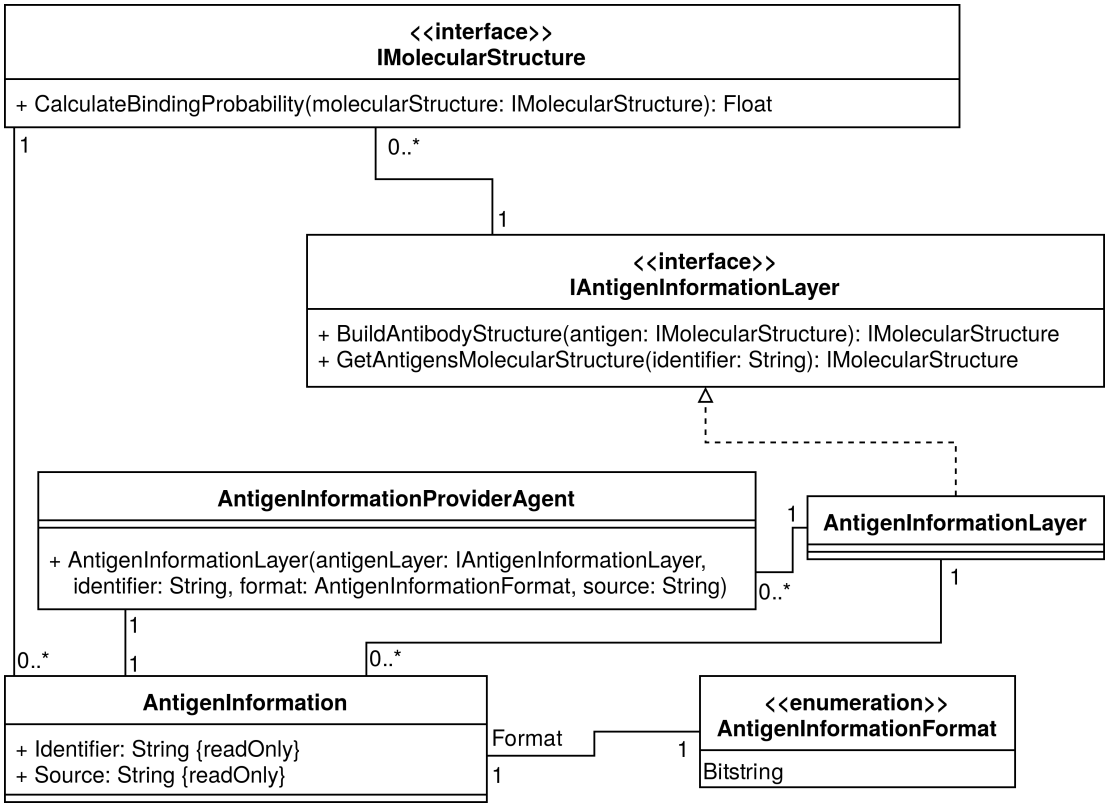


Figure 3.3.: Overview about the public parts of the AntigenInformationLayer.

The class diagram fig. 3.3 gives an overview about the public parts of the *AntigenInformationLayer*. Each cell receptor, *Antibody* and *Virus* in the model refers a *IMolecularStructure* instance, which represents an antigen itself or the structures to detect a specific antigen. Each *IMolecularStructure* provides a method to *calculate* the *binding probability* with another *IMolecularStructure*. A real number between 0 and 1 expresses the probability.

An *IMolecularStructure* can derive from another *IMolecularStructure*, if the implementation of the B-cells calculates the matching information for an *antigen* using *BuildAntibodyStructure* from the *IAntigenInformationLayer* layer. Meanwhile the initialization of receptors and virus bases on *AntigenInformation* objects. Each of them has an *identifier*, which the agents use to query it using *GetAntigensMolecularStructure*. *AntigenInformation*'s *format* determines the *IMolecularStructure* implementation and *source* contains initialization information. At the time of writing, the only available *IMolecularStructure* implementation bases on *bitstrings* of length 64, which are defined by *source* as hexadecimal string with the prefix *0x*. The binding probability is $\frac{64-n}{64}$, where $n$ is the number of different bits between two bitstrings.

Since MARS lacks a generic feature to map data tables directly on layers (and a identifier on information mapping is not a time series), *AntigenInformationLayer* uses *AntigenInformationProviderAgent* (agents) for an initialization process as described in section sec. 2.4.1.

### 3.1.4. AreaLayer: Distributable implementation of the zones

Hüning (2016) proposes the use of multiple MARS Environment Service Component (ESC) instances for position management in distributed simulations. He distributes agents of different type on separate layers, which each has their own spatial environment. Agents of one layer query the positions of agents of another layer using those interface (and proxy object). Meanwhile the Basic Immune Simulator Model has nine agent types with two additional ones introduced by the author (of this thesis, sec. 3.1.7.1), of which several can occur on all three zones. Each zone concludes in a MARS ESC instance. Thus using exactly Hüning (2016)'s approach would lead to over 30 layers. Since layers and references between them must be defined statically in MARS, a high number of layers is hard to maintain and thus *AreaLayer* uses agents instead of layers to wrap the single MARS ESC instances.

An agent queries a MARS ESC wrapper agent's identifier by calling *GetAreaID* of *AreaLayer*'s interface *IAreaLayer* (see diagram fig. 3.4). The agent passes the target zone's (*area*'s) *name* and the target agent type to the method. If it does not already exist, *AreaLayer* will instantiate

an MARS ESC and the wrapper agent. Anyway *GetAreaID* returns the global identifier of the wrapper agent, which the calling agent further uses to retrieve the MARS agent shadow (proxy) of the wrapper agent. Beside the wrapper agents *AreaLayer* manages information about the *areas* / zones using *AreaProperties* objects. Each *area* has a *name* and cuboid boundaries (defined by *width*, *height*, *depth*). Further the Basic Immune Simulator's zones are *toroidal* (agents leaving at one boundary enter at the opposite one, see sec. 2.3). The initialization process of the *AreaProperties* work similar to the one for *AntigenInformation* (see sec. 3.1.3) using *AreaPropertiesProviderAgents*, which fig. 3.4 does not show.

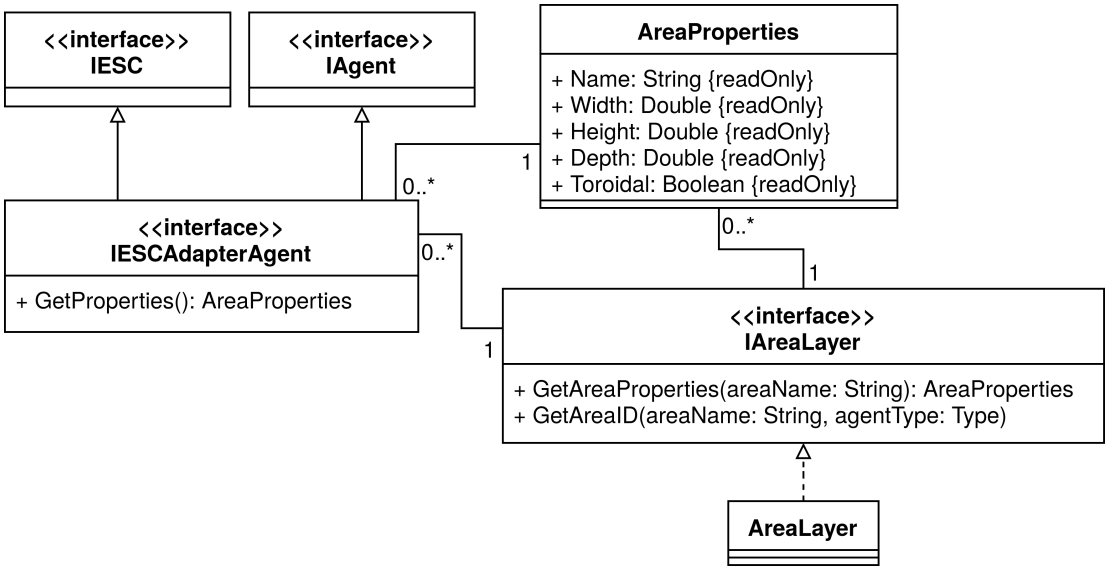


Figure 3.4.: Public classes and interfaces of the *AreaLayer*

### 3.1.5. SignalLayer: Signal modification and retrieval, result output

The *SignalLayer* uses a three dimensional grid for each *signal*, *area* (zone) combination. Each grid rasterizes its *area* with a resolution of *1.0*. Thus every cell has a discrete *Position* with the components *X*, *Y*, *Z* (fig. 3.5). *SignalLayer* retrieves an *area*'s dimensions from the *AreaLayer* (sec. 3.1.4). The creation of a grid happens on demand (with *0.0* in every cell).

As already mentioned by section sec. 3.1.7.2 *Agent* observe signal values by *SignalSensor*s. Each *SignalSensor* is connected to a certain signal *layer* and agent (*owner*) at creation. Further it observers certain signals (specified by constructor argument *scannedSignals*) in a Moore *neighborhood* of certain *size*. Meanwhile a *SignalSensor* gets the center of the neighborhood and

the area to observer from its agent. For that the *Agent* must implement *ISignalConsumer*, which demands *get*ters for the *position* and the *name* of the *area*. The SignalSensor's *Sense* method (called by MARS every simulation step) gathers the signal strengths using *GetSignalStrength* of *ISignalLayer*. Further it creates a *SignalResults* object with the results, which MARS then passes to the agents main logic. The latter can *get* the strength of a certain *signal* (*GetSignal* method) or call *SignalResult*'s *GetStrongestSources* to retrieve the grid cell *Positions* with the strongest signal strength. If the *signal* parameter of latter method specifies multiple signals, *SignalResults* sums their strengths beforehand. Meanwhile if the parameter *withOwnPosition* is true, *SignalSensor* also considers the current cell of the agent.

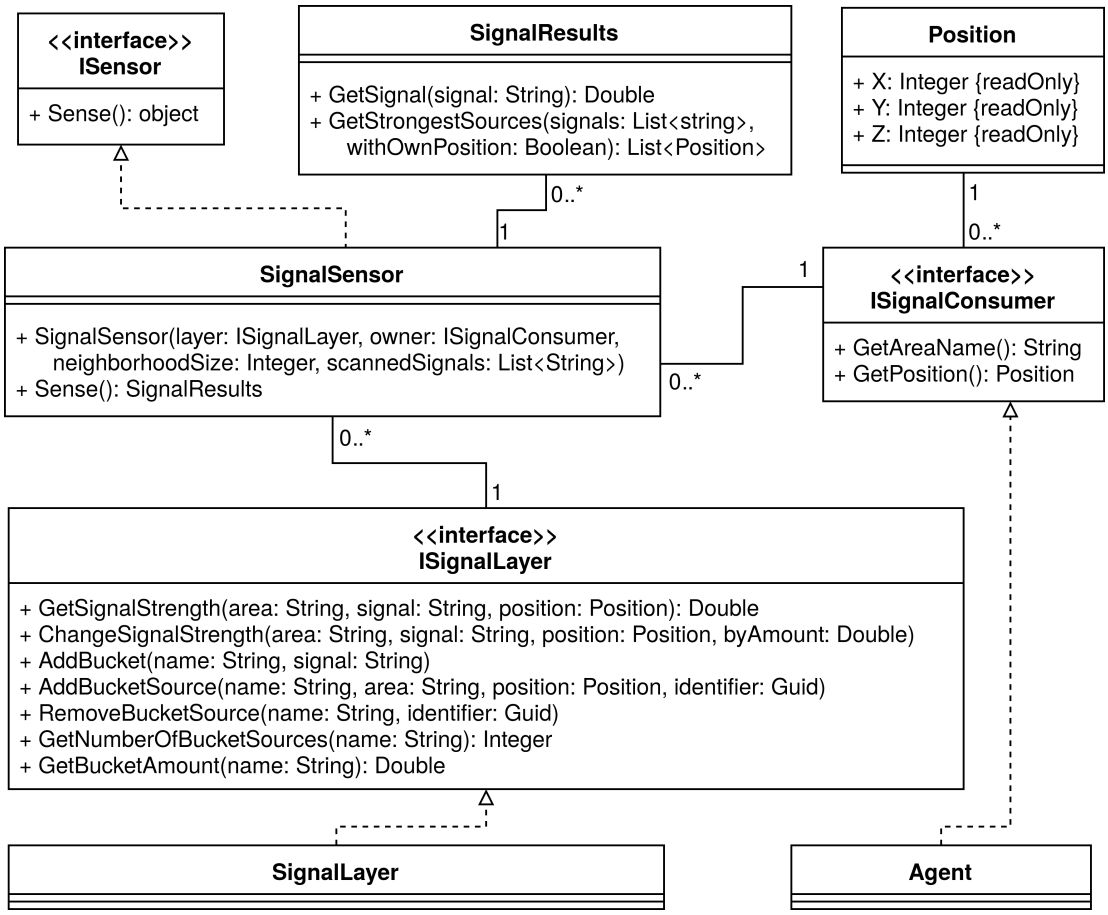


Figure 3.5.: Interfaces and classes of *SignalLayer* and *SignalSensor*.

The *ChangeSignalStrength* method of *ISignalLayer* allows agents to modify the strength of a *signal* in an *area* at a *position by* a certain *amount*. *SignalLayer* does not apply the change

immediately. Instead it adds the sum of all changes to a grid cell's value at the end of each simulation step right after calculating the diffusion of signals. That way all agents sense the same signal strengths on a specific position independent from the processing order of their sensors, reasoning and actions. Meanhwile *SignalLayer* uses the same diffusion formula as Folcik, An, and Orosz (2007). Further signal diffusion on border cells is influenced by cells at the opposite border to map the toroidal zones also for signals.

Finally *SignalLayer* offers the bucket system, which support the MARS Basic Immune Simulator implementation's portals in their task to synchronize signals between zones. Each bucket sums the signal strengths of a set of positions (*sources*, practically portals) in different *area* / zones. *ISignalLayer*'s *AddBucket* method adds a bucket with a certain *name* for a *signal*. *AddBucketSource* adds a source with *identifier* at *position* in *area* to the bucket *name*, while *RemoveBucketSource* removes the source. Finally *GetNumberOfBucketSources* count the sources of bucket *name*, while *GetBucketAmount* returns the sum of *signal*'s strength at the source positions. Like the diffusion *SignalLayer* calculates the bucket sums after MARS processed all agents in a simulation step.

### 3.1.6. TransportLayer

While the portal agents use the *SignalLayer*'s bucket system to exchange signal strengths between zones, the *TransportLayer* plays an important role in the transportation of agents between the zones / areas. The latters layer's design aims to be reusable for other models, where several disjointed parts of the environment are spatially pictured and connected in an abstract manner. One example is the model of a building, where the agents map the visitors, the floors relate to the zones and the elevators to the portals.

Each portal implementation provides an *IPortalInformation* (see fig. 3.6) object, which describes transport relevant parts of the portal's state. In the case of the MARS based Basic Immune Simulator implementation for example agents decide to end their passages at a certain portal, if a certain signal is present near the portal (*SeeSignal* method of *BasicImmunePortalInformation*, fig. 3.6). More general the *IsMatch* method of an *IGenericPortalFilter* object decides if an *IPortalInformation* matches and thus the travel of an agent (related to the filter object) can end at the portal agent (related to the information object).

At the start of its life a portal agent introduces itself to a *TransportLayer* by calling *RegisterPortal* of the *IGenericTransportLayer* interface. The portal passes its *identifier* and initial *information*

about itself. As soon the *information* changes, the portal calls *UpdatePortalInformation*. Further the portal calls *GetPassengers* to retrieve information (global identifier, type) of travelling agents, which can enter an area through the portal. As soon the portal decides to let an agent pass into the area, it calls *TryEndTransport*. If latter method returns *False*, the agent already ended its passage through another portal. Respecting this information ensures thread safety. Meanwhile a call of *StartTransport* marks the start of a travel. It provides the distributable information about the related *agent* and the *portal filter* to the *TransportLayer* implementation.

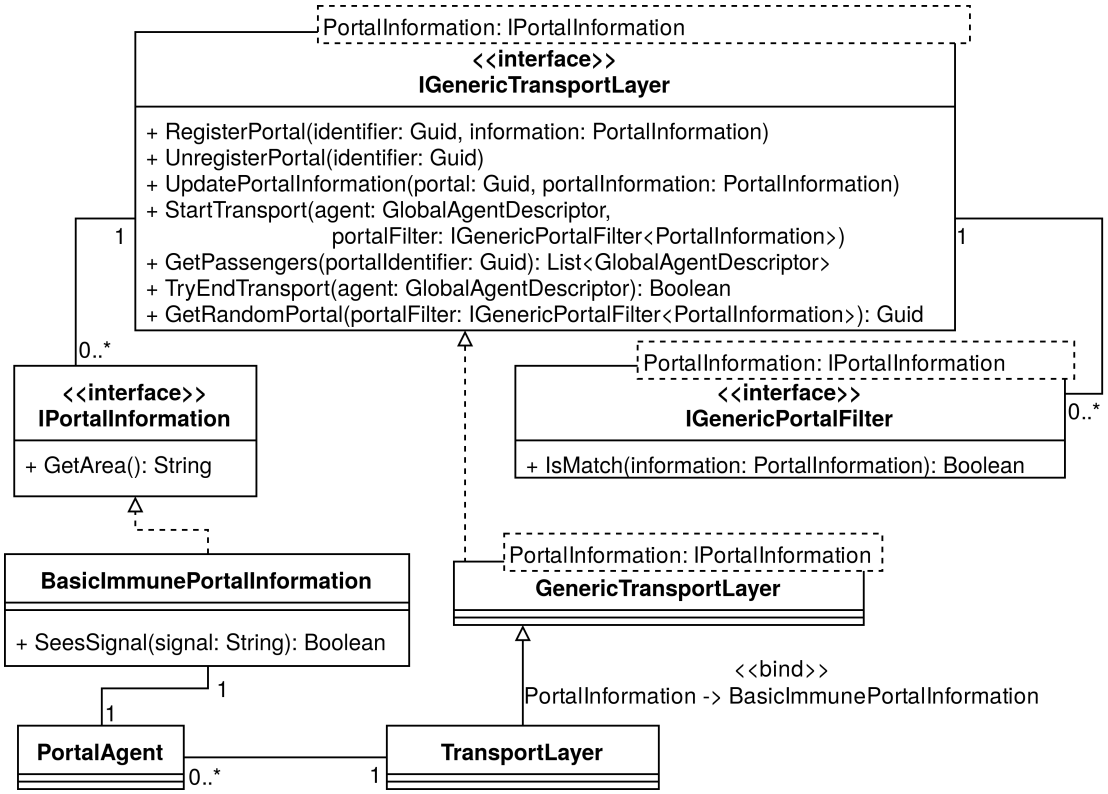


Figure 3.6.: Public interfaces of the *TransportLayer* and their relation to the Basic Immune Simulator *PortalAgent*.

The filter objects does (not) match a portal agent directly, because in this case the *TransportLayer* would have to query the agent shadow of the portal from MARS. At the time of writing this requires the concrete class, which implements the portal. In conclusion the *TransportLayer* component would have a circular dependency with the component, which provides the portal implementation.

### 3.1.7. ImmuneSystemLayer and the agents

#### 3.1.7.1. Agents and their interactions

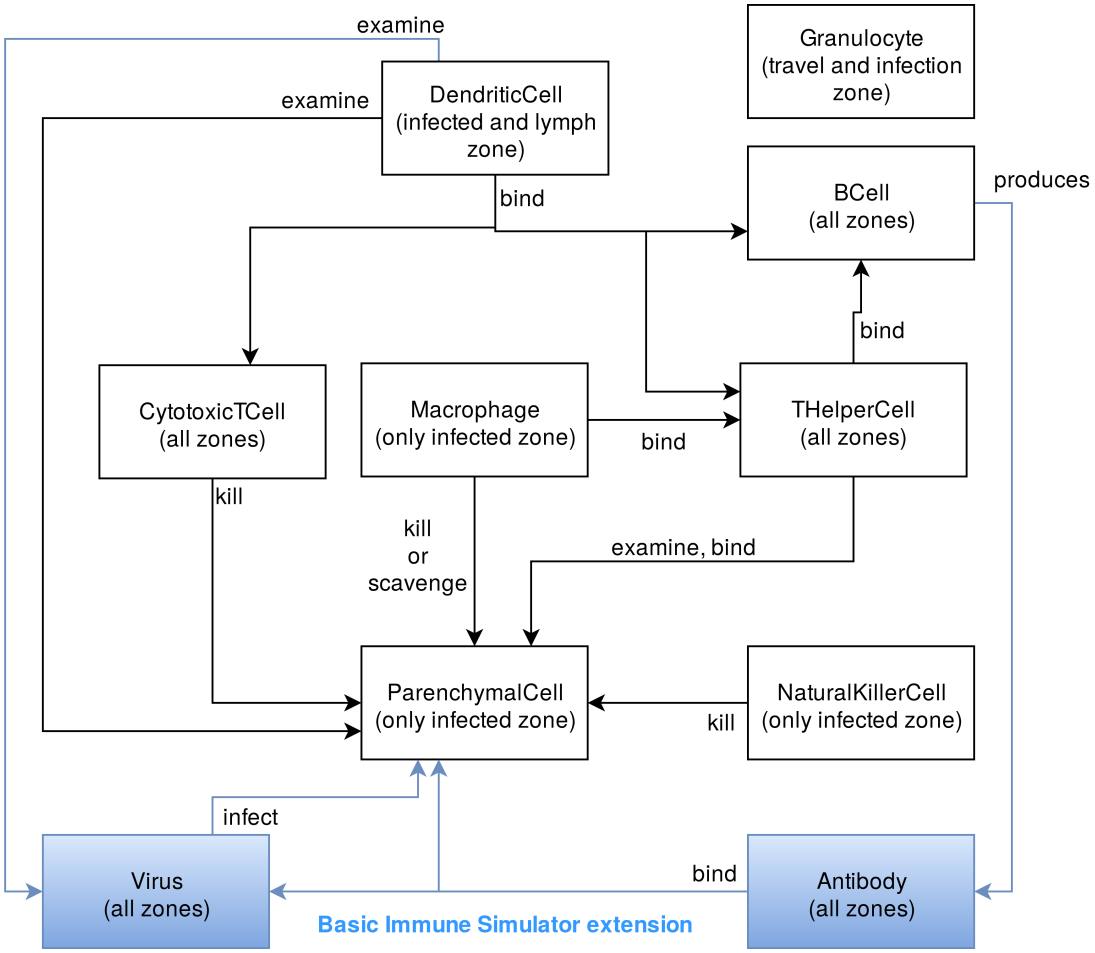


Figure 3.7.: Overview of agent types (except portal agent) of *ImmuneSystemLayer* and their interactions.

Figure fig. 3.7 shows almost all (except the portals) agent types, their interaction (classes) and in which zones they can be found.

The agent types with white background are adopted by the Basic Immune Simulator's model. Meanwhile *Virus* and *Antibody* are additions by the author to prepare the alternate model implementation for scenarios, where the antigen specificity and high mutation of virus play

an important role (e.g. HIV (Neumann 2008)). The movement and life of *Virus* and *Antibody* reflect the diffusion of the related signals in the original implementation of the Basic Immune Simulator. *DendriticCell* agents and infected *ParenchymalCell* agents present the antigen (molecular structure mapping) of the virus to other agents. *T-Cell* agent and other agents representing immune system cells have receptors, which bind the antigen by some probability (see also *AntigenInformationLayer* in fig. 3.2). *B-Cell* agents produce *Antibody* agents, which match the antigen previous overtaken from a *DendriticCell* agent. Finally *Virus* agents infect *ParenchymalCell* agents and reproduce themselves afterwards.

Definitions of two classes of interactions between agents exist:

- An *examination* only affects the agent doing it on another agent and the other agent does not influence it. The interactions take place while the sensing or reasoning of an agent.
- A *binding* between two agents needs the successful outcome of the stochastic experiment if two molecular representations match. It further involves a handshake process, which allows both agents to influence the *binding* success. Finally a complete *binding* influences the state of both agents.

*Makrophage* and *NaturalKillerCell kill* or *scavenge* other agents with a constant probability. Beside that these interactions are *binding* interactions.

In each interaction one agent plays the *active* part, when the processing of his actions initiated the interaction. The other agent has the *passive* part, where a part of its logic executes independently from the main logic of the agent.

#### 3.1.7.2. Internal structure of agents and their lifecycle

Figure fig. 3.8 gives an overview of common internal parts of an agent and how the agent is embedded into its environment of layers and MARS components. The remainder of the subsection describes the agent's part with more details.

Each agent inherits from a *Dalski SpatialAgent* of the MARS framework. This basic agent implementation manages the *sensors*, fetches and stores their results once in every ticket. Further it executes the agent's reasoning. The model's *neighbor sensors* utilize *AreaLayer* and MARS' environment service component (*PositionManagement*) to find nearby other agents.

Further a *neighbor sensor* creates an agent stub on demand using *MARS AgentShadowingService* (see section sec. 2.4.3). Meanwhile the *SignalSensors* query the *SignalLayer* for the signal values at the current agent's position. After the *sensing* phase finished, the *SpatialAgent* executes the *reasoning*.

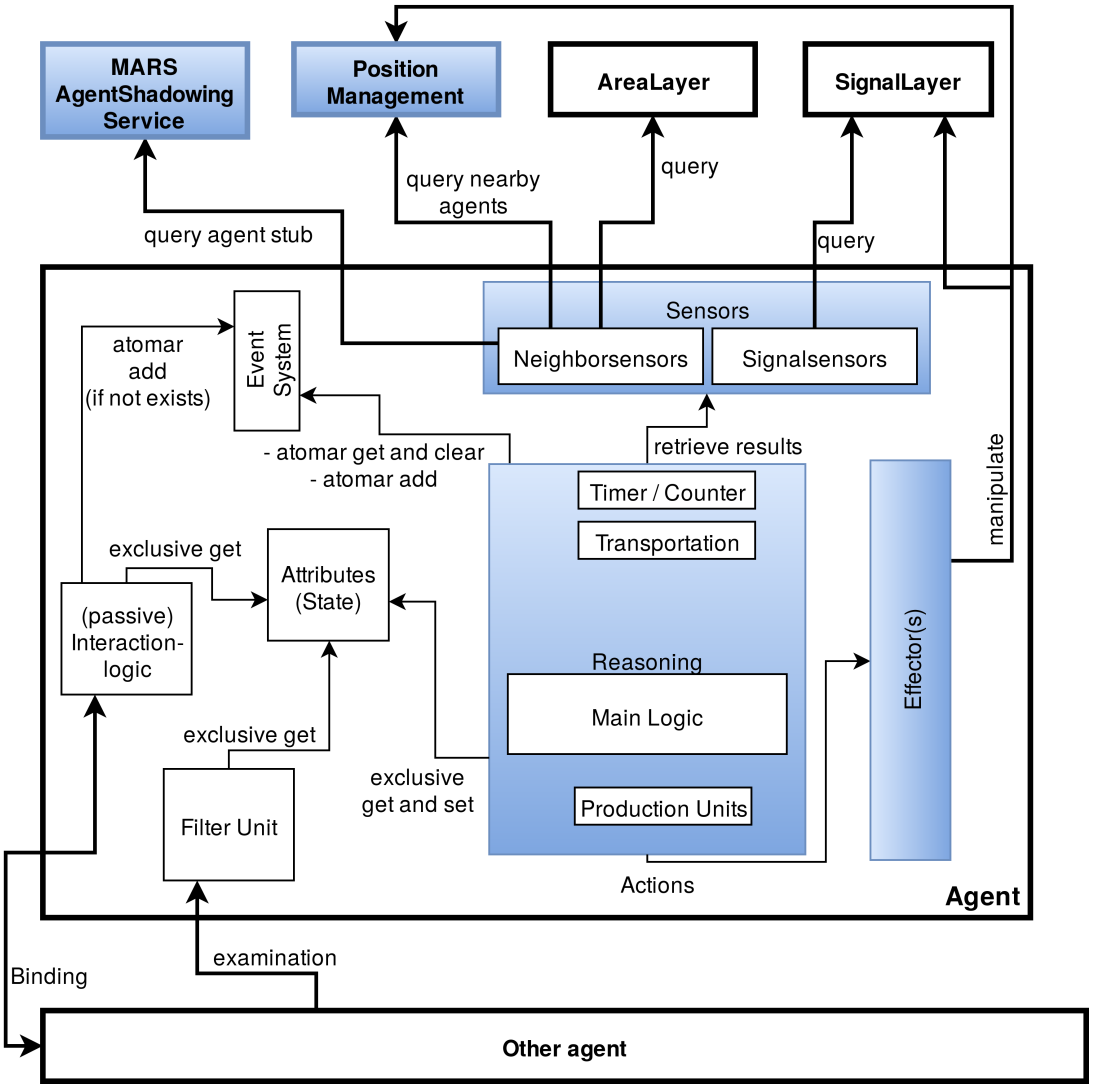


Figure 3.8.: Overview of agent's internal structure and the interaction of the units with each other and other agents / layers. Parts which are mainly part of the MARS framework are highlighted blue.

There the agent's *main logic* makes decisions. For that it is supported by generic logic modules,

which for example measure *time* or *count*. Also units exist to support common *transportation* (switches between zones) or *production* (of signals or agents) choices. *Reasoning* units may retrieve *sensor* results and produce *action* objects. The *SpatialAgent's effectors* executes those after the *reasoning* phase. The actions then submit changes to layers (e.g. *SignalLayer*) or the *position management* (for movement).

At the end of the *reasoning* the *main logic* can change the state and *attributes*. Each agent has several *Attributes*. Some are state independent. In contrast others are inferred directly from the current state of the agent. Since the MARS framework technically processes the *sensing*, *reasoning* and *effectors* of multiple agents parallel, it can happen that one agent changes its *attributes*, while another agent does an *examination* or *binding* interaction on it.

To prevent an inconsistent view on the attribute set, the *main logic*, the *passive interaction logics* and *filter unit* ensure exclusive access to the *attributes*. The *filter unit* is the author's preferred way for an agent to get a consistent way on multiple attributes of another agent. One agent provides a filter object to the *filter unit*, which matches several attribute values. Further the *filter unit* grants exclusive attribute access to the filter object and let it decide if the agent matches certain conditions. This way it is possible that each agent manages the exclusive attribute access completely on its own, which helps to avoid deadlocks.

The *event system* handles the communication between the *interaction* and the *main logic*. It is a high level structure managing the concurrent registration and query of *event* objects, which inform the *main logic* about interactions triggered by other agents. There are interactions, which the *passive partner* only can do once while a simulation step. For example a cell can only be killed once. Thus the *event system* accept the report of certain *events* only once (until the *main logic* clears all events).

#### 3.1.7.3. Agent class hierarchy

While section sec. 3.1.7.1 gives an overview about the agents and their interactions, diagram fig. 3.9 introduces the actual class hierarchy behind the agents. The colour marked interfaces and classes are provided by MARS. For clarity the figure does not show several classes, interfaces and no details (e.g. methods, attributes).

Each agent has one concrete class (e.g. *DendriticCell*), which implements agent specific parts of the units introduced by section sec. 3.1.7.2. Especially a concrete agent class implements the

*MainLogic*. In case of the cell agent and the portals the later derives from the state machines, which (Folcik, An, and Orosz 2007) present in the appendix of their publication. Meanwhile the abstract parent classes (*BodyCell*, *BodyParticle*, ...) implement those parts, which are specific to the MARS implementation of the Basic Immune Simulator (e.g. the *Filter Unit*) or which multiple agents use. Logic for handling the agent's position or an zone / area change is an example for a common agent part.

Further each agent has a specific public interface (e.g. *IDendriticCell*), which extends more generic ones. Other model parts and MARS interact with an agent implementation only over its interfaces. Each agent interface encapsulate its initial parameters into a serializable and immutable object. The agent exposes the parameter object using its *I*Parameter* interface (e.g. *IDendriticCellParameter*). A concrete agent class inherits parameters required by its base classes. For example *BodyParticle* requires an initial position, area name, *SignalLayer* reference and the *AreaLayer* to implement common agent logic. Only *Portal*s (not shown in fig. 3.9) references the *TransportLayer*, since one of their specific tasks is the transportation of other agents between areas.

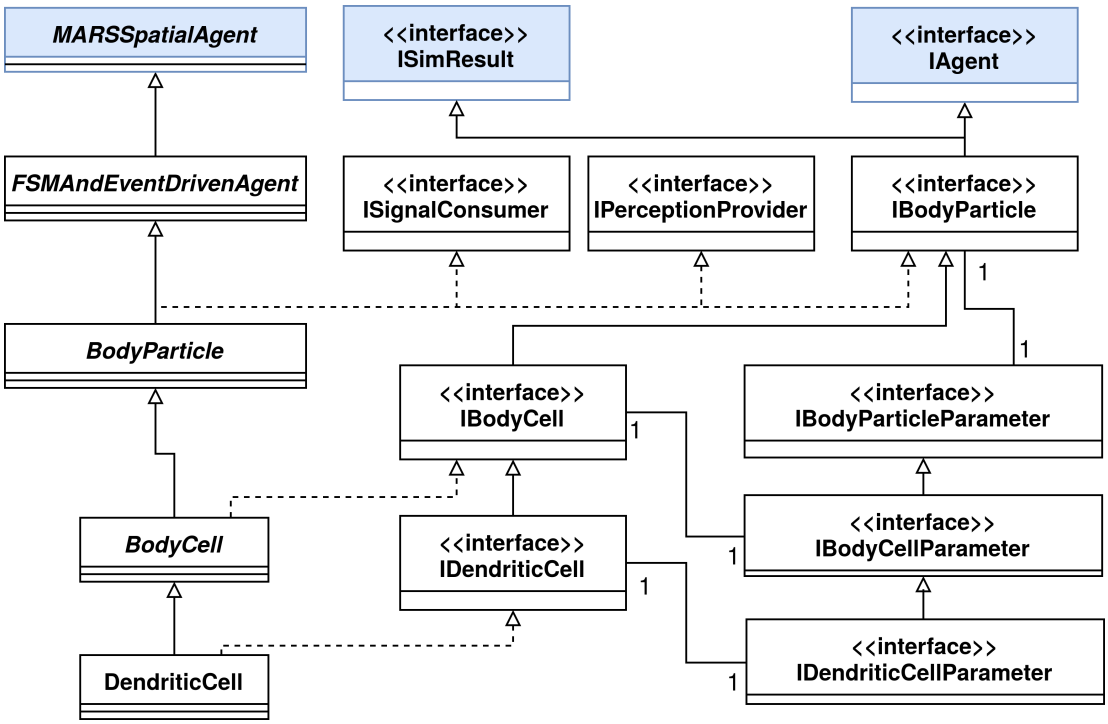


Figure 3.9.: Class diagram shows interfaces and classes of the agent implementations

All concrete classes of the agents introduced by sec. 3.1.7.1 inherit from *BodyCell*, which itself inherits from *BodyParticle* and basically adds logic to represent infection by a *Virus*. Exceptions are *Virus*, *Antibody* and *Portal*, which inherit from *BodyParticle* directly.

#### 3.1.7.4. Parameter groups and (re-)production of new agents

Section sec. 3.1.1 describes the generation of the table files, which contain the agents' parameter data. It introduces parameter groups of agents, where multiple groups base the same type (e.g. *DendriticCell*), but have slightly different initial properties (e.g. non-/inflammatory). Further an agent take all parameters from its group except a few, especially spatial ones. The *ImmuneSystemLayer* offers methods, which support agents to dynamically create new agent instances based on a certain parameter group (e.g. inflammatory *DendriticCells*). For that *ImmuneSystemLayer* saves one concrete parameter object (see also sec. 3.1.7.3) per group. Then an agent can query a copy of the object using the group name (e.g. *InflammatoryDC*) and adjust some parameters (e.g. position, area). Further it initializes an instance of the corresponding agent class by calling one of the class' constructors on the parameter object. Finally the *BodyParticle* constructor of the new agent register the agent at the MARS' simulation runtime. Reproduction of an agent works similar based on the agent's own parameter object.

#### 3.1.7.5. Positioning, moving and neighbour sensing

MARS only offered a continuous spatial position management, when *BodyParticle* was designed. Thus *BodyParticle* maps the grid of a Basic Immune Simulator zone on the continuous space. According to the signal layer (sec. 3.1.5) each grid cell is a cube with a side length of one. Thus the current home cell of an agent is calculated by rounding down its current coordinates. At each simulation step *BodyParticle* changes its position by a vector $v = (x, y, z)$, where *x*, *y*, *z* have each a value of *-1*, *0* or *1*. The spatially static agents, *Portal* and *ParenchymalCell*, override this behaviour. Otherwise child classes of *BodyParticle* choose between a completly random movement vector and a movement vector, which directs to a neighboured grid cell with the heighest values of certain signal strengths (see also sec. 3.1.5).

Meanwhile a *Neighborsensor* (see sec. 3.1.7.2) scan a cuboid part of its agent's current area for other agents. The cube's center equals the center of the agent's current grid cell. Meanwhile the cube has a side length of three, which corresponds to a Moore neighbourhood.

### 3.1.8. Result output and analysis

Basically the *SignalLayer* uses special service agents to output the strengths of each signal / area combination as three-dimensional matrices at every simulation step. Further several properties of the agents, including their position, are recorded using MARS legacy output mode (sec. 2.4.1).

After a simulation run an external, model specific evaluation tool generates additional matrices by counting the agents in each grid cell of an area. For each matrix a certain filter expression defines the agents to be counted. Thus each matrix element specifies the number of matching agents in a grid cell of a certain zone and at a simulation step. Additionally the evaluation tool counts the agent matching a filter independent from the area and position.

In a next step the evaluation tool reduces the number of dimensions of the matrices to two by aggregating counts on different planes. Finally the tool generates a report with following elements:

- A line plot shows the global counts for every agent filter over the simulation steps.
- A heatmap visualizes the agent count / signal strength in the different parts of an area.
- An index document references all images and contains information about the filters.

A previous work[1] of this thesis' author describes the result evaluation tool in more detail.

## 3.2. Implementation state and details

A feature complete implementation exists of a all described layers. Further all of the common agent tools are realized. Meanwhile the agent logic derived from the original Basic Immune Simulator is implemented as described by the attachments of Folcik, An, and Orosz (2007). But the implementation still lacks several details, which only Folcik, An, and Orosz (2007)'s sources describe. Further it contains functional bugs. The C# sources of the actual model can be found in the directory *Code* of the repository[2]. The directory *is_tool* hosts the analysis tool's Python sources and *parameter_preparer* the Python implementation of the parameter generation. *Code/Tests* provides hundreds of unit tests, which especially check the agents.

[1] `http://lukas-grundmann.de/pubs/PO2.pdf`
[2] `https://gitlab.informatik.haw-hamburg.de/mars/model-immune-system`

# 4. Global sensitivity analysis framework for MARS

## 4.1. Requirements

Chapter sec. 2.5 gives an overview about available global sensitive methods. There are many of them. All have their advantages and disadvantages depending on the actual research problem and model to evaluate. Some methods might be suitable to support early decisions for further research, but might not be able to generate the final results as required. Another aspect is, that there are diverse possibilities how to process the results from the multiple simulation runs. Because of the resulting variability of possible usage scenarios and sensitivity analysis setups, a toolset should be designed flexible to support most of them. In this aspect the setup of a sensitivity analysis is similar to defining a model. For the latter high level programing languages have proven to be a flexible and powerful enough tool. Thus they should also fit in the use case of setting up a sensitivity analysis experiment. To support this approach MARS must provide a programing interface to allow the setup and start of model evaluations (simulations).

One interface is already provided by the time of writing through the TeachingAPI service. But it does not hide several technical aspects and malformed requests are often just rejected by the MARS backend services. For the end users the graphical interface abstracts the technical details. Since not every researcher should have to know those, a system is required, which abstracts certain aspects about interfaces and protocols used between the MARS services. It should reflect the user interface workflow as much as possible.

The *MARS Teaching User Interface (UI)* is a helpful tool for end users, especially to design the mapping from data on model parts, since it can visualize the complex relationships better than a textual description / programing language can do. On the other hand the user interface is

not suitable to create thousand of simulations with different parameter sets. For combining the advantages of both worlds, the system for sensitivity analysis with MARS should allow to retrieve mapping information and result configurations in a reusable and adaptable shape.

## 4.2. Architecture

### 4.2.1. Overview of high level components

Graphic fig. 4.1 gives an overview of the sensitivity analysis framework for MARS. It shows the framework's high level components and how they interact with each other. Further it visualizes the components relation to the researcher / end user of MARS. Component designed and developed mainly by other MARS developers have the prefix "MARS" in their name. The author developed the *Result Query Service* and the *Automatisation Service* while a study project. The *Client Library* and *Command Line Interface (CLI)* are subjects of the development part of this thesis. Thus this section covers latter two components in detail, while it only present aspects of the other components, which are important for the interactions.

The user needs to *upload files* (e.g. model binaries, data tables … see also sec. 2.4.2) only once, which does not depend on the input parameters generated by the sensitivity analysis sampling. Further a preliminary mapping can be created. If the mapping references data influenced by input parameters, it requires adaption every simulation (evaluation of the model) run. But the sensitivity analysis setup can reuse parts of the mapping like the column name to agent parameter relations or constant file references (*basic mapping*). Each model binary subject to the sensitivity analysis needs its own *result output configuration* entity. Thus if one input factor chooses model variants and the researcher decides to create an own binary for every of those, multiple *result output configurations* will be the result. For all of the previously descibed operations the user can choose between use of the *MARS Teaching User Interface (UI)*, the *Command Line Interface* or the *Client Library*.

The user has to define the actual sensitivity analysis over the *Experimental Setup*. This includes the used method implementations for sampling and post processing of the scalar outputs. The relations between the sampled parameters and the input files must be defined. Same applies for the aggregation of the model's output to at least one scalar value. Basical the *Experimental Setup* is a program, which contains the algorithms for data generation and data aggregation.

It can implement the sampling and postprocessing directly or use *external sensitivity tools (e.g. libraries)*.

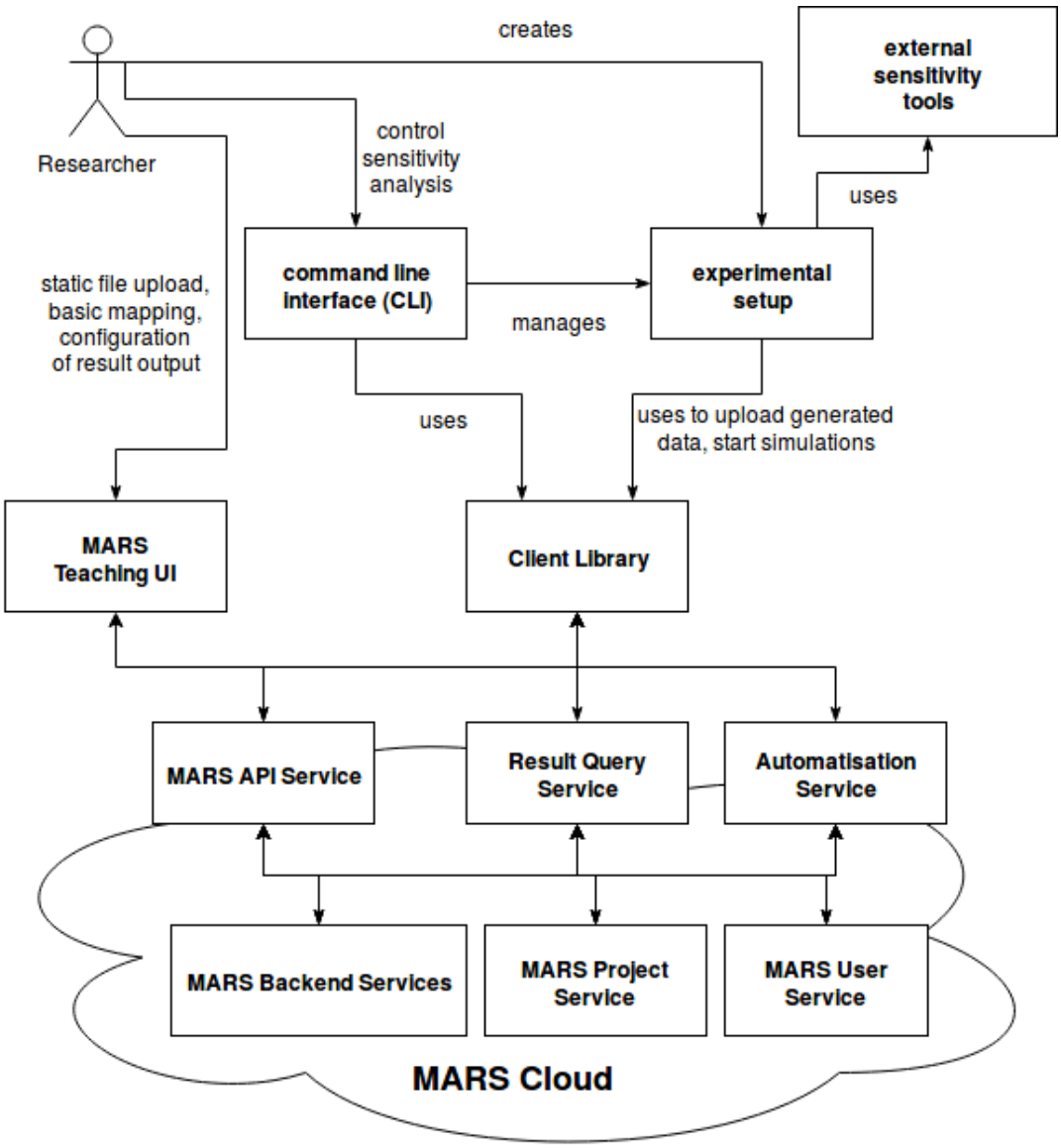


Figure 4.1.: Overview about the high level components of the GSA framework around MARS

The *experimental setup* uses the *Client Library* to *upload generated (input) data*, *start simulations* and query the results to calculate the scalar outputs. For querying the results, the *Client*

*Library* communicates with the *Result Query Service*, which restricts the queries to the result database in the backend based on the client user's permissions. *Client Library* has two options for submitting input data and metainformation to MARS at the moment of writing. First is the *MARS (Teaching) API Service*, which exposes several operations on MARS resource types (e.g. update scenario). It checks the permissions, like the *MARS Automatisation Service* does. The latter has two endpoints. One receives user information and an (compressed) archive with the data. Along with the input files there is a textual, declarative description (manifest / resource definitions) of the MARS resources to create and how they depend on each other. The *MARS Automatisation Service* processes the tasks in the correct order and offers state reports, which a client can retrieve using the second endpoint. Finally several instances of different *MARS Backend Services* do the actual work. Two of them, *MARS Project* and *MARS User Service* (shown explictly by graphic fig. 4.1) play an important role for the permission checks. While the *MARS User Service* authenticates a user, the other service provides information about the permissions, which the user has in the targeted project.

### 4.2.2. Client library

This subsection describes the *Client Library* already introduced in the previous section more detailed. It starts with an overview of the components shown in diagram fig. 4.2 and their interfaces. *Tools* includes all functions, which only depend on external libraries or programs and which basically all other components use. At the time of writing one group of functions (exposed by *DataModels*) offers schema based serialization and deserialization of complex objects. They guarantee immediate failing with a clear problem description, if data in an unexpected structure or with invalid property types is received from the MARS cloud. Also with abilities like name translation of properties, the component simplifies adaptions to minor changes of the MARS API. Generally it reduces the code of other components regarding the mapping between objects in the used programming language and data exchanged with the MARS Cloud. Another part of *Tools* wraps the calls for the building of model sources and further packs the resulting binary (if the build was successful) as expected by MARS' import infrastructure. This part provides the interface *BuildModel*.

The component *Client* handles the communication with the MARS backend services over the *MARS (Teaching) API Service* and the *Result Query Service*. Further it represents resources (e.g. scenarios) available in the MARS cloud to *Client Library* users over wrapper objects. The latter's tasks are to hide technical information (e.g. concrete database table names) and to cache

information locally. Further the wrapper objects offer methods, which make certain complex data modifications (e.g. data mapping) more straightforward. All public parts of the wrapper objects define the *MARSResourceWrapper* interface and are further described in a following subsection.

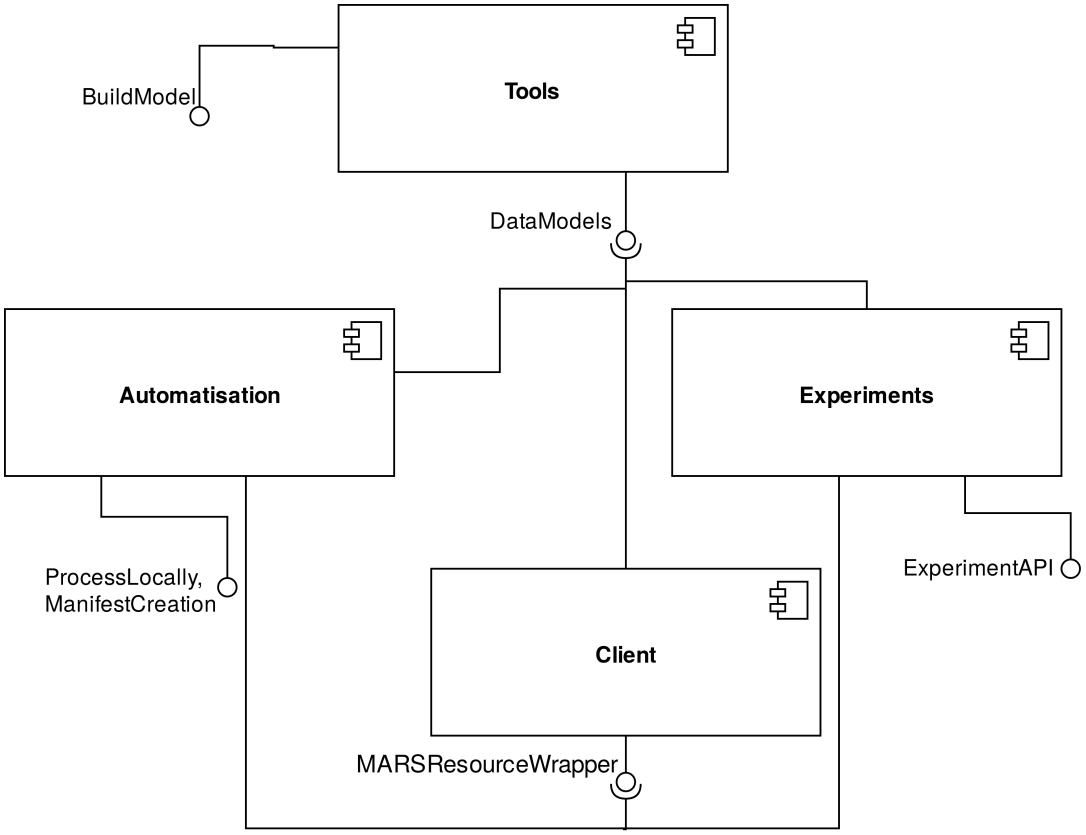


Figure 4.2.: UML component diagram of the Client Library

The *Experiments* component offers a toolset to programmatically declare a model evaluation (or *Experimental Setup* introduced by sec. 4.2.1) as required by sensitivity analysis (see section sec. 2.5). It further contains functions to evaluate an *Experimental Setup* with one or more sets of input parameters. Finally *Automatisation* reimplements the basic functionality (exposed by *ProcessLocally*) of the previously mentioned *Automatisation Service* against the *Client* component and thus the *MARS (Teaching) API Service*. Meanwhile the *Automatisation Service* communicates with the *MARS Backend Services* directly. The *Automatisation* component further lacks the preservation of state in a database (compared to the service) and offers tools to

generate the resource declarations / manifests (interface *ManifestCreation*). While manifests allow to declare resources of all types explicitly, an *Experimental Setup* focus of the relation between a model and scalar input parameters / outputs (as required for a sensitivity analysis). Thus *Experiments* is a high level component compared to *Automatisation*. In the current design both depend on *Client*.

#### 4.2.2.1. Class overview of (component) Client: General resource management

The UML class diagram fig. 4.3 gives an overview about the resource management within the *Client* component. The class *Client* is the entrypoint. It manages the credentials of an user (*name*, *password*) and the Uniform Resource Locator (*url*) of a MARS cloud instance. The *login* method handles the initial authentication using the provided credentials. In case of success the *MARS (Teaching) API Service* will return a *token*, which the *Client* instances use for subsequent authentication processes. After login the caller of *Client* can retrieve accessible *projects* or create a new one (by *create_project*). A wrapper object represents every queried remote resource locally. For each resource type known by MARS, at least one wrapper class exists. Each wrapper object caches the internal representation of the resource. Further relations to other wrapper objects are often cached. Exceptions are: *files*, *scenarios*, *plans*, *runs* of *Project*, *projects* of *Client* and *result_configurations* of *Model*. Thus to get a local metainformation update (e.g. to check if another user changed it), a new wrapper object (for the same resource) can be created calling one of these getter methods. Another option is to call the *refresh* method several of the wrapper classes provide (not shown in the diagram fig. 4.3). Finally several of the *update* methods include another retrieval of the resource's data. Beside methods to create new resources, only the *update* methods trigger an actual update of the resource's data in the MARS cloud. All other attribute setters and high level manipulation methods only work on the locally cached data.

The *Project* class has a method to *start* the *import* of a local file (specified by *path*) of a certain *type* (e.g. MODEL, TABLE_BASED). Other methods *create* a *scenario* (based on a *model*) or a (simulation) plan (based on a *scenario* and a *result configuration*). Metainformation (like a title / name or a description) can be provided by arguments, which diagram fig. 4.3 does not show. Each *File* object represents a dataset uploaded to a *Project* and provides certain metainformation about the dataset over attributes: the *status* like shown in fig. 4.3, the data type and more. *Model* dataset wrapper objects offer a method to *create result* (output) *configurations*, which base on the represented model. Also it offers a getter to retrieve all *result (output) configurations*.

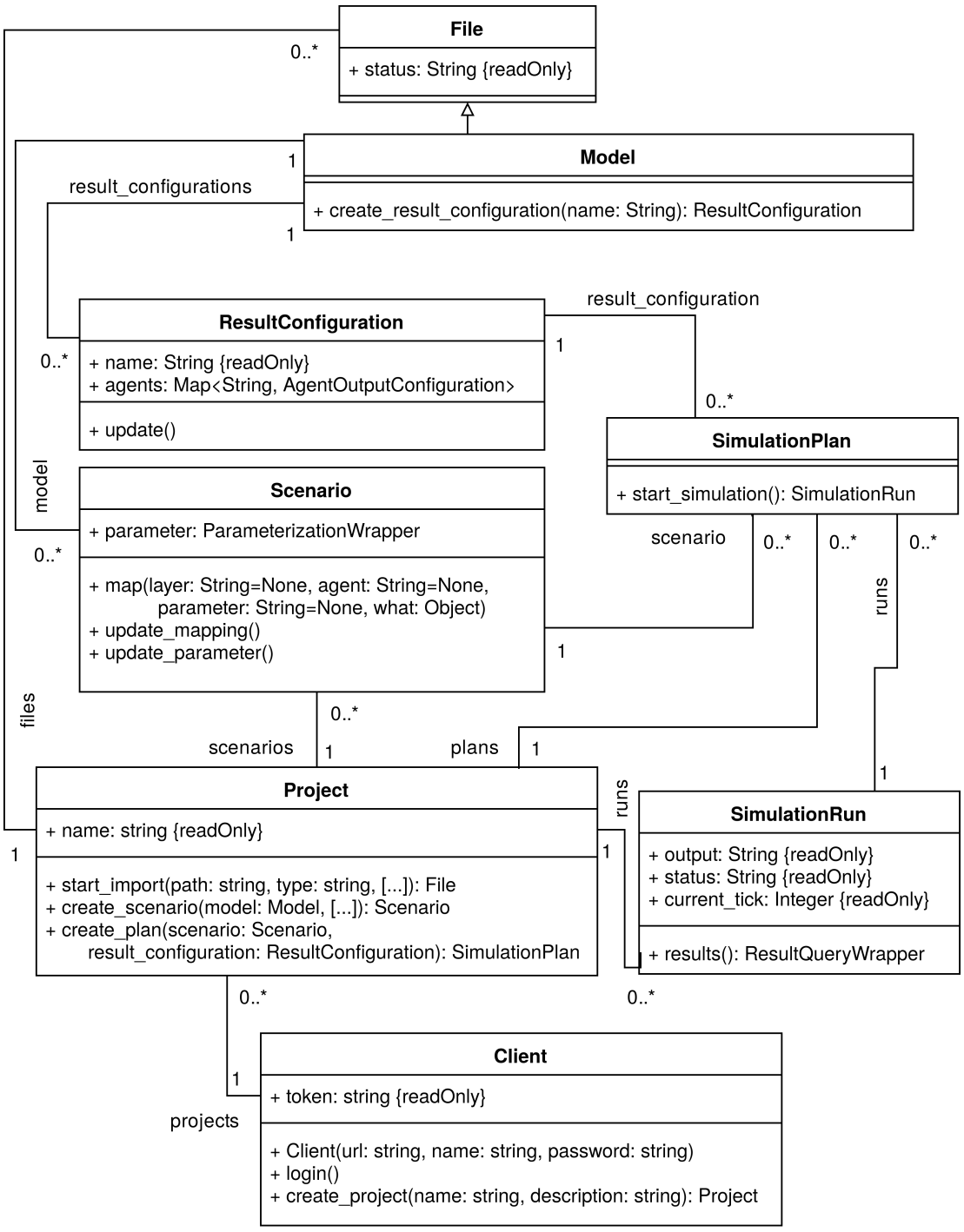


Figure 4.3.: Overview about several classes of the *Client* component. Their public parts make the *MARSResourceMapper* interface (see section sec. 4.2.2). Several attributes and methods are not shown by the diagram to improve clarity.

The *agents* attribute of *ResultConfiguration* exposes the individual settings for every agent. A following subsection describes this feature further. As described earlier in this subsection *update* will send local modifications on the configuration to the MARS cloud. Meanwhile the *Scenario* wrapper class provides the *parameter* attribute to change global scenario configurations (e.g. start date / time of the simulation). Further a method *maps Files* or columns (represented as *File*, column name *Tuple*) on a *layer* or a *parameter* of an *agent*. Another subsection covers all available *map* variants and the setting of scenario *parameters* more detailed. *Parameters* and *mapping* can be *updated* (in the MARS cloud) separately. *SimulationPlan* objects allow to resolve the *scenario* and *result* (output) *configuration*, the related simulation plan bases on. Further they offer a method to *start* a simulation. Finally a *SimulationRun* instance represents an simulation. Among other things a simulation generates logs (*output*) and makes progress (*status*, *current_tick*). Also it produces *results*. The query interface for the latter is described by a following subsection.

#### 4.2.2.2. Class overview of (component) Client: Result output configuration

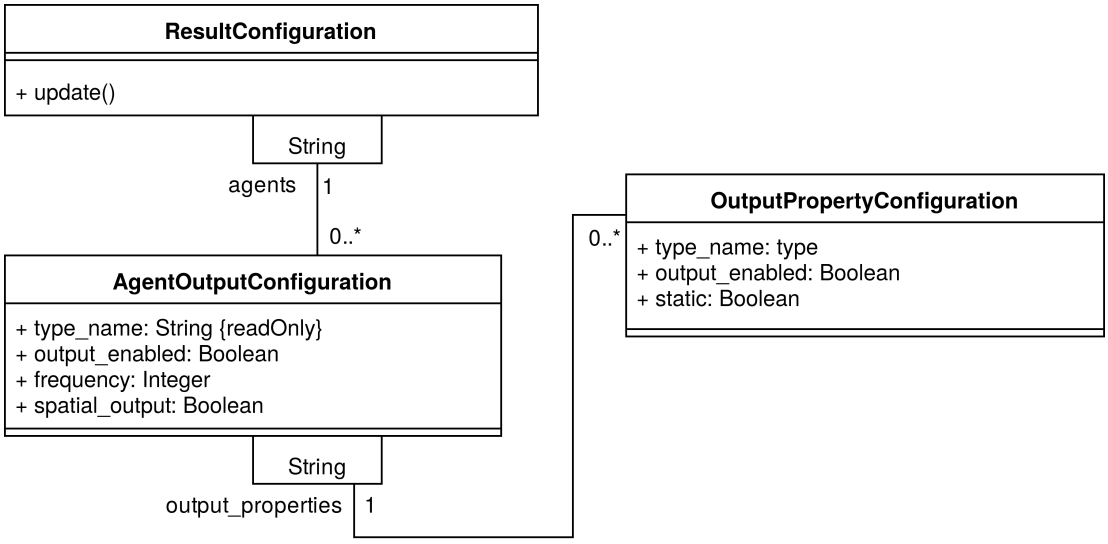


Figure 4.4.: Classes involved into result output configuration

The class diagram fig. 4.4 shows all public classes, which represent a *result (output) configuration*. Section sec. 4.2.2.1 already introduced the class *ResultConfiguration*. As mentioned earlier the *agents* attribute maps agent types on a *AgentOutputConfiguration* instances. Each of the latter

allows to set, if the *output* is *enabled* for the agent type. Further it allows to specify an amount of simulation steps (*frequency*), which must pass between records of the property values into the result database. Further MARS allows to write the positions of spatial agents into the results after each simulation step. This is controlled by the *spatial_output* attribute. Each agent has properties, whose values can be written into the results. Thus the *output_properties* attribute maps their names on *OutputPropertyConfiguration* objects. They allow to *enable* the *output* of a property and to specify, if the property is *static* (set only once at agent's creation and thus only recorded once).

#### 4.2.2.3. Class overview of (component) Client: Scenario updates

Class diagram fig. 4.5 gives an overview about all public scenario related wrapper classes in the component *Client*. The central class *Scenario* (already introduced by sec. 4.2.2.1) provides further wrapper objects, which each represent a certain part of the underlying *model*. Basically these are the agents, the layers and their parameters. Thus the *layers* / *agents* attributes map the layer / agent names on *LayerMappingWrapper* / *AgentMappingWrapper* instances. In turn they connect *parameter* names to *ParameterMappingWrapper* objects. Further the *LayerMapping-Wrapper.agents* map contains only the agents of its layer. Finally *Scenario.categories* groups the layers by their type (e.g. time series layers). Each *MappingWrapper* object offer the assignment of just a *file* and optionally a *column*. As an alternative a *value* can be assigned. The *value* must be of *primitive* type (Boolean, Float, Integer or String). In this case the mapping will be a *value_mapping* (instead of a column mapping). All three attributes can also be read. Not all mapping combinations are valid. The wrapper classes reject several types of invalid mappings (e.g. mapping GIS data on a time series layer). An *AgentMappingWrapper* also allow to manage the agent *count* for the represented agent type.

Beside the structure of maps *Scenario* offers the high level methods *map* and *set_instance_count* to modify the mapping information. Both identify a specific mapping target by a combination of agent, layer or parameter name. The methods base on *find_single_mapping* to retrieve the proper *MappingWrapper* and fail, if the model does not have the specified part. Basically these three data types can be *map*ped and passed to the method's *what* parameter: A primitive value, a *File* object or the tuple of a *File* object and a column name.

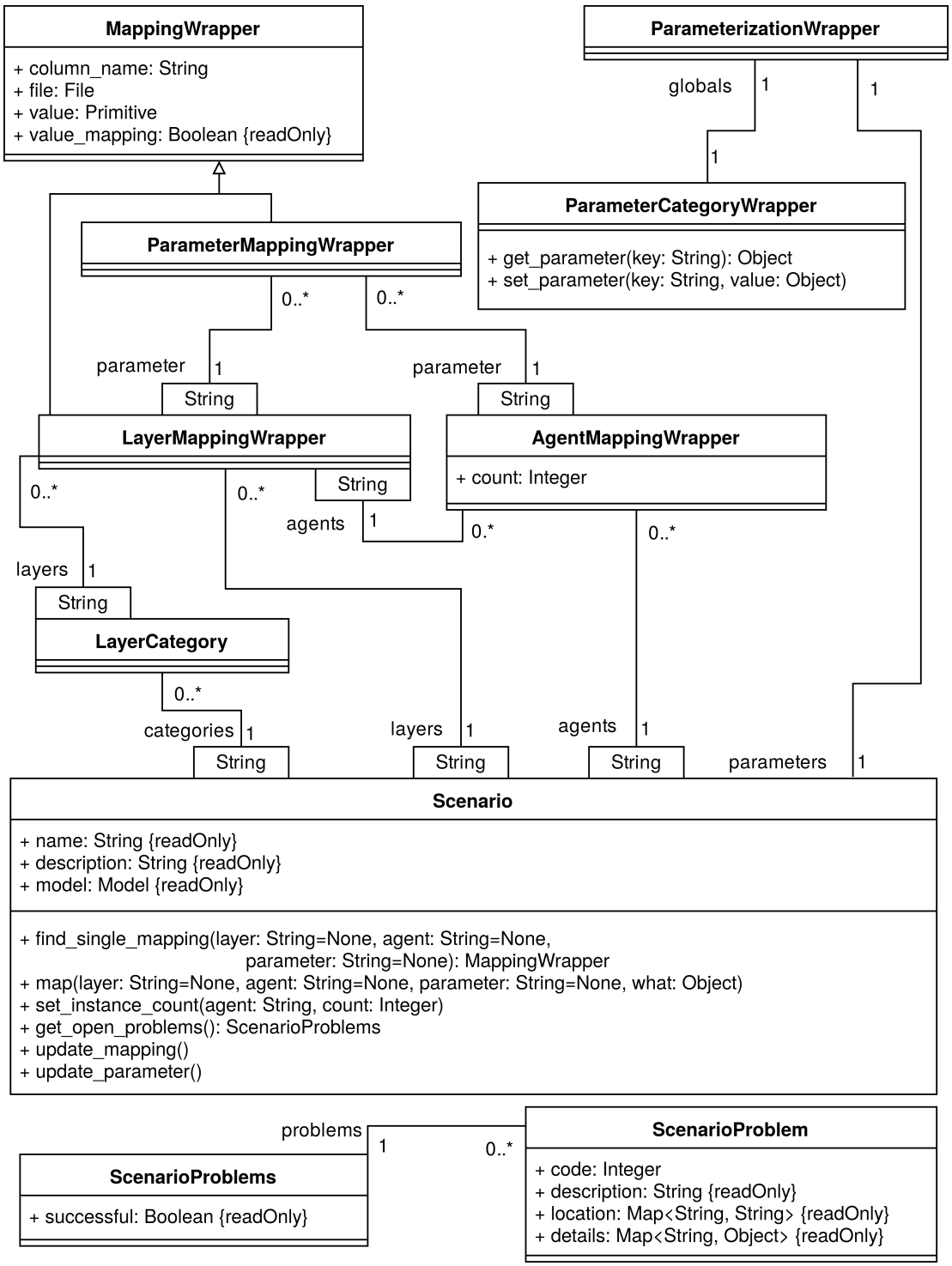


Figure 4.5.: Scenario wrapper classes overview

*Scenario* exposes the *global* scenario parameters by the attribute *parameters*, which refers to a *ParameterizationWrapper* instance. The latter manages categories of scenario parameters. At the time of writing only one exists (*globals*). A *ParameterCategoryWrapper* represents each category. It has methods to *get* or *set* a parameter with a certain name (*key*). These do type checking (on the *value*) and conversion based on informations by the *MARS Scenario Backend Service*. Thus MARS can introduce new parameters and the *Client Library* will immediately support them.

One important task of the *MARS Scenario Backend Service* is to verify the correctness and completeness of a scenario. The *get_open_problems* method of *Scenario* returns a *ScenarioProblems* object, which describes the scenario's state. If everything is fine, the *successful* attribute is true. Otherwise *problems* will contain a list of *ScenarioProblem* instances. Each represents a single problem. It provides the *location* (model part), never changing *code*, a human readable *description* and *details* of the problem.

#### 4.2.2.4. Class overview of (component) Client: Result queries

Diagram fig. 4.6 shows the classes, which wrap the query of results. *QueryWrapper* basically represents a filter for operations on the result database. Multiple filters can be combined by the logical operators *and* and *or*. The result is a new *QueryWrapper*. Each *QueryWrapper* objects offers to *find* agent result data using its filter or to *count* result entries, which match the filter. Each *SimulationRun* object (see sec. 4.2.2.1) refers a *ResultQueryWrapper* instance by its *results* attribute. The latter is a specialisation of *QueryWrapper* and matches all records of the simulation run. Further it offers methods to collect general information about the results. For example *get_number_of_ticks* returns the amount of simulation steps (ticks), for which results are available.

Finally *ResultQueryWrapper* provides *ResultQueryPropertyWrapper* objects. They can create a *QueryWrapper* and thus a filter for a specified result property. The filter consists of the internal property name, one comparison operator and a constant operand. Certain methods of *ResultQueryPropertyWrapper* or its specialisation *NumericResultQueryPropertyWrapper* determine the comparison operator. Currently *equals*, (negated) *equals(_not)*, *greater(_or_equal)* and *less*(*_or_equals*) are available. Further a *one_of* filter matches, if the property equals one of the elements passed as *that*. Meanwhile *distinct* queries the results for all distinct values of the properties in the selection specified by *QueryWrapper query* (and thus is not a operator

method). A *ResultQueryPropertyWrapper* instance checks the type of *that*. For example the object provided by *tick* (of *ResultQueryWrapper*) does not allow the comparison with floating point numbers, since the simulation step (*tick*) is an integer.

The attribute *agents* of *ResultQueryWrapper* maps the name of an agent type on a dictionary, which map the agent's output property names on proper *ResultQueryPropertyWrapper*s. The concrete implementation (e.g. *NumericResultQueryPropertyWrapper*) and the type checking bases on the output property's type definition by the model binary, which the simulation run used. Only output properties can be selected over *agents*, which were actually enabled in the used result output configuration (sec. 4.2.2.2).

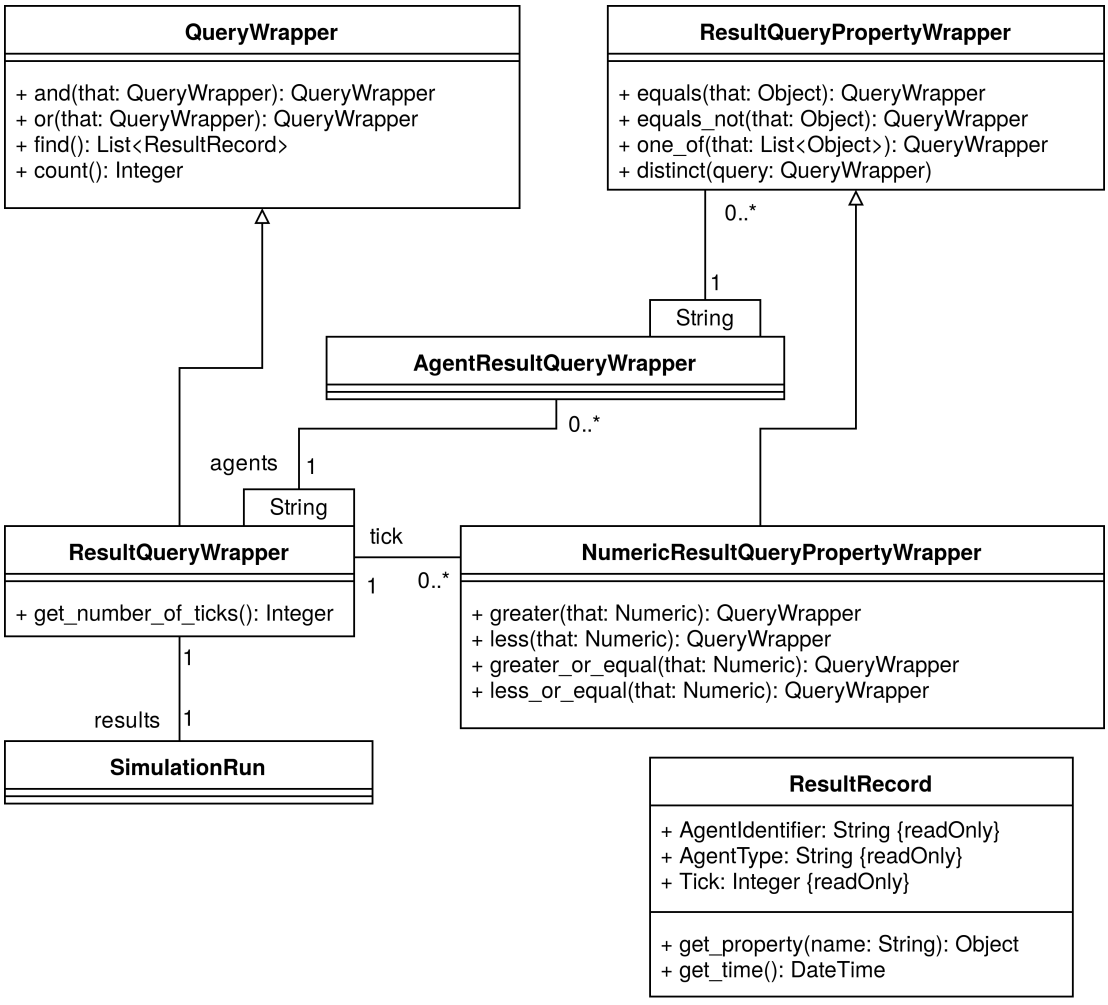


Figure 4.6.: Overview about wrapper classes to query results.

The method *find* of *QueryWrapper* returns the result documents wrapped as *ResultRecords*. Each *ResultRecord* specifies the unique identifier (*AgentIdentifier*) and type (*AgentType*) of the agent, which generated the data of the result record. The method *get_property* returns the value of the output property *name* (of the agent) at simulation step *Tick*. If the underlying result output configuration defines a property as static or specifies an output frequency greater than one (sec. 4.2.2.2), no value might have been recorded at the simulation step. In this case *ResultRecord* will transparently query the last record of it. Meanwhile *get_time()* converts the simulation step into a date and a time (*DateTime* object) based on the global parameters of the underlying scenario (sec. 4.2.2.3).

#### 4.2.2.5. Classes for *Experimental Setup* configuration

The class diagram fig. 4.7 shows several classes, which declare (parts of) an *Experimental Setup* as introduced by section sec. 4.2.1. Each *ExperimentConfiguration* (setup) bases on a *model* and (scalar) *outputs*. A *ModelDefinition* instance declares the model. It bases on a model binary specified by *source*. The latter is either already build (*PrecompiledModel*) or must still be build (*ModelSources*). In first case the (local) *path* points to a MARS compatible model archive, while in the second case it specifies the location of the model's source code. The metainformation *title* and the optional *description* are stored in a *Metadata* object (referenced by *metadata*).

Further *enabled_outputs* of a *ModelDefinition* specifies the output properties of *model*'s agents, which should be recorded. An *EnabledOutput* defines each property by the *agent* and the (*property_*)*name*. Further it can specify, that the property's value is *constant*. Enabling the output for a property implies activating the output for the agent, which provides the property. The output frequency (see also section sec. 4.2.2.2) is fixed to one in the current design.

Further a *ModelDefinition* declares the information needed for setting up a scenario (see subsection sec. 4.2.2.3) by its attributes / constructor arguments *mappings* and *parameters*. The latter is a list of global scenario parameters. Each *GlobalParameter* defines the *name* and the *value* of a global parameter. *Value* may be a primitive value (e.g. boolean or string), date time object or a fitting *Generator* object. The latter generate data as described by a following subsection. Meanwhile *mappings* is a list of *Mapping* objects, which each specify how *data* is mapped on parts of the model. The model part is another time (sec. 4.2.2.3) defined by the combination of *agent*, *layer* and *parameter* names. In the context of an *Experimental Setup data* can be a *File*, a *Generator* a *Column* or a primitive value (e.g. Integer) at the time of writing.

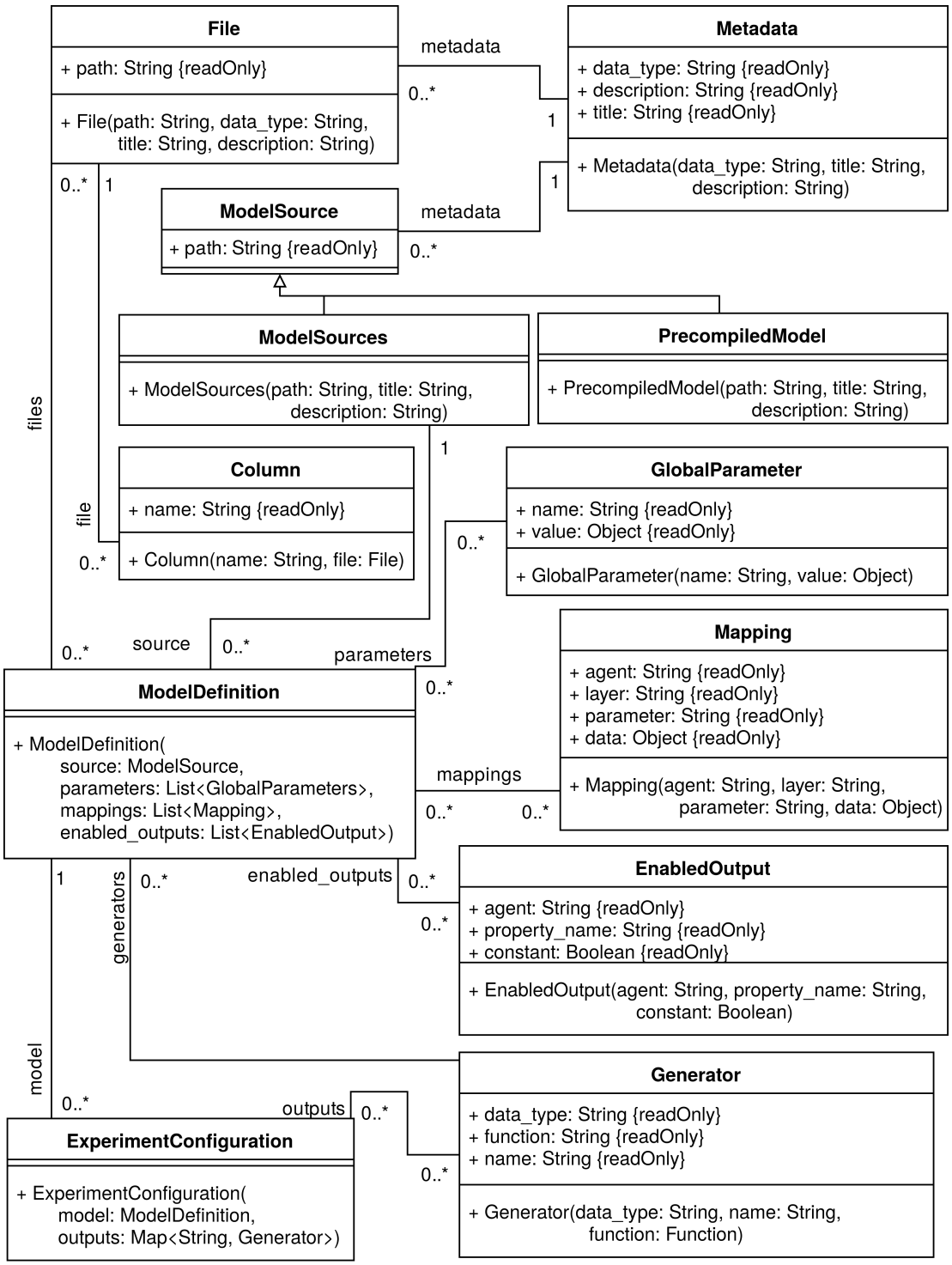


Figure 4.7.: Classes to declare an *Experimental Setup*

A *File* defines input data, which is indepent from the input factor's values and thus constant over all evaluations of the *Experimental Setup*. It has type *data_type* and the local file at *path* stores it. If the data depends on the input parameters, the specified *Generator* creates it at each evaluation. Meanwhile *Column* objects allow to declare the mapping of table columns. For that each object specifies the column's *name* and a *file*, which can be either a *File* or a *Generator* (not visualized by diagram fig. 4.7) for tables.

Finally the scalar results of the *Experimental Setup* are declared by the map *outputs* of *ExperimentConfiguration*. It is a mapping of an output's name on the *Generator* object, which calculates it based on the recorded results of a simulation.

#### 4.2.2.6. Generators

Section sec. 4.2.2.5 introduced *Generator* objects, which create MARS compatible input data from few parameters (e.g. sample generated for a sensitivity analysis). Another task is to aggregate the MARS model's output into multiple scalar output variable. The graphic fig. 4.8 gives an overview about classes, which support the definition of a *Generator*. Every *Generator* creates *data* of a certain *type* (e.g. scalar value or data table) and has a *name*, which can be part of debug messages or the titles of generated MARS resources. Meanwhile *function* is the central part of each *Generator*. *Function* is an user defined high order function, which does the actual work. Its signature (arguments and result type) depends on the concrete *Generator* class.

The basic class *Generator* expects a function, which first argument *data* is a *DataWrapper*. All other arguments can be declared freely by the user. They are input parameters for an evaluation of the *Experimental Setup*, which declares the *Generator*. Meanwhile the evaluation implementation automatically creates *DataWrapper*, which is a data sink for the *Generator*'s *function*. The latter can either *write* binary encoded results using the *BinaryIOStream* object *stream* (of *DataWrapper*). Or it can *write String*s using *text_stream*. *TextIOWrapper* encodes the strings as UTF-8[1] and then it passes them to *stream*. The evaluation code automatically *close*s the data *stream*, after the user defined function finished.

Further a *TableGenerator* creates a table. The list *columns* (exposed as *column_names*) defines the ordering and names of the table's columns. Meanwhile the *data* argument of the *Generator*

[1] https://tools.ietf.org/html/rfc3629

*function* refers a *TableWrapper*, which automatically writes the table header. Further the user defined function can *add rows*. The *TableWrapper* encodes and writes the *column values* in a MARS compatible format.

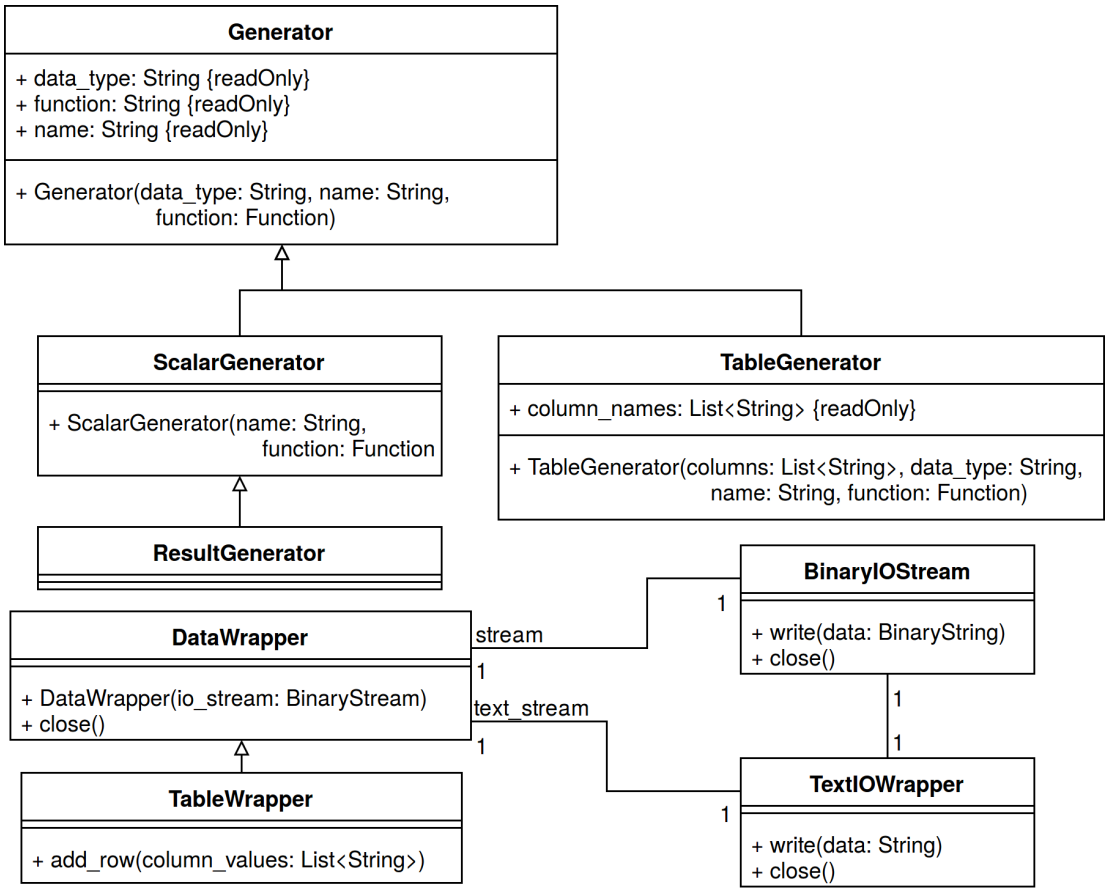


Figure 4.8.: Classes to declare *Generators*

Meanwhile a *ScalarGenerator*'s *function* does not have a special argument like *data*. It just returns the scalar result. Finally *ResultGenerator* except the *function* to have the special argument *result*, which refers to a *SimulationRun* (see subsection sec. 4.2.2.1). Again the result is just returned.

#### 4.2.2.7. *Experimental Setup* evaluation

Beside tools to declare an *Experimental setup*, the *Experiments* component implements batch evaluations of them. This subsection mainly cover the workflows for that.

Each batch evaluation bases on an *Experimental Setup* and a set of input parameter samples. The latter basically is structured as a table, where each column represents an input parameter and each row a sample to be evaluated. Further each batch evaluation is done in a certain MARS project and has an unique identifier.

In a first step the *Experiments* component stores the input parameters for each single evaluation into a database. Further the database stores an unique identifier within the batch, state information and results for every single evaluation. Next the *Experiments* component prepares the constant parts of the *Experimental Setup*. This basically includes:

- the import of the model binary.
- creation of a result output configuration for the model.
- importing all the constant data files, which does not depend on the input parameters.

Each step results in one or more MARS resources, which are then stored into the database together with an identification of the related *Experimental Setup*'s part (e.g. local file path).

After preparation of a batch *Experiments* schedules its evaluations, which each involves these three processes:

- **Preparation** creates the remainding MARS resources.
- **Simulation** triggers a simulation run in the MARS cloud and waits for its completion.
- **Aggregation** runs the output generators.

The successful completion of each process updates the main state of the single evaluation (in the database). An interrupted *Preparation* or *Aggregation* process can not be resumed. But *Experiments* will repeat it. Same applies to a process, which failed by temporary problems (e.g. connection issue). *Experiments* keeps track about the retry count. If the latter gets about a certain threshold, the single evaluation will finally fail. Also the time between two tries increases with the count to overcome a stress situation in the MARS cloud.

The flowchart fig. 4.9 visualizes the *Preparation* process. First *Generator objects* (sec. 4.2.2.6) of the *Experimental Setup* create *data* and *single values* based on the *parameter sample*. Next *Experiments* uploads the *data* to the MARS cloud using the *Client* component (see sec. 4.2.2.1) and triggers the *import*. The result (in case of success) are several files. Their titles and description contain information about the evaluation, especially the used input parameter values. In the next step *Experiments creates* and configures a *scenario* object. This involves the merging (global) *parameter template* and *mapping template* with the results of the *data generation*, the *data import* and the global batch preparation described earlier. Finally a *plan* is created based on the *scenario's identifier* and the *result configuration identifier*, which was created while the batch preparation. The database persists the *plan identifier* as the *Preparation*'s result.

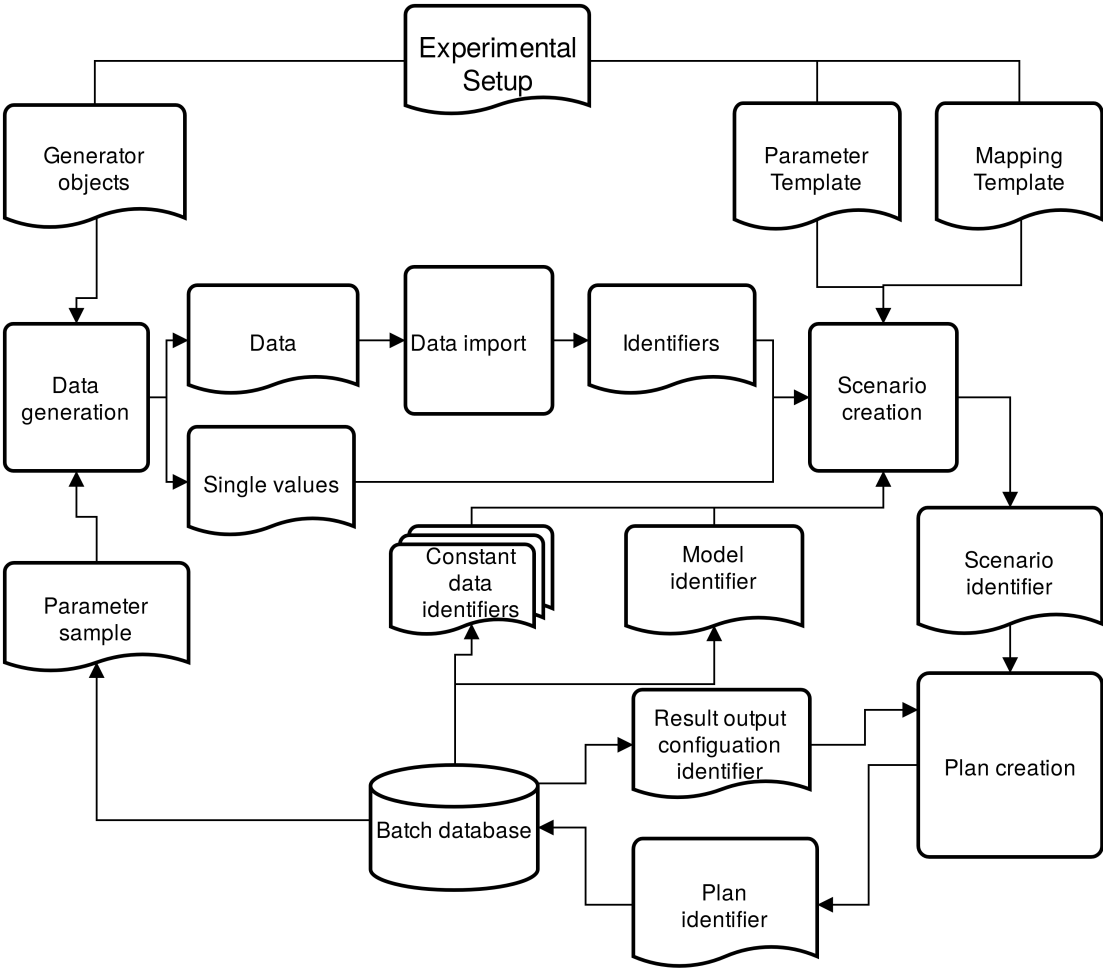


Figure 4.9.: Preparation of a single evaluation

The *Simulation* process continues with creating simulation runs based on the plan. Their identifiers are stored immediately into the database. Afterwards *Experiments* regularly queries the state's of the simulation runs from the MARS cloud. As soon as MARS completed the simulations, *Experiments* schedules the *Aggregation*, which uses *Generator* objects of the *Experimental Setup* to calculate the scalar outputs. Afterwards it stores the outputs into the database.

### 4.2.3. Command line interface (CLI)

The core design of the command line interface (CLI) is inspired by Kubernetes' CLI utility[2] in the aspect, that it basically offers methods on resources. As described by the previous subsections, there are following resource types supported by MARS: user, project, file, scenario, result output configuration, (simulation) plan and run. The CLI allows to create, describe, delete, get (list) and update resources (of these types). Further the CLI has resource independent commands. This includes building (and packing) of model sources, testing a *Generator* (see sec. 4.2.2.6) and controlling batch evaluations (sec. 4.2.2.7). All CLI commands can have (optional) arguments to control their behaviour or to pass information.

The architecture of the CLI utility follows a model view controller pattern and diagram fig. 4.10 show its components. *Argument Parser* analyzes the user input and parses the arguments (and command) from it. It offers an interface, which allows the *Controller* to declare the expected arguments and to retrieve the argument's values. Further *Controller* calls the function of *Use Cases*, which implements a specified command. Finally the *Controller* hands the results of the *Use Cases* method back to the user using the *OutputPrinter* component.

*Use Cases* offers functions for resource independent commands and a public class for each resource type, which in turn has command methods. For example figure fig. 4.11 shows the class for result output configurations. The method *create*'s arguments provide the information, which must or can be available while the initialization of a resource. In case of a result output configuration this is the *model* and the *name*. Another resource can be specified by either its complete unique identifier or parts of its name (title in case of files). If *Use Cases* finds multiple matching resources in latter case, it throws an exception with all matches. Then the *Controller* returns the choices to the user. Meanwhile the *delete* methods for all resource types must know the *resource* to be removed. Resolving the *resource* works similar to *model*

[2] https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands

as described before. *Cascading* deletion also removes resources, which depend on *resource*. *Describe* returns several information *Table*s (two dimensional array) about a resource. For a result output configuration these include amongst others a metainformation table and a table with the output settings for every output property of the underlying model (see sec. 4.2.2.2). The method *get* returns a table, where each row represents a result output configuration. The columns contain metainformation like the name or the model identifier. Finally the *update* of a result output configuration *resource* takes several arguments (not extensive list):

- *Sub_command* specifies if the method call *enables* or *disables*.
- *Output_properties* is a list of agent name and property name tuples, which the *ResultConfigurationHandler* shall *enable* or *disable*.
- *Agent* relates to the general output settings of the listed agent names.

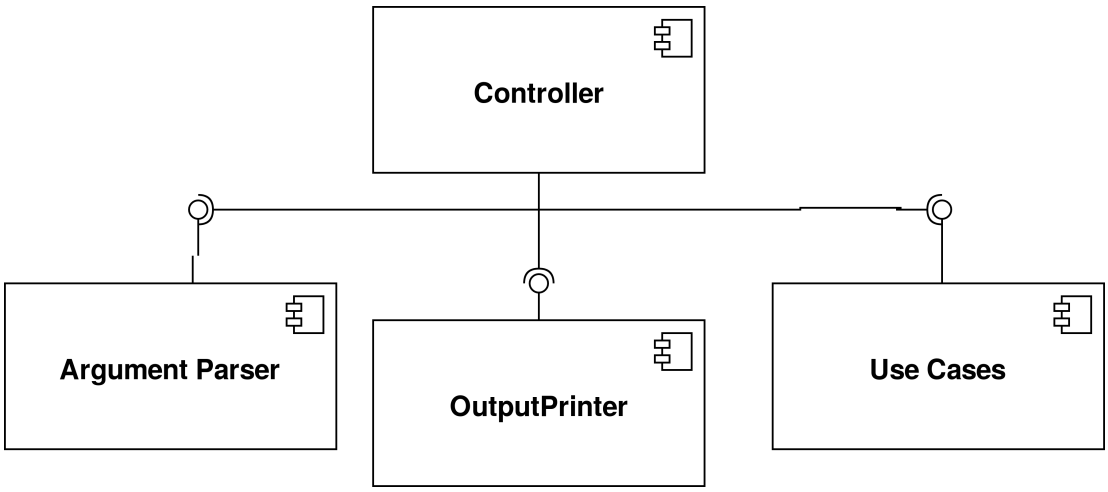


Figure 4.10.: Component overview of the command line utility

**ResultConfigurationHandler**

```
+ create(model: String, name: String)
+ delete(resource: String, cascading: Boolean=False)
+ describe(resource: String): List<Table<String>>
+ get(): Table<String>
+ update(resource: String, sub_command: String, output_properties: List<Tuple<String, String>>,
          agent: List<Tuple<String, Integer>>, [...])
```

Figure 4.11.: CLI handler class for result output configurations

## 4.3. Implementation state and details

Implementations exist of most features, which this chapter describes. This subsection gives an overview about the feature's sources, state, tests and delivery.

The command line interface (CLI) and the client library share a source repository[3]. Further there exist one continous build pipeline, which runs the automatic tests and delivers both as one package[4]. Basically a Python subpackage implements each described component. Also the structure of the sourcecode directory *mars_tools* reflects the components. One exception is the CLI, since in opposite to its design the implementation integrates the *OutputPrinter* into the *Use Cases*. Further the *configure_* methods separate the configuration of Pythons standard *argparse* module from the *Use Cases*. Thus *argparse* has the role of the *Argument Parser*, while the *configure_* methods act as the *Controller*. But there is no separation on the package level.

Table 4.1 gives a state and test overview of the client library (sec. 4.2.2).

Table 4.1.: Implementation state overview

| Feature | State | Automatic tests | Section |
|---|---|---|---|
| Resource creation | no users | system tests | 4.2.2.1, 4.2.2.2 |
| Resource retrieval | no users, no input data | system tests | 4.2.2.1, 4.2.2.2 |
| Resource updates | scenarios, result cfg. | system tests | 4.2.2.1, 4.2.2.2 |
| Resource deletion | highly experimental | - | - |
| Result queries | no result wrapping | - | 4.2.2.4 |
| Data models | supports complex objects | component | 4.2.2 |
| Model building | done | - | 4.2.2 |
| Manifest creation | not implemented | - | 4.2.2 |
| Manifest processing | no result cfg. | component | 4.2.2 |
| Experimental setup classes | as described | - | 4.2.2.5, 4.2.2.6 |
| Batch evaluation | local, SQLite3 based | - | 4.2.2.7 |
| Command line interface | for all implemented features | - | 4.2.3 |

[3] `https://gitlab.informatik.haw-hamburg.de/mars/mars-python-client`
[4] Package *mars-python-client* in MARS' Pypi feed: `https://nexus.informatik.haw-hamburg.de/repository/pypi/simple`

# 5. Usage examples

This chapter shows how to technically use the systems, which the previous two chapters describe. Further it documents the evaluation, how well MARS handles the presented use cases. Since the author did not manage to implement the Basic Immune Simulator completly with MARS at the time of writing, it makes no sense to search for non-influential parts yet. Thus the last part of this chapter does a sensitivity analysis with MARS on a variant of the wolves and sheep model (similar to Prestes García and Rodríguez-Patón (2016)).

## 5.1. Environment of the experiments

### 5.1.1. Hardware

While the experiments an average computer was the client. Listing 5.1 show its hardware resources as reported by the operating system: 15909 Megabyte of memory (line 4), 2 physical cpu cores (line 18, line 16 => 4 logical cores) and a Solid State Disk (SSD) hard drive. Line 28 shows benchmark results of the SSDs read speed (498 MB/s) and line 31 shows a measured write speed of 200 MB/s.

The MARS cloud ran on a heterogeneous Kubernetes cluster with computation nodes, which differ significantly regarding their hardware resources. Table 5.1 shows an overview about the processor and memory resources. They were basically determined the with the Linux utilities *free* and reading */proc/cpuinfo* using *cat* as on the client (lst. 5.1). But instead of the client a Kubernetes pod on the specific node ran the commands. Listing 5.2 shows the usage of the Kubernetes client utility *kubectl* to identify the pod and run the command. A Ceph[1] shared storage persists data. A database pod ran the benchmark of listing 5.1's line 29 to get an idea about the write speed of a single pod on a Kubernetes persistent volume provided by Ceph.

[1] `http://ceph.com/`

The result was 92 MB/s. It should be taken in account that the shared storage (used for the experiment) can provide this speed to multiple pods at the same time. Since the author had no exclusive access to the Kubernetes cluster, it is possible that others used it a the same time. Thus quantitative performance numbers in this work has to be taken with care.

Client and Kubernetes Cluster were connected over the internet. The client side internet connection has a nominal bandwidth of 25 MBit/s upstream and 5 MBit/s downstream.

```
1  lukas@skybookHP:~$ free -h
2      lukas@skybookHP:~$ free --mega
3                     gesamt       [...]
4      Speicher:        15909      [...]
5      [...]
6  lukas@skybookHP:~$ stress -c 5 &
7      [...]
8  lukas@skybookHP:~$ cat /proc/cpuinfo
9      processor   : 0
10     [...]
11     model name  : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
12     [...]
13     cpu MHz     : 3099.877
14     cache size  : 3072 KB
15     physical id : 0
16     siblings    : 4
17     core id     : 0
18     cpu cores   : 2
19     [...]
20 lukas@skybookHP:~$ sudo smartctl -a /dev/sda
21     [...]
22     Device Model:     Samsung SSD 750 EVO 250GB
23     [...]
24     SATA Version is:  SATA 3.1, 6.0 Gb/s (current: 6.0 Gb/s)
25     [...]
26 lukas@skybookHP:~$ sudo hdparm -tT --direct /dev/sda
27     [...] disk reads: 1496 MB in  3.00 seconds = 498.32 MB/sec
28 lukas@skybookHP:~$ sudo dd if=/dev/zero of=~/tmpfile \
29 oflag=dsync bs=1G count=1
30     [...]
```

```
31     1073741824 bytes [...] copied, 4,53743 s, 237 MB/s
```

Listing 5.1: Hardware resources on the client reported by its Linux operating system.

```
1 lukas@skybookHP:~$ kubectl -n mars-mars-beta get po -o wide | \
2 grep icc-node-3
3     [...]
4 sim-mon-svc-vs8kw   [...]  icc-node-3.ful.informatik.haw-hamburg.de
5 lukas@skybookHP:~$ kubectl -n mars-mars-beta exec -it | \
6 sim-mon-svc-vs8kw -- free -m
7     [...]
```

Listing 5.2: Finding pod and running command with kubectl

Table 5.1.: Hardware resources of some Kubernetes cloud computation nodes

| Node | Processor | physical / logical cores |
|---|---|---|
| icc-mars-node-01 | 8 / 8 | 128909 MB |
| icc-mars-node-02, -03, -04, -05, -07 | 6 / 12 | 64468 MB |
| icc-mars-node-06, -08 | 6 / 12 | 64400 MB |
| icc-mars-node-09 | 12 / 24 | 96744 MB |
| icc-mars-node-10 | 4 / 4 | 15871 MB |
| icc-mars-node-11 | 4 / 4 | 16070 MB |
| icc-mars-node-12, -13 | 4 / 4 | 8006 MB |
| icc-mars-node-14 | 4 / 8 | 16055 MB |
| icc-node-1, -2, -3 | 8 / 8 | 128909 MB |

### 5.1.2. Software

An Ubuntu 16.04 Linux distribution operated the client[2]. Further the experiment execution based on Python 3.5.2. Newer versions probably work, but are not tested. Older ones will certainly break parts of the described tools. .NET Core 2.0.3 or newer can build the model binaries. MARS package feeds provide all required libraries (by models and the toolset). Listing

[2] Most Unix based systems (e.g. Mac OS X, comparable hardware) should be able to run the showed experiments. Though this was not tested at the time of writing. Windows probably is more tricky. But basically all used libraries and tools have ports.

5.3 shows the required changes on the package manager's configuration files on an Ubuntu 16.04 based system. Listing 5.4 shows how to use the Python package management tool *pip* to install the tools described by chapter 4. Since those tools wrap the calling of the .NET Core tools (e.g. compiler), this section does not cover, how to install the libraries required by the models explicitly.

The use of *virtualenv*[3] prevents, that library versions required by the MARS Python tools conflict with the requirements of other applications on the client. Since the MARS Python tools are strict about the required versions at the time of writing, the usage of *virtualenv* is recommended.

```
lukas@skybookHP:~$ cat /home/lukas/.pip/pip.conf
    [global]
        index-url=https://nexus.informatik.haw-hamburg.de/\
        repository/pypi/simple
lukas@skybookHP:~$ cat /home/lukas/.nuget/NuGet/NuGet.Config
    <?xml version="1.0" encoding="utf-8"?>
    <configuration>
      <packageSources>
        <add key="mars"
         value="https://nexus.informatik.haw-hamburg.de/\
         repository/nuget-group/" protocolVersion="2" />
    [...]
```

Listing 5.3: Modifications on clientside configuration files of the package managers

```
lukas@skybookHP:~$ virtualenv --python=python3.5 master_env
    [...]
    Installing setuptools, pkg_resources, pip, wheel...done.
lukas@skybookHP:~$ source master_env/bin/activate
(master_env) lukas@skybookHP:~$ pip install mars-python-client
    [...]
    Successfully installed [...] mars-python-client[...]
```

Listing 5.4: Install of the tools described by the current chapter and previous chapter

The experiments were executed against the offical MARS Cloud beta deployment. Commit 4cc77012a595bb8fda5a36a0a26d22a25e64965d of its repository[4] declares its state (e.g. versions of

[3] https://virtualenv.pypa.io/en/stable/
[4] https://gitlab.informatik.haw-hamburg.de/mars/mars-beta

services) at the time of writing. While doing the experiments the MARS Cloud beta deployment was accessed over the URL `beta.mars.haw-hamburg.de`.

Finally the official[5] or any other compatible *Git* client can be used to check out the sources, on which this chapter bases on.

### 5.1.3. Registration at the MARS Cloud and initialization of client tools

Beside the software described in the previous section an unlocked MARS Cloud account is required to do the experiments, which this chapter describes. At the time of writing using the MARS Teaching UI is the only available way to request an account (as shown by screenshot fig. 5.1). An administrator of the MARS Cloud needs to unlock the account.

Then the client tools can be configured to use the credentials of the previously created account. Listing 5.5 shows the process for the MARS command line based user interface (MARS CLI). If the latter does not find the specified configuration (in the user's home directory if not specified otherwise), it starts an interactive creation process. The error message in line 11 just informs, that the default project *test* does not already exist. Anyway after configuration MARS CLI continues to *create* the *project my_project* and prints the project's global identifier (last line of listing 5.5).

```
1  (master_env) lukas@skybookHP:~/$ mars create project my_project
2      Since specified MARS configuration file does not exist,
3      it will now be created based on your input.
4      First insert the URL of a MARS cloud deployment
5      [...]: beta.mars.haw-hamburg.de
6      [...]
7      Registered login name: lukas2
8      [...]
9      Password (hidden input):
10     Name of your favourite project: test
11     ERROR:root:No project with identifier or name test found
12     29f90dc3-3395-4a54-94f7-3a1282b787a2
```

Listing 5.5: Interactive creation of the MARS command line tool's configuration

[5] `git-scm.com`

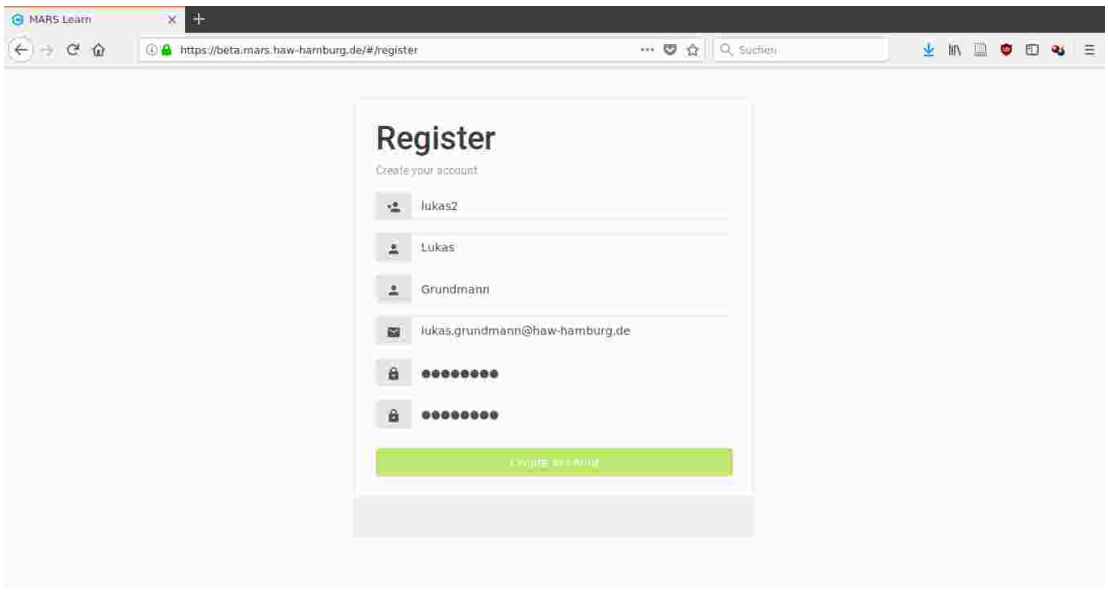


Figure 5.1.: Account creation form of the MARS Teaching UI

## 5.2. Usage of the MARS based Basic Immune Simulator (BIS) implementation

### 5.2.1. Workflow overview

The original workflow to run single evaluations of the MARS BIS model is older than the MARS command line interface (CLI). While the model specific support tool, *is_tool* still is the only way to analyse the results, parts of the simulation preparation features of *is_tools* are deprecated by the MARS CLI. This section only covers the new workflow, while a previous project report[6] of the author describes the old workflow involving the MARS Automatisation Service (see sec. 4.2.1).

The graphic 5.2 shows an overview about the current workflow. First the *Parameter Preparer* (sec. 3.1.1) generates tables with the agent parameters based on a global *parameter file*. Then *MARS CLI* uploads the previously created tables, uses *.NET Core* to build the *model sources* (to the *model binary*) and maps the data on the model based on the *resource manifest*. The final result is a *simulation plan*. Further *MARS CLI* can start *simulation runs* straight from the plan.

[6] `http://lukas-grundmann.de/pubs/PO2.pdf`

Also it can create a new scenario (and plan) based on the old plan and *updated model sources* (build / import process not shown by fig. 5.2) or a *new tick count*. Anyway the user can specify the *number of runs*. Finally the *model specific analysis* (sec. 3.1.8) from *is_tool* generates *plots* and an *index file*, which refers the *plots*.

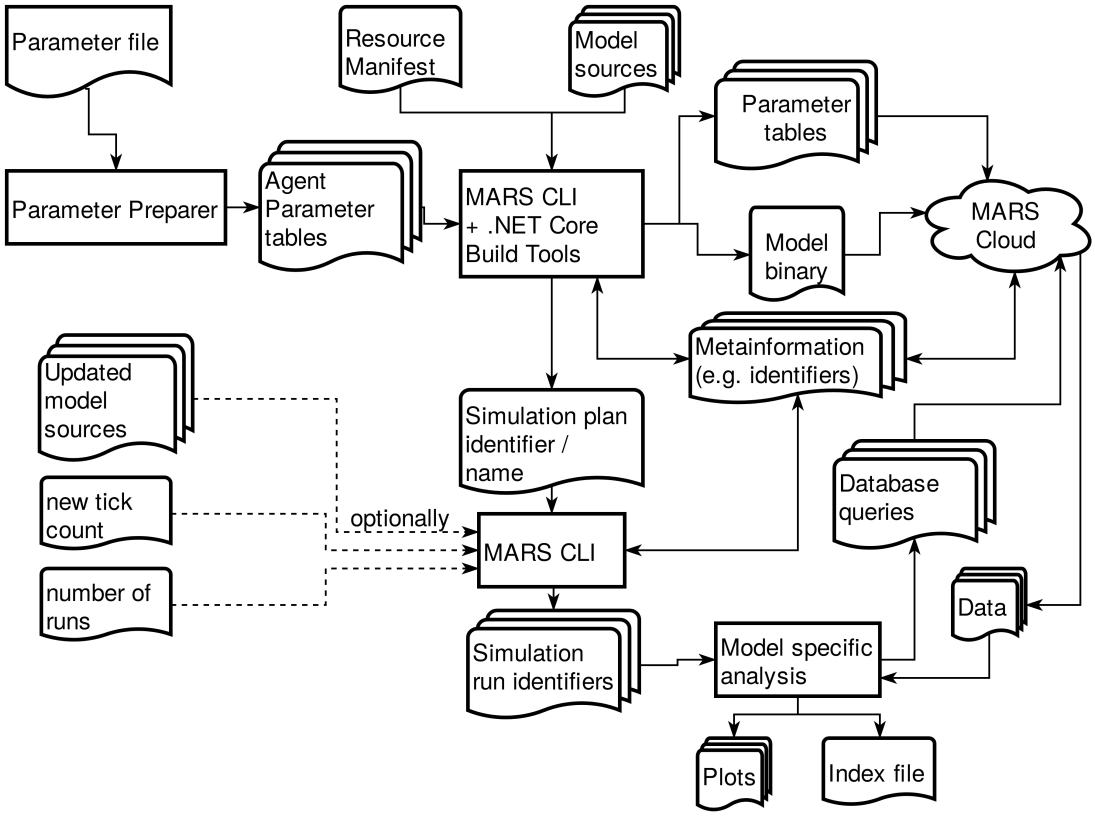


Figure 5.2.: Current workflow of BIS model evaluation

### 5.2.2. Initial, one time preparation

```
(master_env) lukas@skybookHP:~$ git clone https://GITLAB_USER\
@gitlab.informatik.haw-hamburg.de/mars/model-immune-system.git bis
    [...]
(master_env) lukas@skybookHP:~/bis$ pip install is_tool \
parameter_preparer
    Successfully installed [...] immune-system-model-tools-1
    [...] parameter-preparer-0.0.1 [...]
(master_env) lukas@skybookHP:~/bis$ is_tool --config is.cfg \
```

```
9  from-mars-cli-config ~/.mars.config
10 (master_env) lukas@skybookHP:~/bis$ cat is.cfg
11     {"url": "https://beta.mars.haw-hamburg.de",
12     "project": "test", "user": "lukas2", "password": [...]}
```

Listing 5.6: Checkout of MARS BIS and configuration of model specific tools

Listing 5.6 shows the checkout (lines 1, 2) of the MARS BIS model's repository. Further the model specific utilities *parameter_preparer* and *is_tool* are installed (lines 4, 5). Finally in line 8, 9 *is_tool* generates a configuration (*is.cfg*) in its own format from MARS CLI's configuration file (*~/.mars.config*, default location of the file). Afterwards the listing reveals the new file's content, which basically is the same as the MARS CLI configuration provided in section 5.1.2 with different key names.

### 5.2.3. Initial parameter file and agent parameter generation

The file *parameter.yml* in the root of MARS BIS' repository bases on Folcik, An, and Orosz (2007), additional file 17. Listing 5.9 shows few parts of the file. There are the metagroups (sec. 3.1.1) *areas*, *_agents* and *antigens*. *Areas* contains the zone / area definitions. The *depth* of all zones is *1* to reflect the two dimensional grids used by Folcik, An, and Orosz (2007) for agent positioning. *&Zone1* in line 2 and *&Zone2* in line 8 are YAML[7] anchors, which allow to include the zone mappings in a later part of the document (e.g. line 23). Also anchors are used for parameter inheritance (lines 25 and 29). The commentaries (prefixed by #) give a hint to the related parameters of Folcik, An, and Orosz (2007) original BIS implementation. While the *bcells* groups use the *SomeOtherAntigen* antigen representation for their receptor, *virus* bases on *SomeAntigen*. All bits of those both differ as visualized by listing lst. 5.8. Thus the resulting binding probability is 1.0 (sec. 3.1.3).

```
1 (master_env) lukas@skybookHP:~/bis$ time parameter_preparer\
2 -q -o Data/ -i parameter.yml
3     real    0m5.984s
4     [...]
5 (master_env) lukas@skybookHP:~/bis$ head -n 2 Data/parenchymalcell.csv
6     z,[...],y,regeneration-time,area,duration-stressed,_groupname[...]
7     0.0,[...],0.0,1,Zone1,25,parenchymal-cells[...]
8 (master_env) lukas@skybookHP:~/bis$ cat Data/parenchymalcell.csv \
9 | wc -l
```

[7] http://www.yaml.org/spec/1.2/spec.html

```
10      4225
```

Listing 5.7: Parameter generation with peek into the results

Listing 5.7 shows a call of *parameter_preparer* (line 1 and 2). The preceding *time* returns the *real* time, which the call took (~ 6s). While the parameter *-q* surpresses most (debug) outputs of *parameter_preparer*, *-i parameter.yml* specifies the initial parameter file and *-o Data/* sets the directory */home/lukas/bis/Data* as target directory for the parameter tables. The next chapter will show the importance of *Data* being in the root of the local working copy of MARS BIS' repository. The remainder of listing 5.7 shows the parameter set for one *ParenchymalCell* and counts the number of rows in the table file. The row number of *4225* corresponds to the *_number* field in line 19 of listing 5.9 (table contains *4224* data rows plus a header).

```
1 >>> for x in ["0xFF77FFFF", "0x00880000"]:
2 ...     print("{0:0=32b}".format(int(x, 16)))
3 ...
4 11111111011101111111111111111111
5 00000000100010000000000000000000
```

Listing 5.8: Calculation of the binary representation of the bitstrings SomeAntigen and SomeOtherAntigen

```
1  areas:
2      Zone1: &ZONE_1
3          _type: areainformation
4          width: 100
5          height: 100
6          depth: 1
7          [...]
8      Zone2: &ZONE_2
9      [...]
10 _agents:
11     [...]
12     parenchymal-cells:
13         ab1-lysis-threshold: 100 #signal units <=> Ab1_Lysis_Threshold
14         regeneration-time: 1 #ticks <=> DelayRegenerationTime
15         duration-stressed: 25 #ticks <=> DURATION_Stressed
16         area: Zone1
17         activate-at: 0 #ticks
18         _type: parenchymalcell
```

```
19          _number: 4224 #agents; source: BIS java code + own calculation
20          position:
21              type: XYZGrid
22              parameters:
23                  <<: *ZONE_1
24      [...]
25      bcells: &DEFAULT_B_CELL
26          [...]
27          specify: SomeAntigen
28      anti-virus-b-cells:
29          <<: *DEFAULT_B_CELL
30      [...]
31      virus:
32          antigen: SomeAntigen
33      [...]
34  antigens:
35       SomeAntigen:
36           _type: antigeninformation
37           format: bitstring
38           source: "0xFF77FFFF"
39       SomeOtherAntigen:
40           [...]
41           source: "0x00880000"
```

Listing 5.9: Parts of initial parameter file (parameter.yml)

### 5.2.4. Using the resource specification file to prepare simulations

Listing lst. 5.10 shows snippets of the resource specification[8] (*mars_bis.manifest.yml*), which declares the location and metainformation of the previously created parameter files (e.g. *Data/parenchymalcell.csv* in line 2-5) relative to its own path. Further the file specifies the model binaries (line 6-10) source code and a *scenario*. The latter bases on the model (line 15), an external *mapping* file (line 16, see also lst. 5.11) and certain *parameters*. Since MARS lacks the concept of abstract time mapping used by the BIS model, the absolute *parameter* values are not important. Only the resulting number of simulation steps matter.

[8] The author's project report `http://lukas-grundmann.de/pubs/PO2.pdf` provides a format description of the resource specification, while this section focus on the MARS BIS' specific contents.

```
[...]
parenchymalcell:
    data_type: TABLE_BASED
    file_path: Data/parenchymalcell.csv
[...]
bis_model:
    data_type: MODEL
    [...]
    code: ./Code/OnlyModelWithoutTests.sln
    [...]
scenario:
    type: scenario
    name: ${experiment_name}
    [...]
    model: $bis_model
    mapping: mapping.yml
    parameters:
        global_parameters:
            SimulationStartDateTime: 1.1.2017
            SimulationEndDateTime: 5.1.2017
            DeltaT: 6
            DeltaTUnit: hours
plan:
    type: plan
    name: ${experiment_name}
    scenario: ${scenario}
    [...]
```

Listing 5.10: Snippets of the current resource manifest for MARS BIS

```
agents:
  [...]
  ParenchymalCell:
    count: 4224
    parameters:
      ab1LysisThreshold: {column: ab1-lysis-threshold,
                          data_id: "${parenchymalcell}"}
  [...]
```

Listing 5.11: Snippet of the MARS BIS' mapping file

Listing 5.12 shows how to apply the resource declarations using the MARS CLI. The resource specification *./mars_bis.manifest.yml* references three external variables, which the user passes to *mars apply*. The variables influence the names and descriptions of the created MARS resources (see listing 5.13) and aim to support the traceability of certain simulation results. Meanwhile the *-p* flag specifies the target project *demo_project*. It can be omitted, if a default project was defined before.

Again *time* was used to roughly measure the duration of setting up the resources. Listing 5.14 shows that the model build consumes the most of the time (on the client).

```
(master_env) lukas@skybookHP:~/bis$ time mars apply \
./mars_bis.manifest.yml --variable experiment_name="Demo1"\
--variable commit_id=$(git rev-parse HEAD) \
--variable timestamp="$(date)" -p demo_project
    [...]
    INFO:root:Upload of /parenchymalcell from [...] finished
    INFO:root:Upload of /thelpercell from [...] finished
    [...]
    INFO:model-builder:Packed /tmp/tmpummr409e.zip
    [...]
    INFO:root:Upload of /bis_model from /home/lukas/bis finished
    INFO:resource_processing:Check import state for /bis_model
    [...]
    INFO:resource_processing:Did mapping of scenario /scenario [...]
    [...]
    INFO:resource_processing:Created plan /plan from /home/lukas/bis
    [...]
    Finished resources
        [...]
        /parenchymalcell => e39c2a8b-029a-49bd-868a-f8f068e934e8
        /plan => ['5a4a54af0ea6df00013e5cef']
    [...]
    real    1m47.350s
    [...]
```

Listing 5.12: MARS CLI applies the resource specification of the MARS BIS model.

```
(master_env) lukas@skybookHP:~/bis$ mars get -p demo_project file
    identifier   title                                status  [...]
    [...]
```

```
4     [...]        Virus setup for experiment Demo1        FINISHED[...]
5     [...]
6  (master_env) lukas@skybookHP:~/bis$ mars describe scenario \
7  -p demo_project Demo1
8     [...]
9     Name:           Demo1
10    Description:    [...] based on commit 6b1[...] \
11                     around Mo 1. Jan 16:31:16 CET 2018.
12    Model title:    BIS MARS Model for experiment Demo1
13    [...]
14    Open scenario problems:
15    Layer mappings:
16    [...]
17    Mapping of agent ParenchymalCell with count 4224:
18         x: [...] x from e39[...] (PC properties for [...] Demo1)
19         ab1LysisThreshold:  ab1-lysis-threshold [...]
20    [...]
```

Listing 5.13: Evaluation of previously created file and scenario resources

```
1  (master_env) lukas@skybookHP:~/bis$ time mars build -o bis.zip\
2  Code/OnlyModelWithoutTests.sln
3     [...]
4     INFO:model-builder:Packed bis.zip
5     [...]
6     real    1m18.301s
7     [...]
```

Listing 5.14: Measurement of the time consume for a build of MARS BIS'.

### 5.2.5. Start of simulations and retrieving their state

Section 5.2.4 shows how to setup MARS resources, which base on the MARS BIS model and related input data. One resource is the simulation plan. Now this section describes, how to start simulations based on this plan and how to observe the simulations' state. Listing 5.15 shows the straightest way by just creating MARS simulation *run* instances. To get rid of the *-p* project flag, line 4 switches sets *demo_project* as the current default. The remainder of the listing *create*s three simulation *run*s using the same plan. A plan (as every MARS resource with

a name / title) can be specified by parts of its name or the complete unique identifier. MARS CLI reports a failure in the case of multiple matching resources.

Listing 5.16 shows a call involving MARS CLI's *run* command. That command copies a scenario and plan combination with certain modifications. In the example the *-t* flags reduces the amount of simulation steps to *2*. The *-r* option requests *4* simulation runs, which bases on the new plan and scenario combination. Further the user can define the use of alternate model sources / binary and a certain name or description (for the new scenario and plan). For every command MARS CLI shows a help text, if the user adds the "-h" flag. The *run* command waits until all created simulation runs finish. Meanwhile it shows an output as shown by listing 5.17.

Finally listing 5.18 gives an example usage of MARS CLI's *get* to list all *run*s of a project (restriction to a certain plan is possible, use *-h* flag for details). Further *mars describe* returns details about a specific run. The debug output reveals (for the specific simulation) that the time to calculate each simulation step started to grow exponential from tick 10 onwards.

```
(master_env) lukas@skybookHP:~/bis$ mars get -p demo_project plan
identifier                              name
5a4a54af0ea6df00013e5cef                Demo1
(master_env) lukas@skybookHP:~/bis$ mars set-default-project \
demo_project
(master_env) lukas@skybookHP:~/bis$ mars create run Demo1
    5a4a84010ea6df00013e5cf0
(master_env) lukas@skybookHP:~/bis$ mars create run Dem
    5a4a84120ea6df00013e5cf1
(master_env) lukas@skybookHP:~/bis$ mars create run \
5a4a54af0ea6df00013e5cef
    5a4a84390ea6df00013e5cf2
```

Listing 5.15: Start of three simulations based on the previously created plan

```
(master_env) lukas@skybookHP:~/bis$ mars run -t 2 -r 4 Demo1
```

Listing 5.16: Call example of mars run

```
State of simulation runs at 20:19:44
Run                                 State               CurrentTick
5a4a89b10ea6df00013e5cf9            Succeeded           2
5a4a89bb0ea6df00013e5cfa            Running             1
```

```
5a4a89bb0ea6df00013e5cfb            Succeeded           2
5a4a89bb0ea6df00013e5cfc            Succeeded           2
```

Listing 5.17: Example output of mars run

```
(master_env) lukas@skybookHP:~/bis$ mars get run
    identifier     status      plan           current_tick [...]
    [...]13e5cf0   Running     5a4a54af0ea6df00013e5cef   19   [...]
    [...]
    [...]13e5cf9   Succeeded   5a4a89b10ea6df00013e5cf8   2    [...]
    [...]
(master_env) lukas@skybookHP:~/bis$ mars describe run [...]13e5cf0
    Identifier:         5a4a84010ea6df00013e5cf0
    [Plan] Name:        Demo1 (5a4a54af0ea6df00013e5cef)
    Status:             Running
    [...]
    Output:
        [...]
        [AM] Starting creation of agent type: ParenchymalCell
        [...]
        [AM] [...] AgentCount is : 4224
        [...]
        ...done in 2039ms or 00:00:02.0398684
        [...]
        [LIFE] Tick 1 done. Took 3743 ms.
        [...]
        [LIFE] Tick 10 done. Took 3885 ms.
        [LIFE] Tick 11 done. Took 5061 ms.
        [LIFE] Tick 12 done. Took 6612 ms.
        [LIFE] Tick 13 done. Took 9526 ms.
        [LIFE] Tick 14 done. Took 15484 ms.
        [LIFE] Tick 15 done. Took 26781 ms.
        [LIFE] Tick 16 done. Took 58371 ms.
        [LIFE] Tick 17 done. Took 178919 ms.
        [LIFE] Tick 18 done. Took 552396 ms.
        [LIFE] Tick 19 done. Took 2282664 ms.
```

Listing 5.18: Retrieval of simulation run's progress and details

### 5.2.6. Evaluation of the results

In listing lst. 5.19 the model specific analysis (*is_tool create-reports*) is applied on a finished simulation run (of MARS BIS). Basically the analysis can work on unfinished simulation runs. But since it includes all recorded data, agent counts of an finished, but not (yet) fully recorded simulation step can be misleading. Another usage of *time* reveals, that the simulation result report generation takes long and consumes much processor resources of the client. While *real* again is the actual time, the sum of *user* and *sys* specifies the amount of time, in which a processor core handled the observed process. Since the client used for the experiments has multiple logical cores (sec. 5.1.1), *user* plus *sys* can be greater than *real*.

```
1 (master_env) lukas@skybookHP:~/bis$ mars run -n Demo2 -t 18 Demo1
2     [...]
3     5a4bb61f0ea6df00013e5cfe           Succeeded           18
4 (master_env) lukas@skybookHP:~/bis$ time is_tool --config is.cfg \
5 create_reports -o reports 5a4bb61f0ea6df00013e5cfe
6     Created report for run identifier  5a4bb61f0ea6df00013e5cfe
7
8     real    2m51.162s
9     user    5m12.932s
10    sys 2m32.752s
```

Listing 5.19: Generation of a MARS BIS' simulation report

Each result analysis of a simulation run concludes in an archive, which contains the parts of the result report. *Report.md* in the archive's root directory contains metainformation for traceability and refers the images. The digital appendix A.2 contains all report files. Meanwhile the remainder of this section presents only few of them. Figures 5.3 and 5.4 shows, that the virtual infection spreads. First parenchymal cells are getting infected (after tick 2, see fig. 5.4). Then the virus reproduction starts few ticks later. After some further ticks the virtual adaptive immune system begins its response, which results amongst others effects (not shown in this section) into an increase of T-Helper cell agents (fig. 5.6) and finally a growth of the antibody population (fig. 5.5). Figure fig. 5.8 shows the distribution of antibodies in the lymph zone after the last simulation step. Some Antibody agents also reached the infection zone fig. 5.9. Thus they theoretically could start to defeat the infection. But this effect could not be observed by a little longer simulations and observations of a signal, which gets produced at virus and antibody interactions. Meanwhile the agent population's of the innate immune system also just grow (e.g. natural killer cells as shown by fig. 5.10).

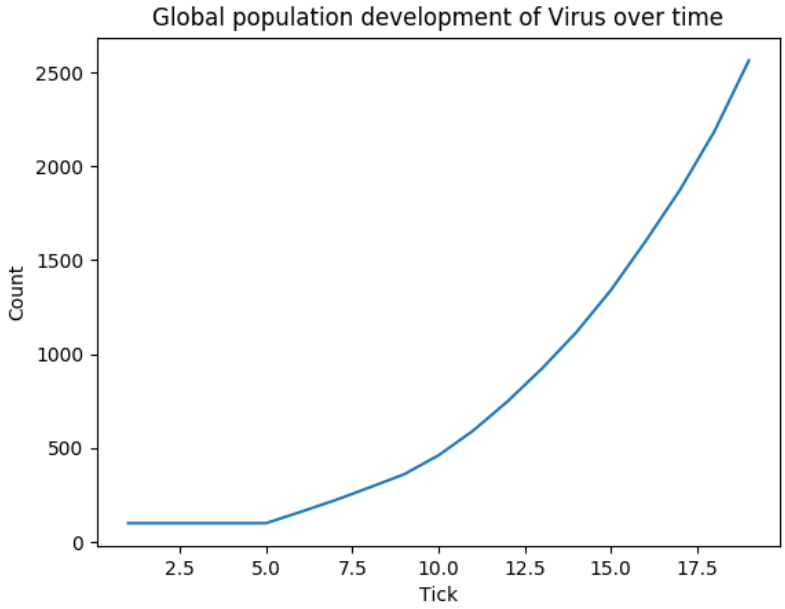


Figure 5.3.: Development of virus population's size

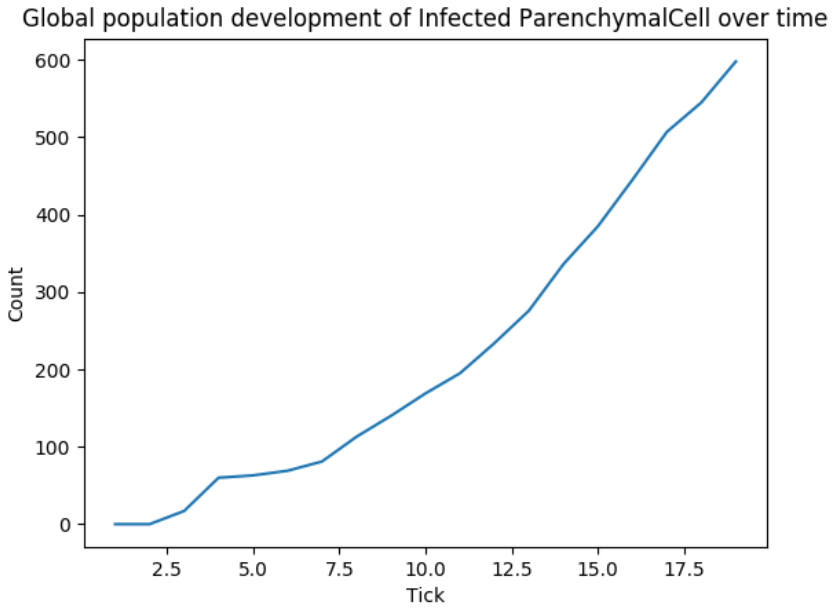


Figure 5.4.: Infected parenchymal cell count over the time

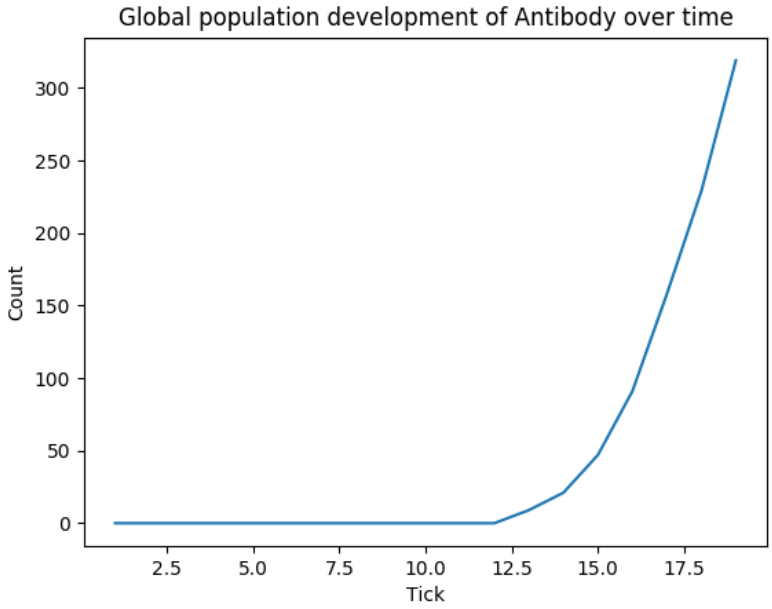


Figure 5.5.: Antibody population's growth

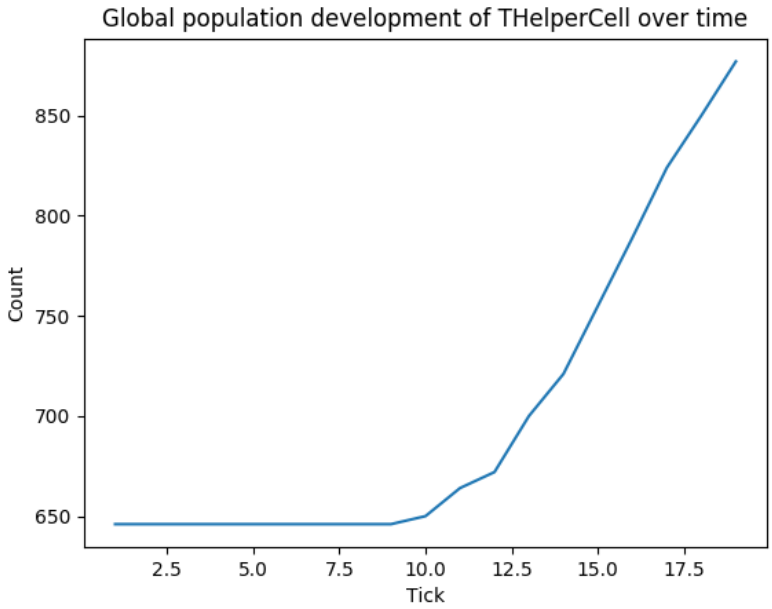


Figure 5.6.: Increase of T-Helper cell agent count

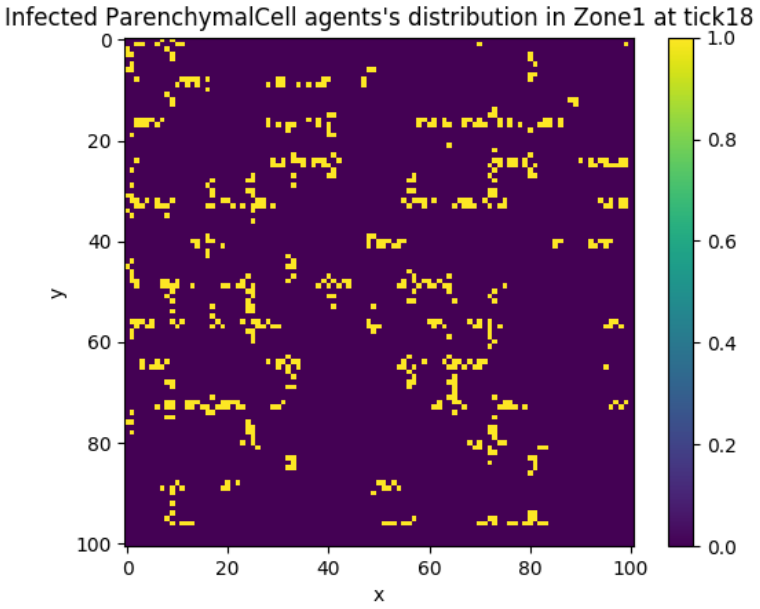


Figure 5.7.: Distribution of infected parenchymal cells in infection zone (1) at 18th tick

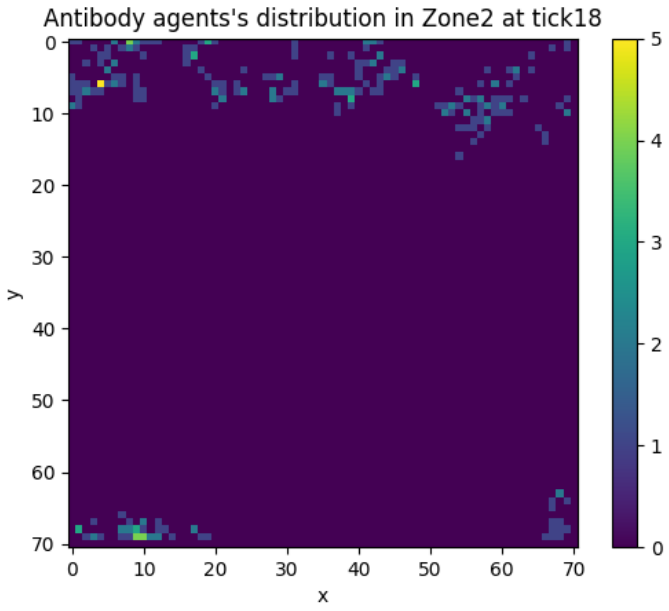


Figure 5.8.: Antibodies in lymph zone at 18th tick

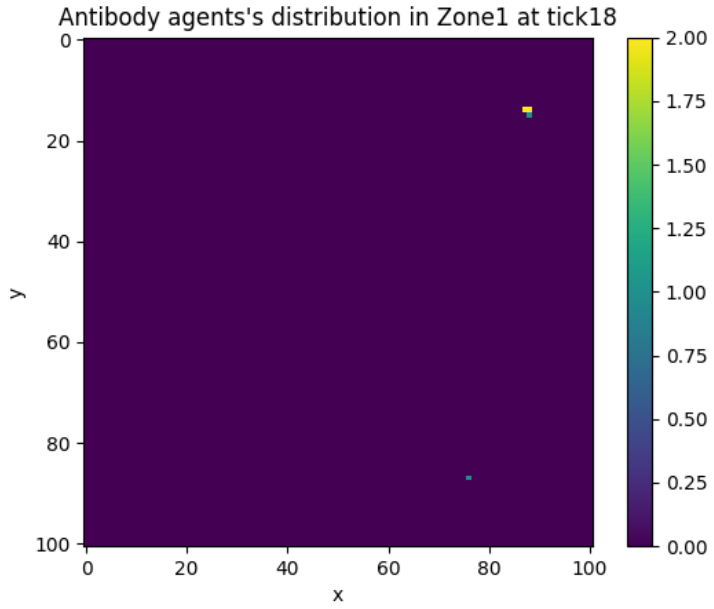


Figure 5.9.: Distribution of antibodies in infection zone at 18th tick

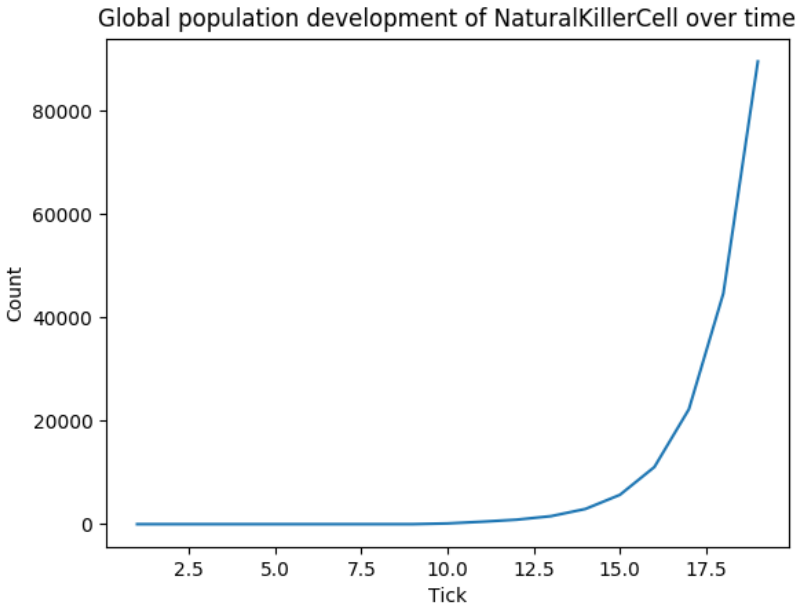


Figure 5.10.: Growth of natural killer cell population

## 5.3. Global sensitivity analysis with MARS

Basically there are two possibles ways to prepare, handle and postprocess a batch evaluation with MARS (e.g. for a sensitivity analysis). The first one uses Python code, which directly can use the *Client library* described by section 4.2.2. Meanwhile if the user wants to involve tools without an interface to Python, the shell compatible wrapper provided through the *MARS CLI* becomes handy, since it translates between contents of files and calls of the *Client library*. This section demonstrates both, the usage of *MARS CLI*'s batch / sensitivity analysis features and calling *Client library* directly. Both ways have the MARS model and the *Experimental Setup* in common and this section starts with describing those, on which the presented sensitiviy analysis example bases.

### 5.3.1. Introduction of the test model

Since the MARS BIS model does not show the expected behaviour in its current state and each evaluation takes a lot of time, the author decided to use a simpler model for the demonstration of a sensitivity analysis with MARS.

Thus the example sensitivity analysis use case bases on a MARS variant of the popular wolves (predator) and sheep (prey) model. It bases on example models for other agent based simulation frameworks, Wilensky (1997) and Sabelli and Kovacevic (2008). Most of the MARS port[9][10] was done by Jan Dalski. The author of this thesis made additions[11] based on Prestes García and Rodríguez-Patón (2016).

The model has three agent types, which represent wolves, sheep and grass. Wolves and sheep wander randomly around and loose energy. As soon the energy drops under a certain hunger level, the agents scan a certain area around their current position for prey. While sheep eat grass, wolves eat sheep. Both target their prey and move towards it. After a wolf or sheep agent successful ate its prey, it gains a certain amount of energy. Meanwhile the prey dies. Wolves and sheep starve, if their energy becomes zero. The death of sheeps and wolves technically is represented by agent removal and grass gets deactivated. The latter activates itself after a certain amount of simulation steps (regrow time), while sheeps and

[9] Sourcecode: https://gitlab.informatik.haw-hamburg.de/mars/model-wolves-and-sheep
[10] Documentation: https://confluence.mars.haw-hamburg.de/display/MP/The+Wolves+Model
[11] Branch *gsa* of sourcecode repository

wolves reproduces themselves each tick with a certain rate. A two dimensional grid represents the spatial environment of the agents.

Following input parameters are equal for all agents:

- Reproduction rate of sheep (*sheep_reproduction_rate*)
- Reproduction rate of wolves (*wolf_reproduction_rate*)
- Amount of energy sheep gain from a grass (*sheep_energy_from_food*)
- Amount of energy wolf gain from eaten sheep (*wolf_energy_from_food*)
- Regrow time of grass (*grass_regrow_time*)

Like Prestes García and Rodríguez-Patón (2016) the sensitivity analysis with MARS (shown in upcoming sections) subjects all five parameters. Though the same ranges, scalar output definitions and sensitivity analysis methods are used as by Prestes García and Rodríguez-Patón (2016), the results can not be compared, since the used models differ. Especially the hunger and the targeting of prey are only present in the MARS version. Meanwhile the wolves' and sheep's initial position and energy amount are random. Each agent uses its own random generator with a *seed*, which an agent parameter provides. Each grid cell hosts a grass agent. Grass agent parameters define the grid cell ($x$, $y$) and if the agent is initially *dead*.

While the later four parameters should allow the sensitivity analysis to create an equal initial state for every model evaluation, MARS undefined execution order probably still causes non-deterministic simulation outcomes.

### 5.3.2. Experimental setup

This subsection covers parts of the *Experimental Setup* for the MARS wolves and sheep model. Meanwhile the full file is available as *gsa/wolves_sheep_experiment.py* (in the *gsa* branch of the model's repository).

The first important part of the *Experimental Setup* are the parameter generator functions. Listing 5.20 shows *get_grass_regrow_time*, which expects the input parameter *grass_regrow_time* as real number between 0 and 1. The function scales the parameter's value to an integer between *20* and *40*, how the model's *Grass* agent expects it. Meanwhile *generate_grass* creates a table, which contains the initial positions and *dead* state of the agents. At the time of writing *MARS CLI*'s batch features does not load generator functions with external dependencies

(e.g. imports, calls of other methods) properly. Thus *generate_grass* (and the not listed generator functions) must import the required module *random* in its body. As described in a previous subsection a fixed *seed* ensures, that every model evaluation has the same initial *Grass* agents. Finally the actual generator *grass_table* bases on *generate_grass* and defines the column names *x*, *y*, *dead*. If their order changes, the argument order of the *add_row* call (line 11) must change accordingly!

```
1  [...]
2  def get_grass_regrow_time(grass_regrow_time: float):
3      return int(20 + 20 * grass_regrow_time)
4
5  [...]
6
7  def generate_grass(data):
8      import random
9      random.seed(1889842121)
10     for x in range(0, 50):
11         for y in range(0, 50):
12             data.add_row(x, y, random.random() < 0.5)
13
14 [...]
15 grass_table = create_table_generator(generate_grass,
16                                        columns=['x', 'y', 'dead'])
17 [...]
```

Listing 5.20: Parameter generator (functions)

Next listing 5.21 shows the definition of the model, the scenario parameters, the mapping and the definition of the agents to be recorded. The *source* (line 2) for the model is sourcecode (*ModelSources*). The environment variable *MODEL_SOURCE* defines the sourcecode's location. Further *ModelSources* defines the (optional) *title* of the final MARS resource (of the model). The *GlobalParameters* are all constants, except the virtual start date and time of the scenario (*SimulationEndDateTime*). The *scalar generator* function *generate_end_time* calculates it based on a parameter, which specifies the number of simulation steps.

While the scaled *gras_regrow_time* is mapped on the argument *regrowTime* of the *Grass* agent type, the *x* column of each grass table is mapped on the *x* parameter. For now the *Experimental Setup* will lead to *N-1* redundant grass tables, where *N* is the number of evaluations. The reason is, that the batch processing calls all generators for all model evaluations (sec. 4.2.2.7) and the

grass table generator uses a the same, fixed random seed as described earlier every time. Thus a possible optimization is to generate the table once and replace *grass_table.columns.x* (line 15) with *Column('x', File('LOCAL_FILE_PATH', mars_tools.clients.DATA_TABLE, 'TITLE'))*, where *File* and *Column* (sec. 4.2.2.5) are both implemented by the same package as *Mapping* (... see source code).

```
wolf_sheep_model = ModelDefinition(
    source=ModelSources(os.environ['MODEL_SOURCES'], title='WSM'),
    global_parameters=[
        GlobalParameter('SimulationStartDateTime', START_DATE_TIME),
        GlobalParameter('SimulationEndDateTime',
                        create_scalar_generator(generate_end_time)),
        GlobalParameter('DeltaT', 1),
        GlobalParameter('DeltaTUnit', 'hours')
    ],
    mappings=[
        Mapping(agent='Grass', parameter='regrowTime',
                data=create_scalar_generator(get_grass_regrow_time)),
        [...]
        Mapping(agent='Grass', parameter='x',
                data=grass_table.columns.x),
        [...]
    ],
    enabled_outputs=[
        EnabledOutput('Sheep'),
        EnabledOutput('Wolf'),
        # EnabledOutput('Grass', 'Dead')
    ]
)
```

Listing 5.21: Definition of model and scenario details

```
def count_wolves(results):
    from mars_tools.clients.mars_client import MARSSimulationRun
    assert isinstance(results, MARSSimulationRun)
    agents = results.results.get_agent_ids('Wolf')
    agents_query = results.results.agent_id.one_of(agents)
    counts = [
        (agents_query & (results.results.tick == tick)).count()
        for tick in range(0, results.current_tick)]
```

```
9     return sum(counts) / float(len(counts))
```

Listing 5.22: Scalar output definition Average number of wolves

For the current wolves and sheep *Experimental Setup* and MARS' result output implementation it is sufficient to enable the recording for *Sheep* and *Wolf* agents (line 19 and 20), since model specific agent properties are not covered by the current scalar output definitions. Listing 5.22 shows the scalar output generator *count_wolves*, which calculates the average count of wolves. In line 4 the function uses the *ResultQueryAdapter* (sec. 4.2.2.4) get all distinct *Wolf* agent identifiers. The following line builds a query object, which matches each result record, whose identifier property references a *Wolf* agent (thus its value must be in previously queried set). Further *count_wolves* iterates over every simulation step (*tick*) and creates a final *count* query for each by combining the identifier query with a *tick* number match. Since the batch processing calls the output generator after simulation finish, the use of *results.current_tick* is a valid source of the number of simulation steps. Finally the last line calculates the average count.

Listing 5.23 shows the final *Experimental Setup* / *ExperimentConfiguration* for the wolves and sheep model. The *count_sheep* method is like *count_wolves*, but for the *Sheep* agents. The idea behind *models* being a list is to support sensitivity analyses, where the model itself is an input factor (see also sec. 2.5.3). But this feature is not fully implemented and actually required at the time of writing.

```
1 wolf_sheep_experiment = ExperimentConfiguration(
2     models=[wolf_sheep_model], outputs={
3         'wolf_count': create_scalar_generator(count_wolves),
4         'sheep_count': create_scalar_generator(count_sheep)
5     })
```

Listing 5.23: Top level definition of an Experimental Setup

### 5.3.3. Preparation of a batch evaluation for a sensitivity analysis

First step is to determine the required input parameters for each model evaluation. *MARS CLI* offers a tool to extract that information from an *Experimental Setup* as shown by listing 5.24. The directory */home/lukas/wsm* (*~/wsm*) is a local copy of *gsa* branch from the MARS Wolves and Sheep repository. The first line defines the local model source location (sec. 5.3.2). The *-f*

flag of *mars batch* specifies the Python file, which describes the *Experimental Setup*. There are further options for the case, that the file specifies multiple setups (see *mars batch -h*). Meanwhile the *variable* subcommand prints the name and the expected type, if the *Experimental Setup* specifies the latter. Further the *-o* flag controls, if optional parameters and their default values are of interest. Finally the flag *–sa-problem* results in a SALib compatible problem description[12], which is further encoded as JSON. The example saves the problem description as *problem.json*. It is now possible to define the boundaries in the file. But since the generator functions (in the example *Experimental Setup*, sec. 5.3.2) does it already, this step is skipped here.

```
1  (master_env) lukas@skybookHP:~/wsm$ export MODEL_SOURCES=$(pwd)
2  (master_env) lukas@skybookHP:~/wsm$ mars batch -f \
3  gsa/wolves_sheep_experiment.py variables -o
4      grass_regrow_time of type float is required
5      number_of_ticks with default value 300
6      [...]
7  (master_env) lukas@skybookHP:~/wsm$ mars batch -f \
8  gsa/wolves_sheep_experiment.py variables
9      grass_regrow_time of type float is required
10     sheep_energy_from_food of type float is required
11     [...]
12 (master_env) lukas@skybookHP:~/wsm$ mars batch -f \
13 gsa/wolves_sheep_experiment.py variables --sa-problem > problem.json
14 (master_env) lukas@skybookHP:~/wsm$ cat problem.json
15     {"bounds": [[0, 1], [0, 1], [0, 1], [0, 1], [0, 1]],
16      "names": ["sheep_energy_from_food", "wolf_energ[..]],
17      "num_vars": 5}
```

Listing 5.24: Query the input parameters using MARS CLI

#### 5.3.3.1. Sampling and batch preparation in Python

The python snippet in listing 5.25 uses the *SALib* to create Morris samples and further initializes a batch evaluation. The *main* method first loads the *problem* from the previously created file (line 19). In the next line *morris.sample* from the external *SALib* generates Morris (sec. 2.5.5.1) compatible samples with a grid level of *10*, $\Delta$ of *5* and *100* elementary effects.

[12] https://github.com/SALib/SALib/blob/master/docs/basics.rst

Meanwhile in line 21, 22 the default *MARS CLI* configuration (sec. 5.1.3) is loaded and a *client* instance is created from it. Then the Python code initializes an *ExperimentEvaluator*, which implements the batch processing logic described in section 4.2.2.7. The *ExperimentEvaluator* bases on the *Experimental Setup* (*wolves_sheep_experiment.wolf_sheep_experiment*) and the *client*. At the time of writing MARS creates a simulation container in the cluster immediately after the creation of a simulation run. Early experiments showed, that this overloads the cluster's control backend[13] in case of many planned model evaluations. As a workaround the author of this thesis enhanced *ExperimentEvaluator*, such that it waits for simulation runs to be finished before creating new ones. The parameter *max_parallel_simulations* defines how many simulation runs each batch evaluation may have at the same time. It has big influence on the cluster resource usage (if each simulation uses at most one calculation node).

Finally *evaluator.initialize_batch_evaluation* builds the model (if required) and prepares all other files, which are not results of the generators. Further it loads the *samples* into a local database. Since *samples* does not provide column names and the batch evaluation does a assign a column on a generator parameter by name, another argument of *initialize_batch_evaluation* takes *problem['names']*. It is important that the order of the provided column names equals the order of the actual columns in sample. The parameter *project* specifies the MARS project, where the batch evaluation manages all its resources. To prevent name collisions each batch evaluation should use its own project. The method *initialize_batch_evaluation* creates a random name, if none is provided. Also it creates the project, if it does not already exists. In case of success the method returns the unique *batch* (evaluation) *identifier*.

```
1  import json, csv
2  from SALib.sample import morris
3
4  from mars_tools.experiments.evaluation import ExperimentEvaluator
5  from mars_tools.clients.mars_client_configuration import \
6  MARSClientConfiguration
7
8  import wolves_sheep_experiment
9
10 PROBLEM_DESCRIPTION_PATH = 'problem.json' # Path of problem description
11
12
```

[13] Originally the whole cluster was overstrained, since created containers actually run after creation. But after adjustments by the administrators the cluster now runs a container only, if enough resources are free.

```
13 def load_problem(path):
14     with open(path) as problem_file:
15         return json.load(problem_file)
16
17
18 def main():
19     problem = load_problem(PROBLEM_DESCRIPTION_PATH)
20     samples = morris.sample(problem, 100, 10, 5)
21     configuration = MARSClientConfiguration.load()
22     client = configuration.create_client()
23     evaluator = ExperimentEvaluator(
24         wolves_sheep_experiment.wolf_sheep_experiment, client,
25         max_parallel_simulations=20)
26     batch_identifier = evaluator.initialize_batch_evaluation(
27         samples, problem['names'], project='Batchproject1')
```

Listing 5.25: Create samples with Python and SAlib

#### 5.3.3.2. Initialize a batch evaluation with MARS CLI and sample files

The *MARS CLI* expects samples as tables, where each row holds the input parameters for a evaluation and the columns correspond to the generator parameters with the same name. Each element of the table is saved JSON encoded. All JSON encoded fields are further stored as comma-seperated values (CSV) with the specifications of table 5.2. Basically *MARS CLI* uses the standard Python modules *json* and *csv* with default format settings. The first table row is a header, which provides the column names.

Table 5.2.: Sample file specifications

| Aspect | Details |
|---|---|
| Delimiter | , |
| Doublequote | the quote character is doubled |
| Escape character | none used |
| Terminator of line | \r\n (carriage return + newline character) |
| Quote character | “ |
| Quoting | fields with special characters |
| Ignore of whitespace after delimeter | no |

| Aspect | Details |
|---|---|
| More requirements | field values must be JSON encoded |

Listing 5.26 shows the first two lines of an example sample file. Listing 5.27 demonstrates how to initialize a batch evaluation with *MARS CLI*. Like prior the option *-f* specifies the *Experimental Setup* definition. The *-p* parameter corresponds to the *project* argument of *initialize_batch_evaluation* (sec. 5.3.3.1), *sample.csv* is the sample file used in the example and the final output (*t9762ac7c_d049_4d2c_bd00_d4f8c69321e6*) the global identifier.

```
"""sheep_reproduction_rate""",[...],"""wolf_reproduction_rate"""
0.1111111111111111,[...],0.3333333333333333
```

Listing 5.26: Example of sample file

```
(master_env) lukas@skybookHP:~/wsm$ mars batch \
-f gsa/wolves_sheep_experiment.py initialize -p BatchDemonstration2 \
samples.csv
    [...]
    INFO:experiments:Build of model WSM finished
    INFO:experiments:(Build and) uploaded all required constant files
    INFO:experiments:Wait on import of: [...]
    INFO:experiments:Created and configured result configuration [...]
    INFO:experiments:Preparation finished
    t9762ac7c_d049_4d2c_bd00_d4f8c69321e6
```

Listing 5.27: Initialization of batch evaluation with the MARS CLI

### 5.3.4. Execution of the batch evaluation

The next step after the initialization of a batch evaluation is to create a batch evaluation worker. Listing 5.28 shows an addition to the Python snippet lst. 5.25 (previous section). The method *resume_batch_evaluation* of *evaluator* on the *batch* evaluation *identifier* proceeds the work, which the database reports as unfinished.

At the time of writing, *resume_batch_evaluation* reads all open tasks greedy into its memory and starts to work on them. The method reports significant progress on a single evaluation to the database. Thus *resume_batch_evaluation* can safely be interrupted and started later again

on the same batch. But multiple instances of the method should not run on one batch at the same time, because both would process equal evaluations (and do rendundant work). Another aspect to be careful about are changes on the *Experimental Setup* definition, while a batch evaluation uses it. The changes will not affect a running *resume_batch_evaluation*, since it already loaded the definition into memory. But changes between interruption and resume of a batch evaluation corrupt results of sensitivity (or similar) analysis, if changes on the (output) generators make the already finished evaluations incomparable to the pending ones.

Listing 5.29 shows, how the *MARS CLI resume*s a batch evaluation (*t9762ac7c_[...]* in the example). Again the *-f* option specifies the *Experimental Setup* definition. The *resume* subcommand basically is a wrapper around *resume_batch_evaluation*. Thus all considerations regarding interruption, one call per batch and changes on the *Experimental Setup* apply on the usage of *MARS CLI*, too.

```
1     [...]
2     batch_identifier = evaluator.initialize_batch_evaluation(
3         samples, problem['names'], project='Batchproject1')
4     evaluator.resume_batch_evaluation(batch_identifier)
```

Listing 5.28: Resume the work on a batch from Python code

```
1 (master_env) lukas@skybookHP:~/wsm$ mars batch \
2 -f gsa/wolves_sheep_experiment.py
3 resume t9762ac7c_d049_4d2c_bd00_d4f8c69321e6
4     INFO:experiments:Loaded definitions from database
5     INFO:root:Resume single evaluations
6     [...]
```

Listing 5.29: Resume the work on a batch with a MARS CLI call

#### 5.3.4.1. Get progress information about batch evaluations

There are multiple ways to get state information about a batch evaluation. *Client Library* uses the standard Python *logging* module. The official Python documentation provides information[14], how to configure the logs. The *MARS CLI* enables logs with verbosity level *logging.INFO* and above. Further it reduces the verbosity of logs by the *requests* library to *logging.ERROR*, since at *logging.INFO* every single HTTP request causes a log.

[14] https://docs.python.org/3/library/logging.html

*MARS CLI*'s *batch state* command provides progress summaries of batch evaluations as shown by lst. 5.30. The parameters are the same as for the *resume* command. Meanwhile the command first shows, how many evaluations are in which state. *CREATED* is the initial state. *PREPARED* means, that all MARS resources for the evaluation except a simulation run are ready in the MARS cloud. *SIMULATING* indicates, that a simulation is ongoing. An evaluation in state *SIMULATION_DONE* waits for the execution of the output generators on the client. *FINISHED* or *FAILED* mean a final successful / failed completion of an evaluation.

The *state* command then prints information about the evaluations in state *SIMULATING*. Finally *state* gives a count of the problems of unfinished evaluations. It is important to keep in mind, that the client repeats tasks of evaluations after errors, which it classifies at temporary (e.g. connection loss between client and MARS Cloud). Thus the problem count of *state* can be treated as indicator, how smooth the whole environment works.

```
(master_env) lukas@skybookHP:~/wsm$ mars batch \
-f gsa/wolves_sheep_experiment.py \
state t9762ac7c_d049_4d2c_bd00_d4f8c69321e6
    INFO:root:Load states from database
    INFO:experiments:Loaded definitions from database
    INFO:experiments:Login at MARS started
    State of batch evaluation t9762ac7c_d049_4d2c_bd00_d4f8c69321e6:
        375 / 600 evaluations are in state CREATED
        185 / 600 evaluations are in state PREPARED
        40 / 600 evaluations are in state SIMULATING
    Details about running evaluations:
        0 uses run 5a50ccfabeef2d0001be2d1a with state Running, 6
          and was started at 14:19:54 01/06/18.19.2018
        1 uses run 5a50ccfb0ea6df00013e6abe with state Running, 3
          and was started at 14:19:56 01/06/18.19.2018
        [...]
    Sum of problems of unfinished evaluations: 0
```

Listing 5.30: State information of batch evaluation with MARS CLI

Listing 5.31 demonstrates the *MARS CLI* command *batch problem*, which gather and shows informations about the recorded problems. Without the flag '–include-finished' the command only shows the problems recorded for evaluations not (yet) in the state *FINISHED*.

```
(master_env) lukas@skybookHP:~/wsm$ mars batch \
-f gsa/wolves_sheep_experiment.py \
```

```
problems t9762ac7c_d049_4d2c_bd00_d4f8c69321e6
    INFO:experiments:Loaded definitions from database
    INFO:experiments:Login at MARS started
    Problems of evaluation 1 with state SIMULATION_DONE
      -  MARSClientHTTPError([...]Service Temporarily Unavailable[...]
    Problems of evaluation 4 with state SIMULATION_DONE
      -  MARSClientHTTPError([...]Service Temporarily Unavailable[...]
    [...]
```

Listing 5.31: Getting problems of batch evaluation run

#### 5.3.4.2. Profile the progress of a batch evaluation

The *Client Library* has a *benchmark* feature, which allows to profile the progress of a batch evaluation. Basically it records the state counts (like shown by *MARS CLI batch state* command) together with a timestamp (absolute seconds in unix time => after 1.1.1970) into a file. Listing 5.32 shows how to integrate the *benchmark* feature into listing 5.28 (and lst. 5.25). An instance of *Benchmark* does the counting and saves the count results into the local file *PATH_OF_PROFILE_FILE*. Meanwhile *resume_batch_evaluation* triggers the count every time the state of a single evaluation changes. It then calls *benchmark.on_evaluation_state_update* and passes a state information object to it.

```
[...]
from mars_tools.experiments.evaluation import Benchmark
[...]
def main():
    [...]
    benchmark = Benchmark(PATH_OF_PROFILE_FILE)
    try:
        evaluator.resume_batch_evaluation(
            batch_identifier, benchmark.on_evaluation_state_update)
    finally:
        benchmark.close()
```

Listing 5.32: Create batch progress profile using Python

Listing 5.33 shows the use of *–statistics-file* option of *MARS CLI*'s *batch resume* command. The option enables the batch profile creation, too.

```
mars batch -f gsa/wolves_sheep_experiment.py \
resume t71443335_2c06_41f8_9b1b_16915fb1e88a \
--statistics-file profile_t71443335_2c06_41f8_9b1b_16915fb1e88a
    [...]
```

Listing 5.33: Creation of progress profile using MARS CLI

The python script lst. A.1 in the appendix uses *matplotlib* to visualize batch progress profiles as line graphs. The example output fig. 5.11 shows the progress of evaluating the *Experimental Setup* of section 5.3.2 with samples created like in listing 5.25. The graph shows that *600* evaluations were processed in about *4,5 hours*. It took about a quarter of an hour to generate and upload all data (*CREATED* line hitting count of zero).

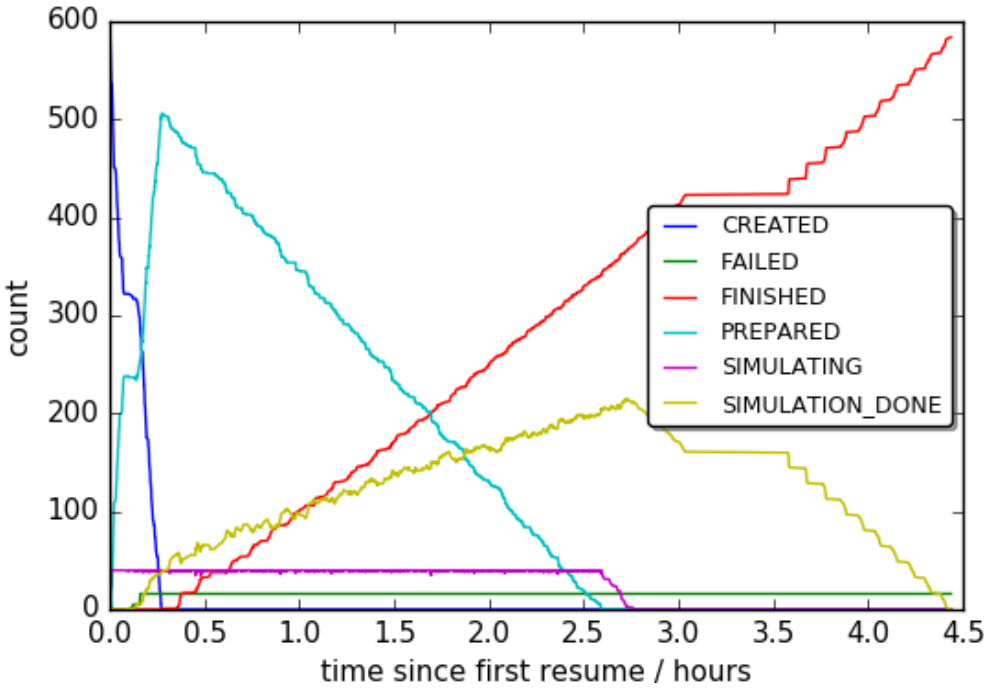


Figure 5.11.: Progress of a batch evaluation

The client kept the count of simulation runs (at the same time) around *40*. The MARS cloud processed finished the last simulation after about *2,75 hours* (*SIMULATING* line reaching *time* axis). Processing the simulation results took significantly longer. The author attached to the container running MARS' result database (using *kubectl* like in sec. 5.1.1) and analyzed

the processor usage with the tool *top*. The result database hit the resource boundaries of its calculation node. Flat *FINISHED* and *SIMULATION_DONE* lines in figure 5.11 (short after third hour) show an batch evaluation interruption by the author. The reason was, that an increasing problem count (gateway timeouts, server side connection closes) indicated an instable MARS cloud. Using *MARS CLI*'s *batch problems* command (lst. 5.31) showed, that the reason for all *FAILED* evaluations were a rather rare type of temporary errors, which the batch processing wrongly categorized as definite problem.

### 5.3.5. Gathering and processing results of the batch evaluation

#### 5.3.5.1. Dealing with failed single evaluations

Certain sensitive analysis setups can cope with missing results of some evaluations. For example if a researcher used a much higher sample number than at least required by the chosen sensitivity analysis method to reach a certain confidence level. In any case the input samples must be dumped, which the not available output values base on.

In the case that a misleadingly problem categorization or a bug in the user defined generators lead to an evaluation failure, another solution is available. *MARS CLI*'s *batch reset-failed* (usage like *resume*, *state* ... before) command resets the state of all failed single evaluations back to *CREATED*. Thus another use of the *batch resume* command on the batch evaluation will give the evaluations another try.

#### 5.3.5.2. Query scalar outputs and postprocessing example

The *ExperimentEvaluator* class' (sec. 5.3.3.1) *get_results* method returns a table (as two dimensional list) with the input samples. Also it returns a map, whose keys are the output names defined by the *Experimental Setup* (e.g. *wolf_count* and *sheep_count* in lst. 5.23). A map's value is a list of the scalar outputs. The method *get_results* guarantees, that:

- The *i*th element of an output list derives from the evaluation of the *i*th row / sample in the input parameter table.
- The order of the returned input samples equals the order of samples, which the batch initialization process (sec. 5.3.3.1) retrieved.
- Input samples and output values of failed evaluations are not part of the results.

Listing 5.34 extends lst. 5.28 by a call of *get_results*. Also it shows, how to pass the method's results to the Morris post processing implementation of the *SAlib* (corresponding to the Morris sampling in the beginning, sec. 5.3.3.1). Passing the *names* values from the *problem* definition as second argument of *get_results* ensures, that the column order of the returned *input_samples* is as *SAlib* expects it (based on the *problem* definition). Listing 5.34 shows parts of an example output of lst. 5.34.

*MARS CLI*'s wrapper command *batch result* for *get_result* merges the output values as additional columns with the input sample table. Then it saves the resulting table in the format, which sec. 5.3.3.2 describes. Listing 5.36 shows an example call.

```
[...]
        benchmark.close()
    input_samples, outputs = evaluator.get_results(
        batch_identifier, problem['names'])
    for name, values in outputs.items():
        indices = morris_post.analyze(
            problem, numpy.array(input_samples), numpy.array(values),
            num_levels=num_of_levels, grid_jump=grid_jump)
        print('Indices for scalar output', name)
        for i in range(0, len(problem['names'])):
            print('of parameter', problem['names'][i] + ':',
                  indices['mu_star'][i], 'with confidence interval',
                  indices['mu_star_conf'][i])
```

Listing 5.34: Result retrieval and postprocessing with SAlib and Morris method

```
Indices for scalar output sheep_count
of parameter sheep_reproduction_rate: 27.831900000000005 [...]
of parameter grass_regrow_time: 33.92208 [...]
of parameter wolf_energy_from_food: 31.563299999999995 [...]
```

Listing 5.35: Output example of listing

```
(master_env) lukas@skybookHP:~/wsm$ mars batch \
-f gsa/wolves_sheep_experiment.py result -o results.csv \
td1349870_2194_4797_8b2f_7facd019977b
    [...]
    INFO:root:Generate result table ...
```

Listing 5.36: Retrieval of results of a batch evaluation using the MARS CLI

# 6. Discussion

This chapter points the main conclusions of this thesis out and suggests possible consequences and improvements.

## 6.1. Retrospective assessment of the original motivation

The current state of the MARS based Basic Immune Simulator (BIS) implementation cannot be subject of a sensitivity analysis, which goal is to identify non-influental model parts for the answer of the originally research question (by (Folcik, An, and Orosz 2007)). Anyway the restrospect shows, that the demand of using the MARS implementation for such model reduction actually does not really exist. Folcik, An, and Orosz (2007) answered their questions with their implementation and functional design. For further analysis about which parts of the agent logic are actually needed, it would probably have been an easier approach to to port the original BIS implementation from Repast J to Repast Symphony. Then the R and Repast solution of Prestes García and Rodríguez-Patón (2016) can be used.

## 6.2. Advantages and disadvantages of MARS

The conclusion of sec. 6.1 renders the original idea about sensitivity analysis of the MARS BIS model senseless. However, both batch evaluation with MARS and the MARS BIS implementation still provide value. MARS' approach to offer simulation as a service in the Cloud has potential. For example it allows even global distributed teams to straightforwardly work on the same experiment. One team member can introduce data, another one can schedule simulations based on them and a third member can evaluate the results. MARS allows the access to all required assets over the internet. Further MARS ability to store all values of every agent's attribute is expensive, but creates also a valuable source of knowledge to debug unexpected end

results. In the case of sensitivity or similar analyses, the extensive results also allow to apply different scalar output definition in hindsight, without the need of redoing the simulations (assuming, that the already existent results are sufficient).

In the meantime, MARS architecture with it stateless services and container based deployment is theoretically highly scalable. However, experiments based on sec. 5.3.4.2 indicated, that in current MARS deployments the single node database setups are a bottleneck for use cases like described in the previous chapter. For another experiment the wolves sheep setup was simplified, such that the model itself counts the agents after each simulation. Then a single instance of a technical agent type exposes the counts over attributes, which MARS records and the scalar output generators read. Also the *Experimental Setup* maps the grass and seed arguments on data files, which are only uploaded once.

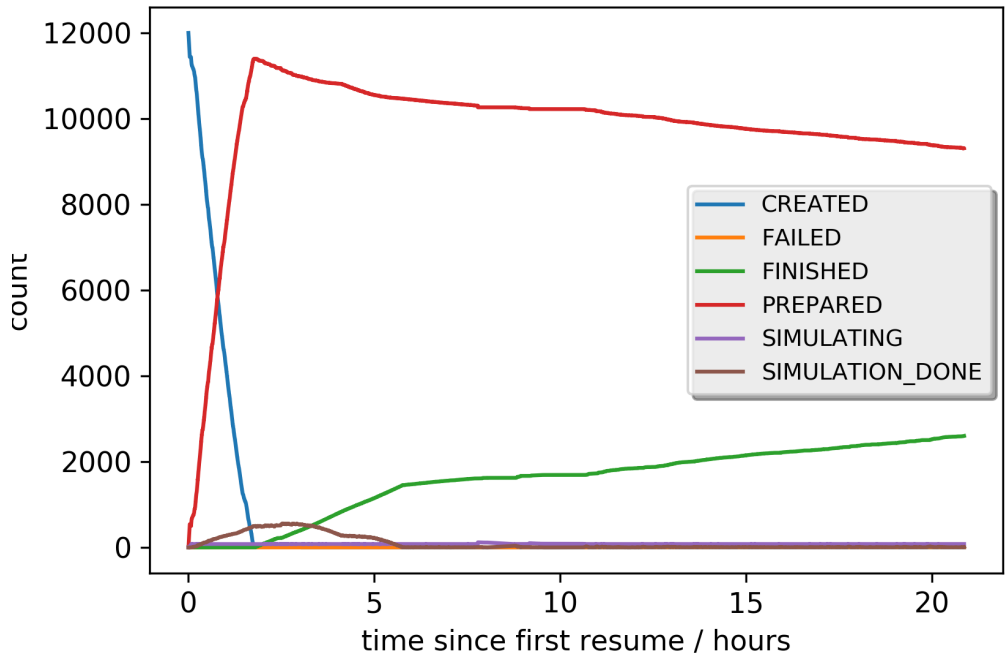


Figure 6.1.: Progress of a batch evaluation based on modificated wolves sheep setup

The lineplot fig. 6.1 visualizes the progress of a batch evaluation (with more samples this time). The results are more preparations in less time. Further the backlog of finished, but not fully evaluated simulation runs decreases to zero over the time. Finally after about six

hours much more evaluations were completly evaluated than with the old setups. At the same time, the random observations showed a much lesser load on the result database container and improved usage of the calculation node's resources. Overall, setups similar to the one described in sec. 5.3.4.2 should profit from a MARS deployment, which have more powerful databases or database clusters.

After six hours another weakness of MARS regarding automated simulations started to show stronger effects. An unreliable simulation monitoring and progress reporting of MARS negatively impacts the batch evaluation's efficiency. At the time of writing, MARS often reports the termination of a simulation long time after it actually happened or even not at all. For instance the output of *mars batch state* reports zero simulated ticks for an evaluation, even multiple minutes after an analysis using *kubectl* shows a successful exit of the related container. Thus cases exist, where the batch evaluation logic needless waits until it runs the output generators. Even worse, the limitation of the maximal number of parallel, active simulation runs (sec. 5.3.3.1) prevents the client from creating new simulation runs, since it believes MARS still has enough outstanding work to do. This finally leads to an underutilization of of the available resources.

## 6.3. Proposed Improvements of the batch evaluation system

The batch evaluation system should be improved for a better user experience. The most crucial aspect is an integration of the batch evaluations features into the MARS Cloud, since it would allow to access *Experimental Setups*, input samples and the final scalar outputs from all over the world. New MARS services would have more abilities for optimisations or to work around the issues like the simulation progress monitoring. For instance until now, uploads of generated data and download of database query results depend on the connection between the client and the MARS Cloud.

The architecture of the batch evaluation is ready to a certain degree for this transition with its processes, which store state informations and resource identifiers into a database. The most crucial point about moving the generators in the MARS Cloud is data isolation. A generator function must only be able to access datasets, for which it has permissions. Thus the author suggests the development of batch worker services in containers, which are network isolated from MARS' backend services and databases. The workers are exchanging tasks, their states and resources over MARS' API services.

The generator framework only provides helper classes for the creation of tables, yet. Additions for the MARS specific layer initialisation files would simplify the development of generator functions.

## 6.4. Future of the Automatisation Service and resource definitions

The author suggests to drop the *Automatisation Service* project, since the client library offers basically all of its features and more. However, a useful, reusable asset should be the resource definition format, which is especially helpful for modelling projects with much input files and mappings. Further it is worth to consider the merge of the *Experimental Setup* (or parts of it) with the resource definitions, since there is redundancy between them.

## 6.5. MARS Teaching UI and Command line interface (CLI)

The MARS CLI has been a side product from this thesis. It offers another user interface to MARS beside the *MARS Teaching UI*. Originally the CLI was only planned as user interface prototype for the batch evaluations and as supplement to be mainly used inside of shell scripts for automatisation purpose (aside from the batch evaluation). In its current state the *MARS Teaching UI* is to insufficient to be used in combination with batch evaluations. It lacks live reloads. To see resources created outside of one *MARS Teaching UI* instance (e.g. by the CLI), the browser has to make a manual reload. Afterwards the user has to reselect the project and resource view, which prohibits an efficient workflow. But more crucial is *MARS Teaching UI*'s lack of paging in resource lists. For instance, since the *MARS Teaching UI* renders all scenarios of a project in its scenario view at once, it takes very long until the browser shows the scenarios created for a typical sensitivity analysis. Especially for mapping the *MARS Teaching UI* is a very intuitive and valuable. Therefore, the MARS group should consider to make it ready for showing great amounts of resources and to refresh the current view, whenever the shown content changes.

# 7. Summary and outlook

Section 2.1 introduces the cells of the adaptive and the innate immune systems and their relations, while sec. 2.2 gives an overview about available agent based immune system models. Based on the previous subections sec. 2.3 describes the original Basic Immune Simulator (BIS), which Folcik, An, and Orosz (2007) used to research the relation between the innate and the adaptive immune system. Subsection 2.4 describes the MARS simulation as a service system, which bases on multiple services running in compute clouds. Further the subsection gives an overview about the MARS libraries, the typical simulation workflow and the ability of MARS to distribute simulations. Section 2.5 describes a typical sensitivity analysis workflow, which consists of the three phases sampling, evaluation and postprocessing. Also the section explains the relation between uncertainty and sensitivity analysis and goals of the latter. Section 2.5 continues with an overview about basic sensitivity analysis approaches, which all do multiple model evaluations with different input parameter sets and multiple scalar outputs. Finally the subsection introduces two comparable frameworks for sensitivity analysis.

Section 3.1 first describes a tool for the usage of the MARS BIS implementation, which transforms an initial configuration file into agent parameter tables. Following subsections gives an overview about the model's components for information management and location management. Another subsection describes the basic, modular agent architecture, which extends the sensor, reasoning and effector paradigm of MARS by internal events. Further the subsection introduces the agents, their interaction and class hierarchy. It also describes the design of the agent's production, their positioning, movement and sensors for other agents. Section 3.1 describes a tool, which creates visual result reports from MARS BIS simulation results.

Section 4.1 analyses that basically a system is required to schedule batch evaluations with MARS. Section 4.2 then suggests a workflow with such batch system for sensitivity analysis and introduces the system's components. One component is a client library, which allows to manage MARS resources. The batch evaluation subsystem bases on the client library. Further it provides a way to define complex input data generation from input parameter sets (e.g. samples)

and to define the aggregation of scalar outputs based on a simulation's results. Further sec. 4.2 introduces a command line utility (CLI), which bases on the client library.

Section sec. 5.1.2 describes how to prepare a client for the usage examples. Then sec. 5.2 gives an overview about the current workflow for single simulations with the MARS BIS model. Further it describes, how to use the MARS BIS specific tools in combination with the CLI for MARS. Listings with descriptions show the preparation, management and result visualization of MARS BIS simulations. Finally sec. 5.2 shows parts of actual results and concludes, that sensitivity analyses with MARS BIS does not make sense in the model's current state. Consequently a sensitivity analysis example defines input and output generators for a modification of the MARS Wolves and Sheep model example (sec. 5.3). Afterwards the sec. 5.3 guides through sampling, model evaluation and postprocessing phase using the example setup and a sensitivity analysis method based on Morris. Further it analyses, how well the batch evaluation system works against the current MARS Beta deployment. The performance analyse also gives a hint, how many simulation throughput is realistic at the time of writing.

Chapter 6 concludes, that the batch evaluation experience with MARS can be improved with a more powerful databases setup. It further explains, how MARS currently unreliable simulation progress reporting restrains efficient usage of the available computing cloud hardware. Further sec. 6 suggests to build services, which integrate the batch evaluation feature more into the MARS cloud system. Finally sec. 6 motivates the enhancement of MARS current graphical user interfaces by live reloading and paging.

The MARS BIS model has reached such a technical state, that its further development direction requires the advise of domain experts from the immunology. Anyway there will be no more modifications of the model to let it behave exactly like the Basic Immune Simulator. Even unfinished it provides generic parts, which might be of use for other modelling projects with MARS.

As soon MARS offers reliable simulation progress information, the batch evaluation system provides a scalable (with the available hardware), technical base for sensitivity analyses of models without much data in-/output. Deploying MARS with more powerful database setups, will improve the experience for data intensive setups.

# A. Appendix

## A.1. Script to lineplot batch progress profile using matplotlib

```
#!/usr/bin/python3

import argparse, csv, os
from matplotlib import pyplot
import matplotlib


matplotlib.rcParams.update({'font.size': 11})


def plot(progress_file, output_path, width, height):
    try:
        with open(progress_file) as opened_file:
            reader = csv.reader(opened_file)
            header = next(reader)
            columns = [list() for x in header]
            for row in reader:
                for index in range(0, len(row)):
                    columns[index].append(float(row[index]))
            columns = {header[i]: columns[i]
                       for i in range(0, len(header))}
            time_column = columns.pop('Time')
            time_column = [(x - time_column[0]) / 3600
                           for x in time_column]
            figure = pyplot.figure(figsize=(width, height), dpi=300)
            plotter = figure.add_subplot(111)
            for name in sorted(columns):
                plotter.plot(time_column, columns[name], label=name)
```

```
29              plotter.set_ylabel('count')
30              plotter.set_xlabel('time_since_first_resume_/_hours')
31              plotter.legend(loc="center_right",
32                             prop={'size': 9}, shadow=True, fancybox=True)
33              figure.savefig(output_path)
34      except IOError as e:
35          print('Could_not_read', progress_file + ':', repr(e))
36
37
38  def main():
39      parser = argparse.ArgumentParser(
40          description='Plots_a_MARS_batch_progress_file')
41      parser.add_argument('progress_file')
42      parser.add_argument('--width', help='width_in_inches',
43                          default=6, type=int)
44      parser.add_argument('--height', help='height_in_inches',
45                          default=8, type=int)
46      parser.add_argument('output_path')
47      args = parser.parse_args().__dict__
48      plot(**args)
49
50
51  if __name__ == '__main__':
52      main()
```

## A.2. Digital appendix on attached compact disc

| Path | Description |
|---|---|
| example_simulation_report/ | full simulation report example from section 5.2.6 |
| progress_profile_to_plot.py | lst. A.1 |

# B. Acknowledgements

My gratefulness goes to Prof. Dr. Thomas Clemen, who motivated me to join the MARS group. While he supervised the major part of my master studies, he gave me the freedom and provided methodical knowledge to dive into interdisciplinary topics. Further he helped me with reducing the uncertainties of my selfmade goals.

Also I thank Prof. Dr. Philipp Jenke and Prof. Dr. Stefan Sarstedt for their advises, how to systematically acquire and present the theoretical background about agent based simulations in the immunology.

I'm also very thankful to Prof. Dr. Greg Kiker for a first introduction of the sensitivity analysis.

Christian Hüning introduced me in several new technologies and worked with me on introducing them into the MARS system. Meanwhile we had very interesting discussions about computer science topics. I'm very thankful for all of our common projects.

I also thank the rest of the MARS team for their constructive collaboration and discussions. Also many "bullshit" talks always were welcome to sidetrack myself from brain damaging topics and cheered me up. The kangaroo will never forget the precious time, you gave him/her/it.

My coworkers and supervisors at my working place always showed understanding for stressful study phases.

The unconditional, altruistic support through my friends, family and parents was a big help overall, contributed to my success and is priceless. I thank them all very much.

# Bibliography

Andrew Emerson, Elda Rossi. 2007. "ImmunoGrid - the Virtual Human Immune System Project." *Studies in Health Technology and Informatics* 126: 87–92.

Bernaschi, M., and F. Castiglione. 2001. "Design and Implementation of an Immune System Simulator." *Computers in Biology and Medicine* 31 (5): 303–31. doi:10.1016/S0010-4825(01)00011-7.

Borgonovo, E. 2007. "A New Uncertainty Importance Measure." *Reliability Engineering & System Safety* 92 (6): 771–84. doi:10.1016/j.ress.2006.04.015.

Dalski, Jan, Christian Hüning, and Thomas Clemen. 2017. "An Output and 3D Visualization Concept for the Msaas System Mars." In *Proceedings of the 2017 Spring Simulation Multiconference*. ADS '17. Virginia Beach, Virginia, USA: Society for Computer Simulation International. `http://dl.acm.org/citation.cfm?id=3106078.3106079`.

Folcik, Virginia A., Gary C. An, and Charles G. Orosz. 2007. "The Basic Immune Simulator: An Agent-Based Model to Study the Interactions Between Innate and Adaptive Immunity." *Theoretical Biology and Medical Modelling* 4 (1): 1–18. doi:10.1186/1742-4682-4-39.

Helton, J. C., J. D. Johnson, C. J. Sallaberry, and C. B. Storlie. 2006. "Survey of Sampling-Based Methods for Uncertainty and Sensitivity Analysis." *Reliability Engineering & System Safety*, The Fourth International Conference on Sensitivity Analysis of Model Output (SAMO 2004), 91 (10): 1175–1209. doi:10.1016/j.ress.2005.11.017.

Hüning, Christian. 2016. *Analysis of Performance and Scalability of the Cloud-Based Multi-Agent System MARS*. Hamburg.

Hüning, Christian, Mitja Adebahr, Thomas Thiel-Clemen, Jan Dalski, Ulfia A. Lenfers, Lukas Grundmann, Janus Dybulla, and Gregory A. Kiker. 2016. "Modeling & Simulation as a Service with the Massive Multi-Agent System Mars." In *Proceedings of the Agent-Directed Simulation*

*Symposium*, 8. ADS '16. San Diego, CA, USA: Proceedings of the 2016 Spring Simulation Multiconference. http://dl.acm.org/citation.cfm?id=2972193.2972194.

Minunno, F., M. van Oijen, D. Cameron, and J. Pereira. 2013. "Selecting Parameters for Bayesian Calibration of a Process-Based Model: A Methodology Based on Canonical Correlation Analysis." *SIAM/ASA Journal on Uncertainty Quantification* 1 (1): 370–85. doi:10.1137/120891344.

Morris, Max D. 1991. "Factorial Sampling Plans for Preliminary Computational Experiments." *Technometrics* 33 (2): 161–74. doi:10.2307/1269043.

Neumann, Jürgen. 2008. *Immunbiologie - Eine Einführung*. Berlin, Heidelberg: Springer-Verlag.

North, Michael J., Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. 2013. "Complex Adaptive Systems Modeling with Repast Simphony." *Complex Adaptive Systems Modeling* 1 (1): 3. doi:10.1186/2194-3206-1-3.

Nossent, Jiri, Pieter Elsen, and Willy Bauwens. 2011. "Sobol' Sensitivity Analysis of a Complex Environmental Model." *Environmental Modelling & Software* 26 (12): 1515–25. http://www.sciencedirect.com/science/article/pii/S1364815211001939.

Pianosi, Francesca, and Thorsten Wagener. 2015. "A Simple and Efficient Method for Global Sensitivity Analysis Based on Cumulative Distribution Functions." *Environmental Modelling & Software* 67 (Supplement C): 1–11. doi:10.1016/j.envsoft.2015.01.004.

Pianosi, Francesca, Keith Beven, Jim Freer, Jim W. Hall, Jonathan Rougier, David B. Stephenson, and Thorsten Wagener. 2016. "Sensitivity Analysis of Environmental Models: A Systematic Review with Practical Workflow." *Environmental Modelling & Software* 79 (Supplement C): 214–32. doi:10.1016/j.envsoft.2016.02.008.

Prestes García, Antonio, and Alfonso Rodríguez-Patón. 2016. "Sensitivity Analysis of Repast Computational Ecology Models with R/Repast." *Ecology and Evolution* 6 (24): 8811–31. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5192867/.

Rapin, Nicolas, Ole Lund, and Filippo Castiglione. 2011. "Immune System Simulation Online." *Bioinformatics* 27 (14): 2013–4. doi:10.1093/bioinformatics/btr335.

Rapin, Nicolas, Ole Lund, Massimo Bernaschi, and Filippo Castiglione. 2010. "Computational Immunology Meets Bioinformatics: The Use of Prediction Tools for Molecular Binding in the

Simulation of the Immune System." *PLoS ONE* 5 (4): e9862. doi:10.1371/journal.pone.0009862.

Sabelli, Hector, and Lazar Kovacevic. 2008. "Biotic Complexity of Population Dynamics." *Complexity* 13 (4): 47–55. doi:10.1002/cplx.20210.

Saltelli, Andrea, Paola Annoni, Ivano Azzini, Francesca Campolongo, Marco Ratto, and Stefano Tarantola. 2010. "Variance Based Sensitivity Analysis of Model Output. Design and Estimator for the Total Sensitivity Index." *Computer Physics Communications* 181 (2): 259–70. doi:10.1016/j.cpc.2009.09.018.

Saltelli, Andrea, M Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. 2008. *Global Sensitivity Analysis. The Primer*. Vol. 304. John Wiley & Sons, Ltd.

Saltelli, Andrea, Stefano Tarantola, Francesca Campolongo, and Marco Ratto. 2004. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. New York, NY, USA: Halsted Press.

Sarpe, Vladimir, and Christian Jacob. 2013. "Simulating the Decentralized Processes of the Human Immune System in a Virtual Anatomy Model." *BMC Bioinformatics* 14 (Suppl 6): S2. doi:10.1186/1471-2105-14-S6-S2.

Schütt, Christine, and Barbara Broeker. 2009. *Grundwissen Immunologie*. Auflage: 2. Aufl. Spektrum Akademischer Verlag.

Song, Liyan, Yunbo Guo, Qingkai Deng, and Jinming Li. 2012. "TH17 Functional Study in Severe Asthma Using Agent Based Model." *Journal of Theoretical Biology* 309 (September): 29–33. doi:10.1016/j.jtbi.2012.05.012.

Spear, R. C., and G. M. Hornberger. 1980. "Eutrophication in Peel Inlet—II. Identification of Critical Uncertainties via Generalized Sensitivity Analysis." *Water Research* 14 (1): 43–49. doi:10.1016/0043-1354(80)90040-8.

Tay, Joc Cing, and Atul Jhavar. 2005. "CAFISS: A Complex Adaptive Framework for Immune System Simulation." In *Proceedings of the 2005 ACM Symposium on Applied Computing*, 158–64. SAC '05. New York, NY, USA: ACM. doi:10.1145/1066677.1066716.

Wendelsdorf, K.V., M. Alam, J. Bassaganya-Riera, K. Bisset, S. Eubank, R. Hontecillas, S. Hoops, and M. Marathe. 2012. "ENteric Immunity SImulator: A Tool for in Silico Study of Gastroenteric

Infections." *IEEE Transactions on NanoBioscience* 11 (3): 273–88. doi:10.1109/TNB.2012.2211891.

Wilensky, U. 1997. "NetLogo Wolf Sheep Predation Model." Center for Connected Learning; Computer-Based Modeling, Northwestern University, Evanston, IL. `http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation`.

Wu, Sichao, Henning S. Mortveit, and Sandeep Gupta. 2017. "A Framework for Validation of Network-Based Simulation Models: An Application to Modeling Interventions of Pandemics." In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 197–207. SIGSIM-PADS '17. New York, NY, USA: ACM. doi:10.1145/3064911.3064922.

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

Hamburg, 12.1.2018 Lukas Grundmann