Content

CONTENT ... **i**

ABSTRACT ... **ii**

1. INTRODUCTION ... **1**

2. FUNDAMENTALS AND RELATED WORK ... **2**

2.1. FUNCTIONAL SIMULATION TOOLS ... 5
2.2. STRUCTURAL SIMULATION TOOLS ... 8
2.3. SIMULATION OF BIOLOGICAL SYSTEMS ... 17

3. REQUIREMENTS ... **19**

4. DESIGN ... **24**

4.1. SUBSYSTEM CONNECTION ... 24
4.2. SUBSYSTEM HANDLE ... 26
4.3. SUBSYSTEM CONTROL ... 29
4.4. TRANSPORT OF VALUES ... 31
4.5. MAPPING ... 35
4.6. LOG AND STATISTIC UNIT ... 41
4.7. GLOBAL CONTROL ... 46

5. IMPLEMENTATION ... **50**

5.1. GENERAL IMPLEMENTATION DETAILS ... 50
5.2. GLOBAL CONTROL DETAILS ... 51
5.3. GROIMP INTERFACE ... 56

6. RESULTS ... **58**

6.1. EXAMPLE: ABC MODEL ... 59
6.2. EXAMPLE: ROOT DEVELOPMENT ... 65

7. DISCUSSION ... **73**

7.1. IN COMPARISON TO EXISTING TOOLS ... 73
7.2. PERFORMANCE ... 74
7.3. OUTLOOK ... 75

8. CONCLUSION ... **77**

APPENDIX ... **78**

A. CD-ROM ... 78

ACKNOWLEDGMENT ... **79**

LIST OF FIGURES ... **80**

LIST OF TABLES ... **83**

BIBLIOGRAPHY ... **84**

Abstract

In Silico simulation of biological systems is an important sub area of computational biology (system biology), and becomes more and more an inherent part for research. Therefore, different kinds of software tools are required. At present, a multitude of tools for several areas exists, but the problem is that most of the tools are essentially application specific and cannot be combined. For instance, a software tool for the simulation of biochemical processes is not able to interact with tools for the morphology simulation and vice versa. In order to obtain realistic results with computer-aided simulations it is important to regard the biological system in its entirety. The objective is to develop a software framework, which provides an interface structure to combine existing simulation tools, and to offer an interaction between all affiliated systems. Consequently, it is possible to re-use existing models and simulation programs. Additionally, dependencies between those can be defined. The system is designed to interoperate as an extendable architecture for various tools. The thesis shows the usability and applicability of the software and discusses potential improvements.

1. Introduction

The importance of modelling and simulation in natural sciences grows gradually. Especially in biology, simulation tools become an inherent part of the work. Scientists use a high variation of different tools to assist their daily scientific work. Beside rather support-orientated tools, like for instance data management systems, document management systems or virtual lab books, biologists use more and more simulation tools as an alternative to real experiments. Additionally, the simulation tools are used in parallel to the experiments in order to control and assist the models and the experiments in the same way.

The invention of powerful computer systems and new programming paradigms allows creating powerful, but also complex software systems. These tools enable the users to create more realistic models, which can use new calculation intensive simulation methods. Some of these tools are able to solve complex mathematical equations, which are used in biochemical models, and others allow the creation of realistic looking 3D simulations.

Before a scientist can use these new modelling and simulating techniques, he has to understand what is necessary to know about the real system to transfer the real world into a model. A model is always an abstraction and a concentration to the important parts of a real system (world). Modelling a real system means to understand the system and to identify and to categorise the mayor parts and dynamics. To model and simulate a biological system it is indispensable to have a detailed knowledge about the following systems parts:

- biochemical processes
- physical structure
- dynamic of the system
- interaction with the environment

In the last decade, quite a few tools have been invented to model and simulate biochemical processes as well as the structure of biological organism. To simulate a complex system like a tree, a flower, an organ or an entire ecosystem it is inadequately to simulate the biochemical process separately from the structure. As mentioned before, an entire simulation is only possible when all four mentioned parts are combined in one simulation. Most of these invented tools simulate only the functional or structural part of the system, and currently, only some approaches exist which are able to combine the different parts into one simulation. Usually, scientists use hard coded simulation tools to simulate an entire system.

The goal of this thesis is to present a new simulation framework approach, which is able to reuse existing simulation tools and combine the parts (simulation tools) to an entire simulation. The framework shall be able to interact with the connected tools, and allows the administrator to define dependencies between the different models.

2. Fundamentals and Related Work

Fundamental research in biology has concentrated for a long time on specific parts of a biological system. Biologists have studied the molecular biology of a single cell and have investigated how parts of the cell work. They have concentrated on DNA replication as well as on transcription, DNA, RNA, protein and membrane structure and function. The invention of new technology, like fluorescence labelling, sequence analysis, electron microscopes or microarray analysis revolutionise the research. A well known example for the capability of new methods is the sequencing of the human genome, known as the human genome project [Ven01].

Traditional techniques for the storage of experimental results (data), e.g. laboratory notebooks, simple text files or spreadsheets, are not adequate for the multitude of data which are now available and accrue every day. To handle this data it has been necessary to collaborate with other natural science and use computer science techniques and methods. With the aid of this interconnection, scientists can concentrate more and more on the investigation of complex systems and interaction between elementary elements. For instance, cells, organs, organism and how cellular processes are regulated as well as reactions of changes in the environment. This new field is called in literature system biology or computational biology [Kli05].

One major task of system biology is to develop tools and algorithm for the simulation and modelling of biological systems. Like in other natural sciences such as physics, where modelling and simulation plays an important part of research, theoretical approaches for the simulation of complex systems have been developed. The models and simulations are based on mathematical concepts like differential equations, net theory, Markov processes and stochastic processes algebra[1]. For example, metabolic network can be modelled as a Petri net [Red96]. Also, it is possible to analyse metabolic pathways using high-level Petri nets [Vos03]. Common tools use ordinary differential equations (ODE) for the simulation of metabolic pathways. Therefore, the system transfers the metabolic network in a set of equations which can be solved [Gor99]. More information about modelling and simulation can be found in [Kli05] and [Wil06].

The knowledge we have about a biological system results from experiments. Additionally, a model can only be as good as the knowledge about the real system. Consequently, it is necessary to provide the information in an easy and standardised way. For this reason, data integration plays an important role here. The majority of data integration tools are data warehouse tools, which based on relational or object-oriented data bases, and allow to integrate and to visualise diverse biological data sets. Currently, a few open source and commercial systems exist. The *ONDEX* system [Koe06], developed at the bioinformatics group at Rothamsted Research Institute[2], use data mining techniques to extract information out of existing biological data bases, and text mining techniques to extract information out of free text. The system combines, with the help of data integration techniques, information in a graph-based data format and provides a visualisation and

[1] For instance, PEPA; http://homepages.inf.ed.ac.uk/stg/research/SIGNAL/; accessed 02.07.2007
[2] http://www.rothamsted.ac.uk; accessed 02.07.2007

analysis functionality. Another open source tool is *BN++*[3], developed at the University of Saarland [BN+07]. Companies usually offer a combination of data warehouse tools and molecular biological tools or services. *Biobase*[4] for instance, provides tools and data warehouse systems for transcription factors, gene regulatory networks, microarray analysis and proteomics [BIB07].

Data integration cannot explain the dynamic of the system or replace the mathematical modelling, but it helps to find the necessary information and dependencies between the elements. It can avoid the problem of using erroneous data or missing important side effects. Errors can occur for instance, when the gene identifiers are misspelled or different names for the same gene are used, which arises frequently.

To explain why models and simulations in biology are inevitable, it is indispensable to define what a model is. Klipp et al. [Kli05] define a model in the following way, 'In the broadest sense, a model is an abstract representation of objects or processes that explains features of these objects or processes'. In other words, a model is a simplified and abstracted view of a part of the real world, concentrated on the principal constituents. A model is the basic prerequisite for a simulation of a system.

The advantages of computational modelling and simulation can be divided into two groups:

The first group contains the model's obvious vantages in comparison to traditional experiments. Modelling is cheap compared to real experiments. Each model is reusable, whereas experiments are abdicated to several conditions like weather, materials or chemical products. Simultaneously, the model is independent from the real objects and causes no harm on animals or plants. In addition, no interaction with the environment nor with the modelled system takes place. Consequently, no falsification of results by unintentional interactions is feasible. The experimental time can be compressed or enlarged in the model at discretion. Thus, a model is able to simulate different scenarios with varying parameters in a shorter time then a real experiment. Moreover, a computational model (simulation) produces more measured data and precise time series. In contrast to a simulation, in real experiments the scientist has to be concentrated on a couple of values to measure without disturbing the complete measurement.

In the second place, modelling demands uniqueness in the problem specification. Problem, hypothesis and general conditions must be defined in an unambiguously way in advance. Hence, modelling helps the scientist to follow a more structured approach. Finally, a model can help to point out gaps in the knowledge or understanding, which can be a trigger for new fundamental research. The relation between model and real experiment, and how to create a new model, will be discussed in the next paragraph.

The development process for a biological model is almost identical as in other natural sciences. First of all, creating a model is a complex iterative process. Generally, a model relies on the data of experiments as well as the scientists' experience. To give a first impression of the complexity of model development, I will present a simplified common modelling workflow:

[3] http://fred.bioinf.uni-sb.de:9180/BNPP; accessed 02.07.2007
[4] http://www.biobase-international.com/pages/; accessed 02.07.2007

1. **Problem specification:** In the first place, the problem must be specified and it must be clear which questions shall be answered with the model. Instantly, a hypothesis in written form shall be defined.

2. **Evaluate available information:** Use data integration tools and other sources to collect and evaluate all available information. Maybe similar models exist or parts of other models can be reused.

3. **Choose model type:** Depending on the available information in point two, choose the structure of the model. The model can be a deterministic or stochastic system. In addition, choose the level of abstraction and the usage of either continuous or discrete variables. Subsequently, select the underlying mathematical formalism.

4. **Create initial model:** During the creation process of the model it is often necessary to concretise the hypothesis or to collect more data.

5. **Verification of the model prediction:** Try to verify the results of the model with a real experiment.

6. **Redefine the initial model:** Generally, the model and the experiment results disagree. Consequently, the points one to six must be repeated in an interactive procedure in order to find out wherefrom the disagreements derive. Hence, a model can be a trigger for new fundamental research or new experiments. After one iteration step and adaptation of the initial model the agreement should improves.

Figure 1 shows a flowchart of the iterative model creation process.

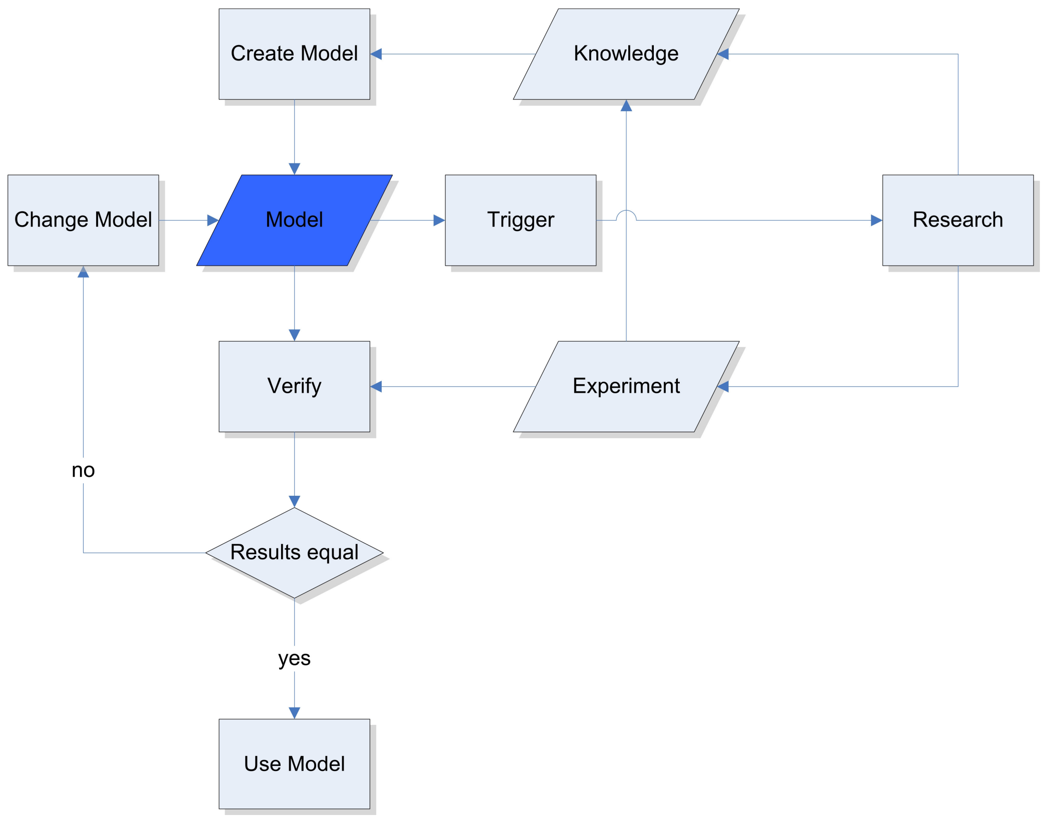


Figure 1: Iterative model creation process.

The major focus of system biology lies on the simulating of biological systems *in silico*. Therefore, various software tools are developed currently. These tools can be classified in functional and structural tools.

2.1. Functional simulation tools

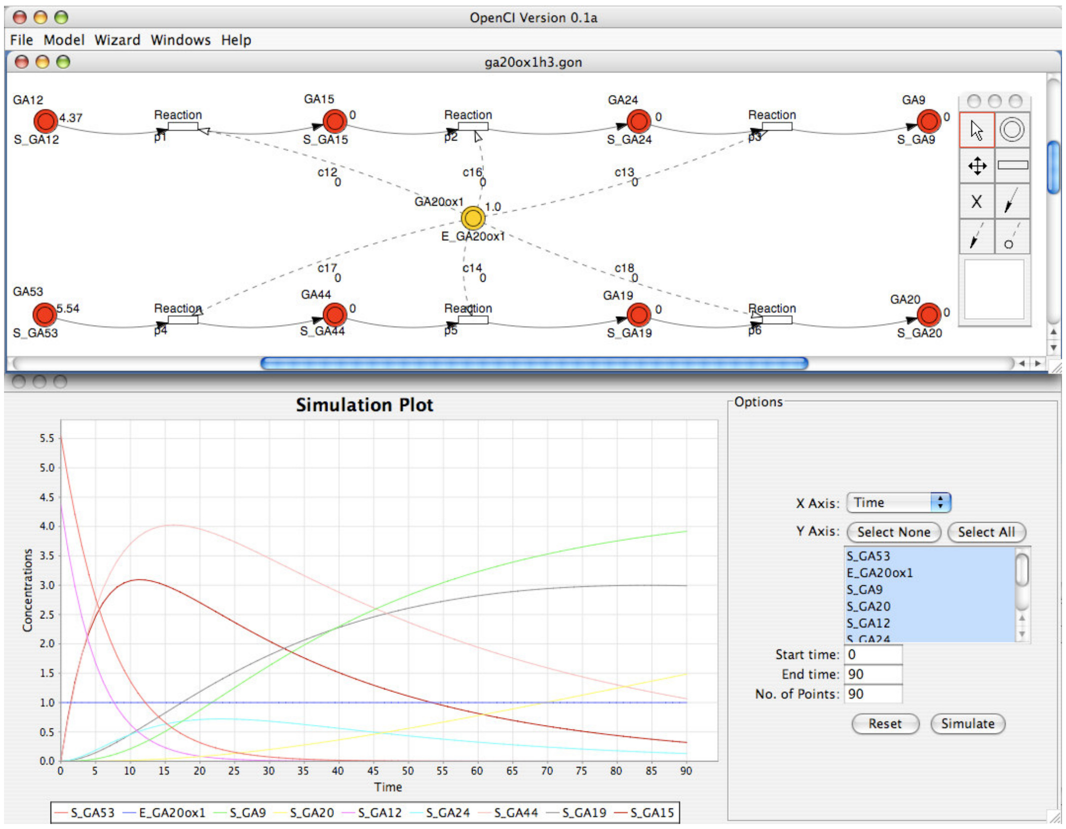
Different categories of functional simulation tools exist. The first category comprises standard mathematical tools beside high-level programming languages and special modelling languages. First of all, scientists search for biochemical networks in tools like *ONDEX*, biological databases and other sources of information. In the next step, they develop the models on paper and transfer those into a set of equations. At least, they use tools like *MATLAB* ®[5], *Mathematica* ®[6] or the *R*[7] environment to solve those equations. Alternatively, scientists use high-level programming languages in combination with open source packages, for instance the *Open Source Physics*[8] project, which provides several algorithms to solve equations. Additionally, numerous modelling languages exist. For one, *PRSIM*[9] allows the definition of systems of concurrent processes. When the processes are synchronised, continuous time Markov chains can be simulated. Muffy Calder et al. [Cal06] for instance, use continuous time stochastic logic and the *PRSIM* language to model a signal transduction network with an example of the *RKIP* inhibited *ERK*[10]pathway. They compare this stochastic modelling approach with an ordinary differential equation model, using *MatLab*® [Cal06].

The second category contains more biological specific systems, which combine stochastic and ODE simulation capabilities with a tool associated control language. In addition, these tools allow defining the models, analysing them and represent the results in one environment with often a graphical user interface. For example, tools like Jarnac, roadRunner or Dizzy. Jarnac [Saur00] developed from Sauro et al.[11] is a scripting environment, implemented in C++, and only available for Win32. The tool uses to solve ODE systems the popular *CVODE* [Coh96] or *LSODA* [Hin83] integrator, which can be selected individually by the user. For stochastic simulations the tool uses an implementation of the *Gillespie* [Gil76] algorithm. Later on Sauro et al. developed a nearly platform independent C# based software tool, called roadRunner. Unlike Jarnac, this tool compiles the models dynamically instead of interpreting them. RoadRunner uses for steady state analysis the integrator *CVODE* and *NLEQ* [Kon07]. Furthermore, Stephen Ramsey [Ram05] developed a tool for stochastic simulations based on *Gillespie*, *Gibson-Bruck* [Gib99] or *Tau-Leap* [Man06] algorithm, called Dizzy.

Graphical modelling environment tools are combined in category three. In contrast to previous presented tools, these tools allow to draw the biological network directly in the tools and to select the corresponding kinetic laws. The first

[5] http://www.mathworks.com/
[6] http://www.wolfram.com/
[7] http://www.r-project.org/
[8] http://www.opensourcephysics.org/
[9] http://sato-www.cs.titech.ac.jp/prism/
[10] In place, only a brief overview, more details are presented in [Cal06]. The ERK pathway (also known as Ras/Raf or Raf-1/MEK/ERK pathway) describe the signals between cell membrane and the nucleus. The protein RKIP inhibits the activation of Raf.
[11] http://128.208.17.155//labmembers.htm; accessed 02.07.2007

tool, *JDesigner* [Saur07], developed at the Keck Graduate Institute[12], uses a model of hyper-graphs to visualise the biochemical networks. The tool allows the user to select predefined rate laws (between the different species) or the user is also enabled to define new rate laws. In addition, the tool uses the functionality of Jarnac or roadRunner for time-course simulation and steady state analysis via the System Biology Workbench (SBW), which will be described in detail in the next paragraph. A further tool for modelling and graphical representation of biochemical networks is CellDesigner, which uses a process diagram [Kit05] notation. CellDesigner uses in the same way as JDesigner Jarnac or roadRunner, via the *SBW* interface, for the simulation. The tools in the next category based on Hybrid Functional Petri Nets with extension[13] [Nag04+]. This net class enables the user to model rule based biological processes in bio-pathways, e.g. gene regulations as well as ODE-based kinetics. Cell Illustrator [Nag04] has been developed by Masao Nagasaki at the University of Tokyo. Artem Lysenko and Tully Yates, from the Rothamsted Research Institute[14], have created an open source tool, named OpenCI, which uses continuous Petri Nets to model the systems. A screenshot of a prototype implementation is shown in Figure 2. The illustration presents a GA20 oxidase pathway with a time course simulation via SBW and roadRunner.



X Axis: Time
Y Axis: S_GA53, E_GA20ox1, S_GA9, S_GA20, S_GA12, S_GA24
Start time: 0
End time: 90
No. of Points: 90

Figure 2: OpenCI: GA20 oxidase simulation with time course simulation via SBW and roadRunner.

[12] http://www.kgi.edu/; accessed 02.07.2007

[13] The extension is necessary to model and simulate more complicated biopahtway processes. The extensions are: first, an entity should contain more than one value, such as list or pair. Secondly, HFPN should handle other primitive types, e.g. boolean, string. Thirdly, the net should handle complex types, like objects.

[14] http://www.rothamsted.ac.uk; accessed 03.07.2007

Hitherto, all presented tools are more or less independent. Accordingly, each tool has to implement its own equation solver, network visualiser, data format et cetera. In regards to the independent data formats, defining a model in tool ‘A’ and reuse the same model in tool ‘B’ require a complete remodelling in tool ‘B’. Hence, data exchange formats have been developed to reduce this problem. Currently, two exchange formats, based on XML, are almost standard. First of all, the System Biology Markup Language (SBML)[15] is supported currently by over 110 software systems [Fin03]. Besides, the Cell System Markup Language (CSML)[16] exists. Both data formats allow representing metabolic, signalling and genetic regulatory pathways. These exchange formats allow the scientists to model a system in different tools, without a direct interaction between these independent tools. Therefore, the System Biology Workbench[17] (SBW) has been developed by Frank T. Bergmann and Herbert M. Sauro at the Keck Graduate Institute, Claremont, USA. SBW is a modular framework, based on broker architecture, connecting existing modelling and simulation tools. This approach allows a direct communication between tools and a reuse of existing functionality, e.g. equation solvers or specific algorithms. The broker controls the communication, based on a message system over TCP/IP and provides binding libraries for the most common programming languages (Figure 3).

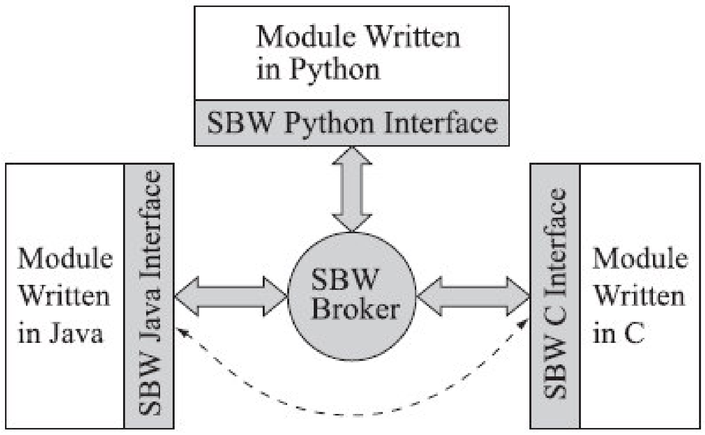


Figure 3: The individual models communicate via the SBW broker (from [Huc07]).

Moreover, the Bio-Spice[18] framework provides a similar functionality like the SBW framework and uses the SBML format also as exchange format between the different tools [Gar03]. The core of the system, written in Java, is called ‘Dashboard’ and allows the combination of connected tools via a graphical user interface to tool chains.

In conclusion, SBW as well as Bio-Spice affords the reuse and the combination of existing tools. However, not one of the frameworks offers the functionality to map parts of the simulation during runtime together. Hence, it is only possible to define chains of tools, and each tool simulates its own model and passes the results as input to the next tool. The goal of these frameworks is to create a control tool, which provides biologists a uniform access to existing computational tools.

[15] http://sbml.org/; accessed 03.07.2007
[16] http://www.csml.org/; accessed 03.07.2007
[17] http://128.208.17.155//research/sbwIntro.htm; accessed 03.07.2007
[18] http://biospice.sourceforge.net/; accessed 03.07.2007

2.2. *Structural simulation tools*

Besides the modelling and simulation of biochemical processes, it is necessary for the simulation of the entire system to simulate the physical structure. Therefore, different techniques have been developed in the last 40 years.

The first approach to simulate branching structures appeared in 1966, based on cellular automata. This approach has been theoretical developed from John von Neumann (1903 - 1957). A cellular automaton is a discrete dynamic system in time, space and state. The smallest unit, called a cell, can have any one of a finite number of states. The state of a cell depends on its previous state and the state of its neighbours' at the previous time state. All cells update synchronously. An example is shown in Figure 4. The left illustration shows cellular automaton which incipient with the starting cell '*1*'. All cells switch to active when they have exactly one connection to an existing active cell. The right illustration shows the result of a modification of this basic rule. All cells, which have a connection to a cell that switch to active in the same time step, switch back to invisible.

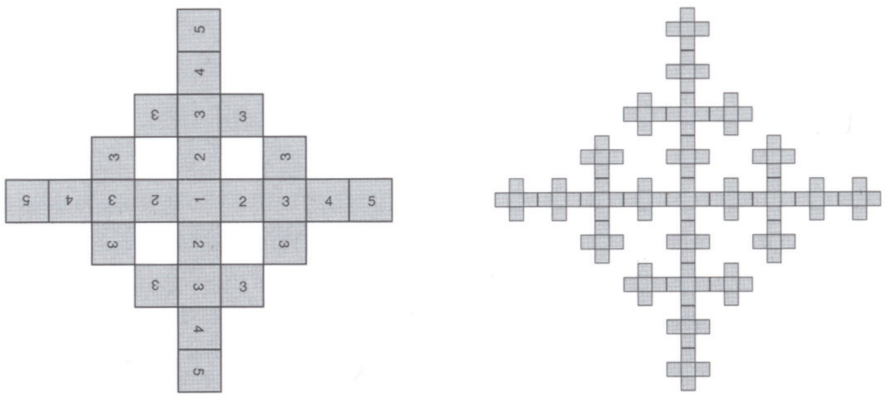


Figure 4 Cellular automata after Ulam [Ula66].

At present, a multiplicity of software tools for cellular automata exists. A list of tools and examples are present in the book, 'Simulation with Cellular Automata' [Wei98]. A similar approach to Cellular Automata is agent based simulations. The smallest unit is called agent here. This method is used for a wide range of economic simulation, social complex environment simulations as well as natural science simulations. The user defines the behaviour of each agent and rules for the interaction. The current state of an agent depends on its previous state and its neighbours' state. Similar to cellular automata, all states update synchronously.

An easy to use, open source multi agent simulation tool is MASON[19], developed from Sean Luke et al. at the George Mason University, Fairfax, Virginia. MASON is a shortcut for **M**ulti-**A**gent **S**imulator **o**f **N**eighbourhoods. The tool is completely implemented in Java and delineates between model and visualisation. Hence, it is possible to run a simulation without or with a visualisation. First of all, the user defines a model with the help of different predefined field types. In the next step, the user can assign a graphical representation to the field elements. All agents in the system are controlled by a global discrete event scheduler, and in

[19] http://cs.gmu.edu/~eclab/projects/mason/; accessed 03.07.2007

each time step the system can save the current system state (checkpoint). This basic functionality is illustrated in Figure 5 (from [Luk05]).

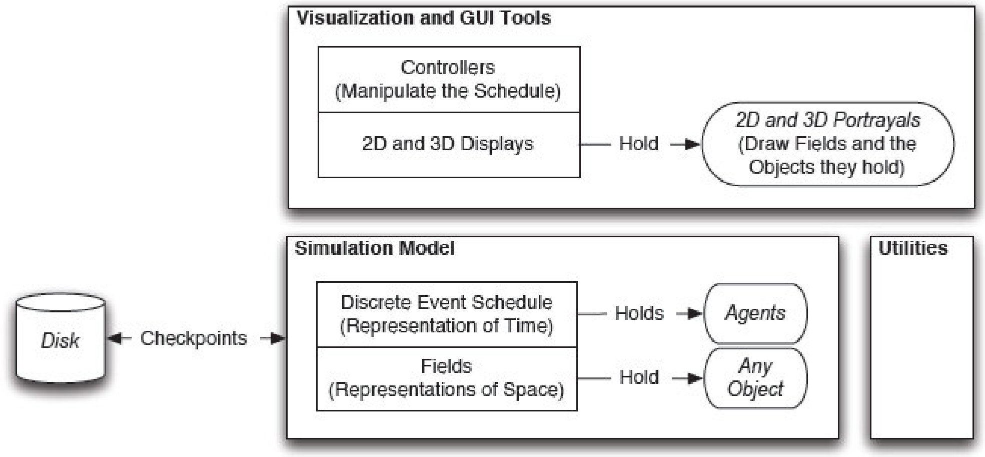

Visualization and GUI Tools
Controllers (Manipulate the Schedule)
2D and 3D Displays
Hold
2D and 3D Portrayals (Draw Fields and the Objects they hold)
Simulation Model
Utilities
Disk
Checkpoints
Discrete Event Schedule (Representation of Time)
Holds
Agents
Fields (Representations of Space)
Hold
Any Object

Figure 5: Basic elements of MASON (from [Luk05])

A good overview and example implementations of simulations using MASON can be found in the publication, ‘MASON: A Multiagent Simulation Environment’ [Luk05].

The breve simulation environment[20] is an alternative multi-agent simulation tool. The emphasis rests upon 3D simulation and segregation between calculation and visualisation is not feasible. The tool includes physical simulation utilities as well as collision detection and use OpenGL for the visualisation. More information can be found in the conference article, ‘breve: a 3D simulation environment for the simulation of decentralized systems and artificial life’ [Kle02].

First more or less realistic looking branching structures were developed from Dan Cohen with procedural techniques in FORTRAN in 1967 [Coh67]. He used three basic procedural rules for the generation of the structure.

In 1968 Aristid Lindenmayer (1925 – 1989) introduced a new technique based on a string rewriting mechanism (L-Systems). In comparison to the imperative programming paradigm, in the rule based programming paradigm no specific execution order for the rules are defined. The rules only specify the changes of the current system and are picked up from the control unit when they are applicable. Familiar examples for the rule based programming paradigm are grammars of natural language and formal grammar, developed by Noam Chomsky. In L-Systems, all applicable rules are executed in parallel. Deterministic L-Systems consist of an ordered triple $G = (V, \omega, P)$. V is an alphabet, $\omega \in V^+$ is a not empty set over the alphabet and $P \subseteq V \times V^+$ is a not empty set of production rules.

[20] http://www.spiderland.org/breve/; accessed 03.07.2007

A simple example:

$$V = (F, +, -, [,])$$
$$\omega = F$$
$$P = \{F -> F[-F]F[+F][F]\}$$

A virtual drawing device, called the ‘turtle’ [Abe82], interprets the string and produces a graphical representation of it. Each character in the string has a geometrical meaning. Start with the initial string (ω), the system uses all applicable rules in each step and produce (P) in each step a new string (s_i). For the visualisation, the string s_i is scanned from left to right and the geometrical structure T_i is constructed by interpreting the occurring character (see Figure 6).

$$\begin{tikzcd}
\boxed{\omega} \arrow[r, "P"] & \boxed{s_1} \arrow[r, "P"] \arrow[d, dotted] & \boxed{s_2} \arrow[r, "P"] \arrow[d, dotted] & \boxed{s_3} \arrow[r, "P"] \arrow[d, dotted] & \boxed{\dots} \arrow[d, dotted] \\
 & \boxed{T_3} & \boxed{T_2} & \boxed{T_3} & \boxed{\dots}
\end{tikzcd}$$

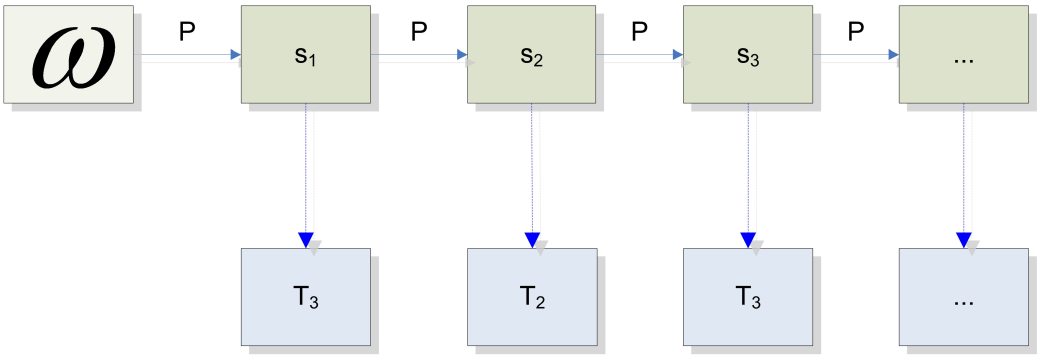


Figure 6: L-system and turtle interpretation.

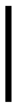
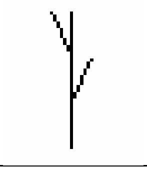
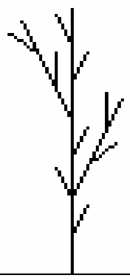
Table 1 gives an overview of those geometrical meanings which are necessary for the understanding of the example above.

Table 1: Turtle geometry: Geometrical interpretation of characters in L-systems.

Character	Geometrical interpretation
F	Moves forward and draw simultaneously a line of length one
+	Rotates 45 degree anti clockwise
-	Rotates 45 degree clockwise
[	Saves current position on the stack
]	Top element of the stack is the current state

Table 2 illustrates the relation between iteration step, string and geometrical form.

Table 2: Illustrate the relation between iteration step, string and resulting geometrical form for the example.

Iteration step	String	Geometrical form
0	F	
1	$F[-F]F[+F][F]$	
2	$F[-F]F[+F][F][-F[-F]F[+F][F]]$ $F[-F]F[+F][F][+F[-F]$ $F[+F][F]][F[-F]F[+F][F]]$	

Over the years, various extensions of the original concepts have been defined. For instance, stochastic L-Systems, declare probabilities to select the rules, allow generating more natural looking structures with more variability or parametric L-Systems, which allow specifying length and diameter of the atomic elements. In the book, 'The algorithmic beauty of plants', Lindenmayer and Prusinkiewicz describe all variation and extensions of L-Systems [Pru90]. The relations between Chomsky classes of language and language classes generated by L-Systems are illustrated in Figure 7.

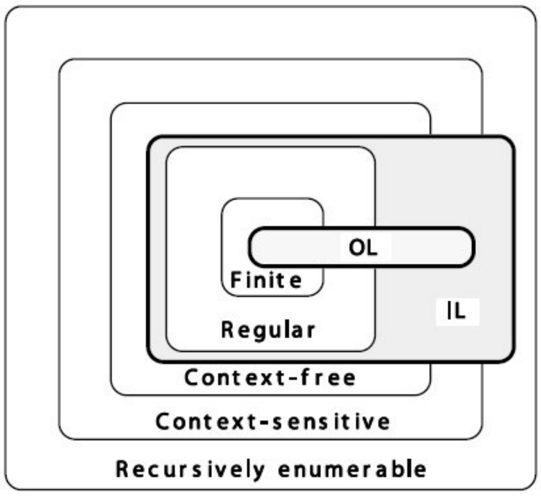


Figure 7: Relation between the Chomsky hierarchy and L-System classes. OL stands for the class generated by context-free L-Systems and IL for the class generated by context-sensitive L-systems.

Figure 8 and Figure 9 show, to illustrate the power of L-systems, two simulated flowers obtained form a context-sensitive L-System.

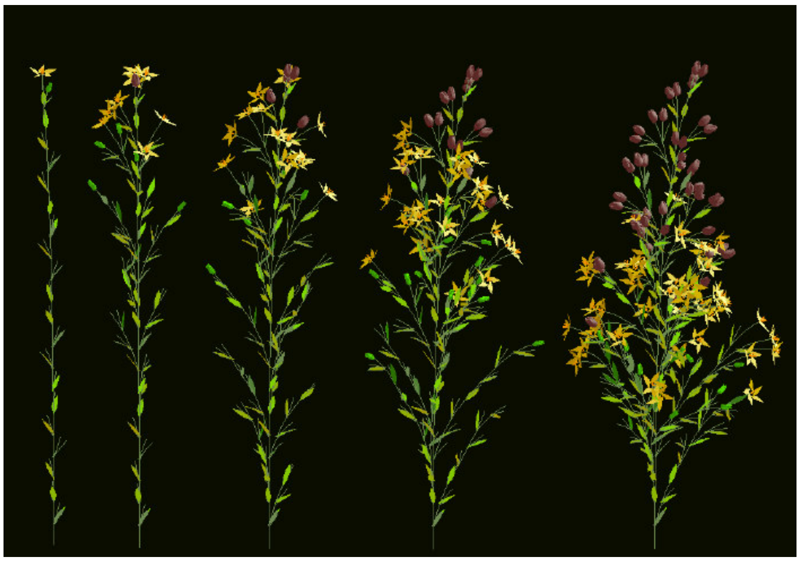


Figure 8: Mycelis muralis, based on a context-sensitive L-System from [Pru90].

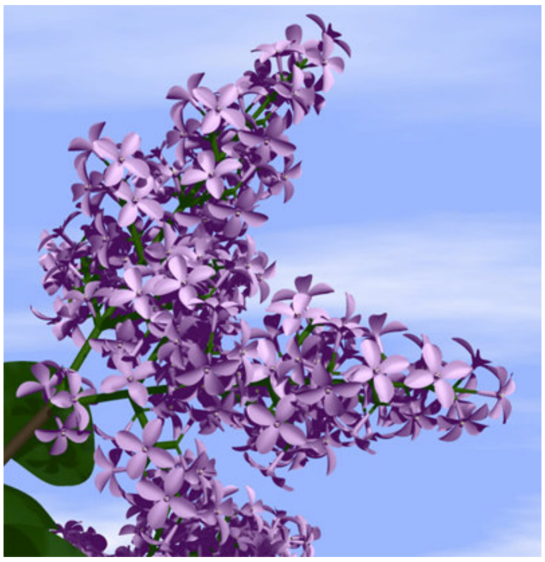


Figure 9: Lilac inflorescences from [Pru90].

L-Systems are based on well known theoretical concepts and create realistic looking models of plants. Nevertheless, even with all extensions, L-Systems have some limitations. In the first place, an element in the system can only be a direct successor of another or it can be a branch element. These relations are a too simplified view of reality. In the second place, classical L-Systems are only appropriate for the creation of one dimensional model with turtle interpretation. Extensions, like ‘map L-Systems’ and ‘cellwork L-Systems’ (see [Pru90]) allow modelling realistic looking two and three dimensional models, but the usage is fairly complicated. Finally, the definition of L-Systems varies from common object-oriented programming styles. The formalism is simple and supports no hierarchy of objects and other OOP features.

With reference to this background, Winfried Kurth designed a new formalism, ‘relational growth grammars’ (RGG), to avoid these problems (see [Kur07]). Together with Ole Kniemeyer, he defines a corresponding programming language, 'XL' (eXtended L-systems language)[21], as an extension of the Java[22] programming language. RGG is based on a well developed theoretical concept about graph grammars [Roz97]. Like L-Systems, RGG is a rewriting system operating on graphs instead of strings. The graph consists of nodes and edges and allows loops. Miscellaneous types of edges (relations) between the nodes are allowed, and therefore, the formalism is called 'relational'. The RGG rules are defined in XL and the graph is rewritten by the XL programme. A node in the graph can be a geometrical object (structure and additional parameter) or a transformation object (rotate, scaling, etc.) of a XL class.

The general syntactic structure of a RGG rule is shown in Figure 10. The whole RGG system consists of several RGG rules, which are usually applied in parallel to the corresponding graph.

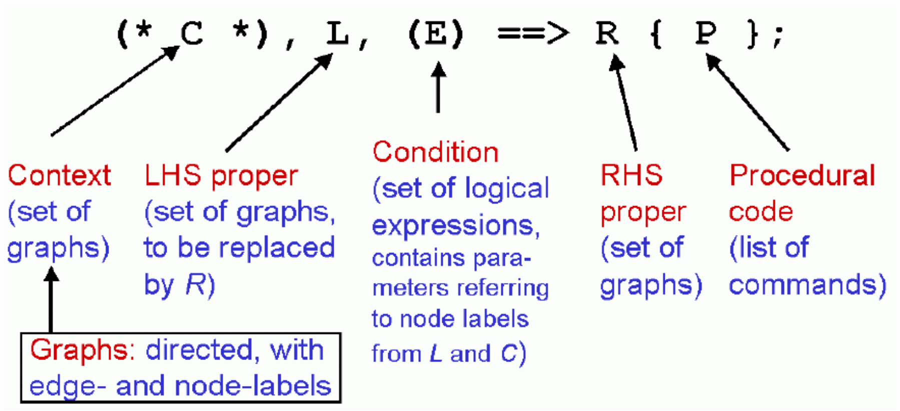

(* C *), L, (E) ==> R { P };
Context (set of graphs)
LHS proper (set of graphs, to be replaced by R)
Condition (set of logical expressions, contains parameters referring to node labels from L and C)
RHS proper (set of graphs)
Procedural code (list of commands)
Graphs: directed, with edge- and node-labels

Figure 10: General syntactic structure of an RGG rule. Simplified, the rule replaces *L* by *R and executes P* (from [Kur07]).

Figure 11 demonstrates an application of a simple RGG rule. The upper part of the figure shows one RGG rule, where the left hand side has to be replaced with the right hand side. All nodes *A* and *B*, which are connected via a direct edge from *A* to *B*, have to be replaced by the nodes *A*, *B* and *C*. Two different types of edges are used in the example (dotted and unbroken arrows). The lower part of the figure illustrates the application of the rule. The rule is applicable to the red marked and with an unbroken blue line framed area on the left side in the graph. Attend that the rule is not applicable to the part of the graph surrounded by the dotted blue line. The right lower part of the figure shows the rewritten graph after rule execution. New graph elements are highlighted in red and surrounded by a blue circle.

[21] http://www.grogra.de/; accessed 03.07.2007
[22] http://www.sun.com/: accessed 03.0702007

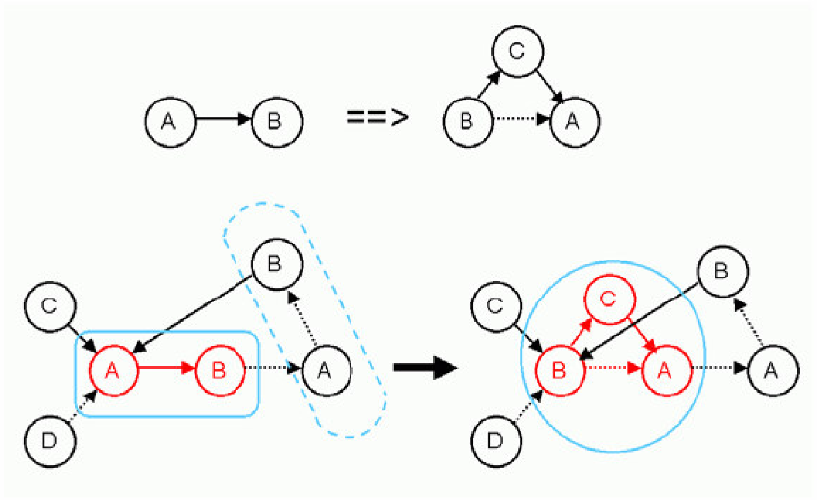


Figure 11: Relational growth grammar rule (upper part) and corresponding graph (lower part) (from [Kur07]).

A simple example of graph rewriting shows the capacity of RGGs against L-Systems. For example, the so called 'crossing over' process of two aligned DNA strings in sexual reproduction can not be expressed in an L-System rule. Figure 12 illustrates this recombination process in a RGG rule. The unbroken arrows represent the successor relation in base sequence of DNA and the dotted lines denote the alignments between two homologous DNA strings [Kur07].

A possible XL specification of the rule is:

a b, c d, (* a -align- c *) ==> a d, c b;

Whereas *a*, *b*, *c* and *d* are objects of the corresponding user defined XL classes. The standard successor edge is represented in the example with a blank, and the alignment edge is specified with *–align-*.

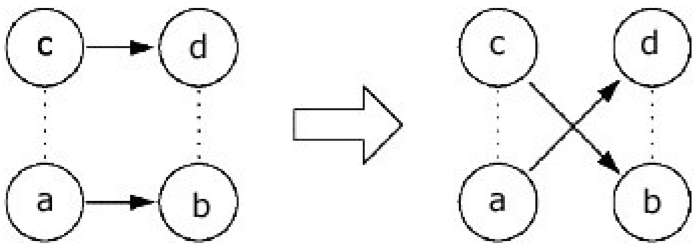


Figure 12: RGG rule for genetic crossing-over (from [Kur07]).

To use relational growth grammar in practice, Kniemeyer developed an integrated development environment named GroImp (Growth-grammar related Interactive Modelling Platform). The IDE contains an XL compiler, an extended editor for XL, a 2D graph visualiser and a 3D modelling and rendering unit. GroImp is a platform independent tool under the GNU public licence. The system allows embedding RGG rules in XL programmes, compile, execute and visualise the results of the simulation with help of OpenGL. For more details, a collection of RGG examples and a XL tutorial, see *http://www.grogra.de*. To clarify the dependencies between XL, RGG and the visualisation, the first example from the XL tutorial is abbreviated present below.

In 1904 the Swedish mathematician Helge von Koch introduced a simple rule-based approach to model a snowflake. The idea is, to divide an initial line into three parts and replace the middle part with two lines as shown in Figure 13.

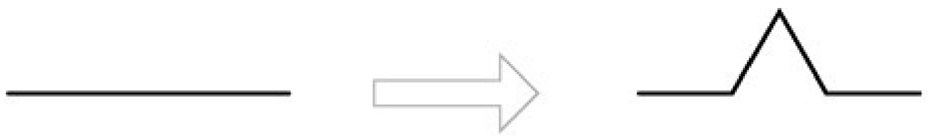


Figure 13: Koch construction step. Replace the middle part with two lines.

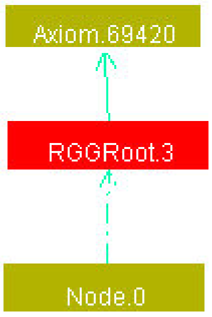
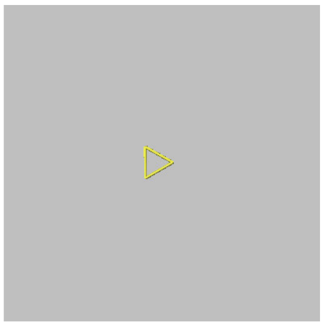
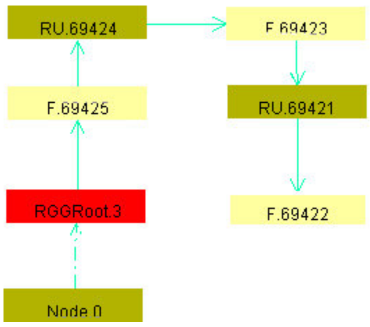
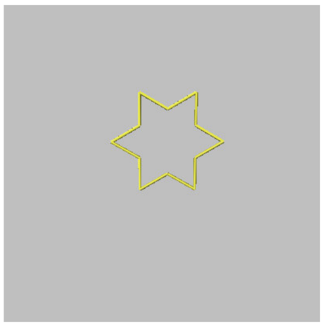
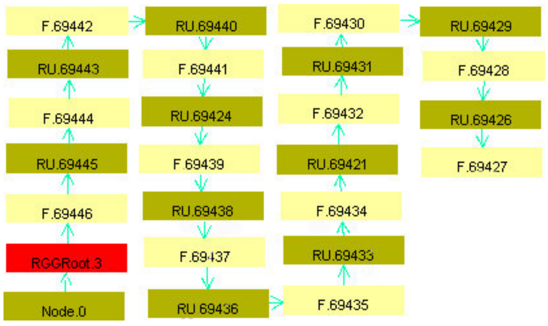
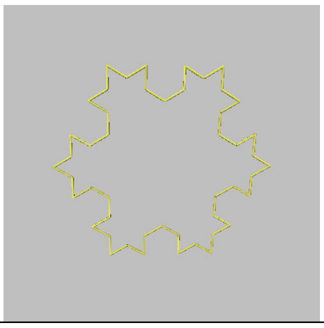
To generate a snowflake with this approach the initial figure has to be a triangle. The transcription of Koch construction step into two RGG rules embedded in a XL programme is shown in the following code:

```
public void derivation() [
Axiom ==> F RU(120) F RU(120) F;
   F ==> F RU(-60) F RU(120) F RU(-60) F;
]
```

The used symbols at this point are *Axiom*, *F* and *RU*, which are at the same time the nodes of the corresponding graph. *Axiom* is the start symbol and the first rule describes the initial triangle. Whereas, *F* draws a straight line and RU turn the orientation by the given angle. The second rule implements the Koch construction replacement. Accordingly, all nodes from type *F*, which are created with the first rule, have to be replaced with the right hand side of the second rule.

Table 3 shows the graph and the geometric outcome after n steps.

Table 3: Geometric outcome after n durations for the snowflake example

Step	Geometric outcome	Graph
0		
1		 RU.69424 F.69423 F.69425 RU.69421 RGGRoot.3 F.69422 Node.0
2		 F.69442 RU.69440 F.69430 RU.69429 RU.69443 F.69441 RU.69431 F.69428 F.69444 RU.69424 F.69432 RU.69426 RU.69445 F.69439 RU.69421 F.69427 F.69446 RU.69438 F.69434 RGGRoot.3 F.69437 RU.69433 Node.0 RU.69436 F.69435
3		Too complex to show

The pictures in Figure 13 and in Table 3 are out of the XL tutorial from GroIMP.

2.3. *Simulation of biological systems*

The simulation of an entire biological system requires a combination of functional and structural simulation. Currently, a wide variation of models and simulations for more or less complex biological systems exist. Examples can be found in medical and botanic area.

The first group of simulations has a medical orientation. Scientists try to simulate organs, organ systems or a musculoskeletal system of human beings and animals. The mayor focus in medical simulations is the virtual heart. Various models for fluid dynamic, contraction, electrocardiogram or drug effects have been developed in the last decade. Scientists used the medical knowledge to create an entire heart model based on biochemical processes and 3D visualisation. All existing tools are hard coded and optimised for only one or two use cases. This kind of specialisation is characteristic for the simulation tools. For instance, some tools use the MatLab fluid dynamic tool box for the simulation of the blood fluid dynamic. In addition, the simulations use self-written tools to combine the structural and functional parts of the model in one new simulation environment. An overview of heart examples is shown in, 'Computational modelling of biological systems: tools and visions', by Peter Kohl et al. [Koh00].

For other human organs similar simulation approaches exist. One example is the 'German HepatoSys[23] competence network of system biology', which tries to model a human liver.

Cornelia Kober et al. developed an anisotropic simulation of the human mandible. The simulation is based on computer topographic base data, organ geometry and load distribution experiments (see [Kon07]).

The mayor issue of the second group is to model and simulate flowers, crops, trees and landscapes. Similar to the first group, most of the software systems for plant modelling are problem specific. One tool or framework is only applicable for a particular plant or a group of similar plants.

As an exception, GroImp allows to model 3D structures and to integrate the biochemical process simulation into the model. The system uses the XL language, an extension of Java programming language, to model biochemical processes like in other high-level programming languages. The disadvantage is that all biochemical processes have to be implemented in XL, and therefore, no existing biochemical simulation tools or models can be reused. For instance, the barley breeder simulation in GroIMP shows varied mutation of barely, which is controlled by various biochemical processes. The model can be found in the GroIMP release (see *http://www.grogra.de).*

The modelling language LIGNUM represents a further example for a similar approach. Functional and structural simulations are combined in the tool with the same disadvantages like in GroIMP. Information about LIGNUM can be found in the thesis from Jeri Perttunen, 'The Functional-Structural Tree Model LIGNUM' [Per07].

Moreover, a huge quantity of non-generic simulation tools exists. Ming et al. developed a framework for the simulation of biological invasions in a heterogeneous landscape. The scientists integrate geographical information, biophysical structures and diffusion dynamic in the workflow [Min04].

[23] http://www.systembiologie.de/en/index.html; accessed 03.07.2007

The correlation between atmospheric CO_2 and crown development is simulated in the model of Chen et al. The simulation uses a 3D model of a tree and a 3D model of the environment to simulate the relationship between photosynthesis, light interaction and growing. More information can be found in [Che97].

An overview of structural and functional plant models are presented at the 5th international workshop, 'Functional Structural Plant Models', in Napier, New Zealand. For more information visit the workshop website[24].

Hitherto, all presented generic or non-generic tools produce good simulation results for specific areas. For each model the functional and structural part of the model has to be completely re-implemented in the tools own formalism.

The thesis introduces a new software framework approach, which combines existing structural and functional simulation tools. Consequently, the system allows the re-usage of already existing and defined models, without reimplementation.

[24] http://algorithmicbotany.org/FSPM07/index.html; accessed 03.07.2007

3. Requirements

The basic idea for the new software system is to combine, in a generic way, biochemical and structural simulation tools to simulate complex biological systems. In principle two possible solutions are conceivable and will be discussed in the following:

1. Create an entire software system and include all discussed methods, like differential equation, stochastic process, Petri Net, L-System, RGG and agent-based simulation (see *Fundamentals and Related Work*). The vantages of such a system are: a uniform user interface, optimised performance and uniform data storage. Disadvantages are: the complexity of the system, reimplementation of all methods, heavy maintainability, and the impossible reuse of existing models as well as the inflexibility for future modelling approaches.

2. Create a software mediator which is placed between existing simulation tools. Vantages of the mediator solution are: re-use of existing tools, no limitation for adding system, relative small software systems, easy to administrate and to maintain as well as global control of the simulation. The main disadvantages are: no optimised performance, concerning the communication between the tools, no uniform user interface for the model definition, various storage files.

Table 4: Vantages and disadvantages of the embedded and the mediator approach. Plus sign connote vantage and minus sign connote disadvantage.

	User Interface	Performance	Data storage	Complexity	Maintainability	Reuse existing tools	Adding new methods
Embedded approach	+	+	+	-	-	-	-
Mediator approach	-	-	-	+	+	+	+

After having compared the advantages and disadvantages of both systems, respectively, the decision was made to use the mediator approach for the new software system. The handicap of inconsistent user interfaces will be compensated by the reusability of existing models and user familiar tools. The performance can be optimised with a good system and interface structure design. The general idea of the mediator approach is presented in Figure 14. The software system is placed in the middle and communicates with the adapted pathway and structural simulation tools, respectively. A user creates the biochemical and structural models and initialises the mediator software.

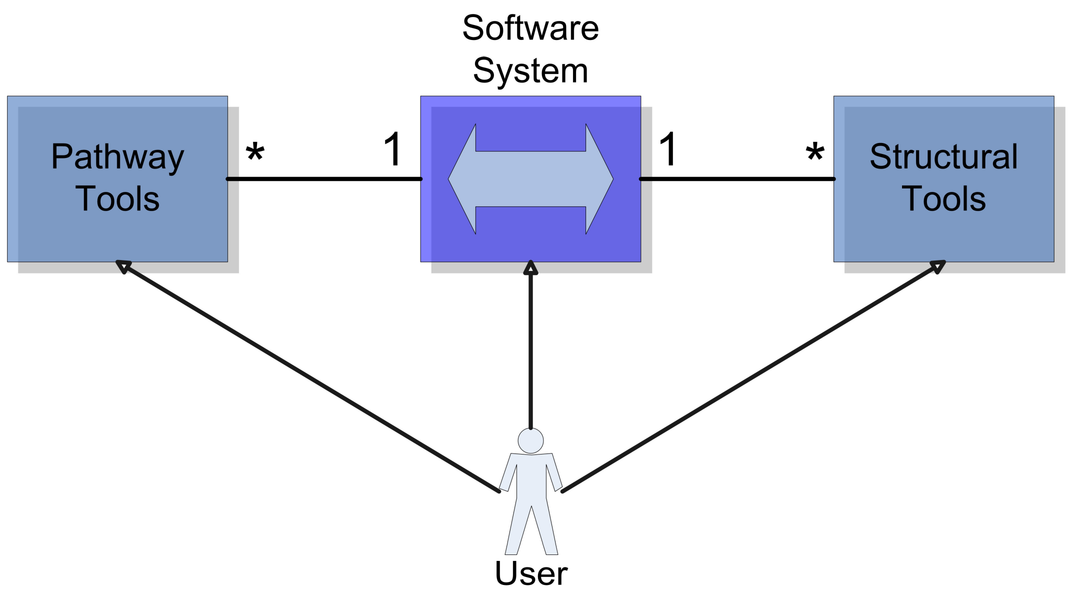


Figure 14: Basic principle of the mediator concept. The connections between the software system and the simulation tools are bidirectional.

First of all, the new system must be able to add n $(n \in N)$ independent modelling and simulating tools. Therefore, a clear and well defined interface structure is necessary. Adding a new tool means in this context that a connection between tool and mediator has to be established (see Figure 15, number one) and information can be transferred in a bidirectional way (see Figure 15, number two). The information, which are called values in the figure, could be current simulation parameters, simulation element values, new values for the simulation or initial configuration parameter values.

In the second place, the software system (BioSimMediator) should have a basic control over all added modelling and simulating tools, in the following called subsystems. Basic control means here that the BioSimMediator is able to start, stop and reset all simulations in the connected subsystems. Furthermore, control over simulation specific commands, like *run* a simulation (see Figure 15, number three).

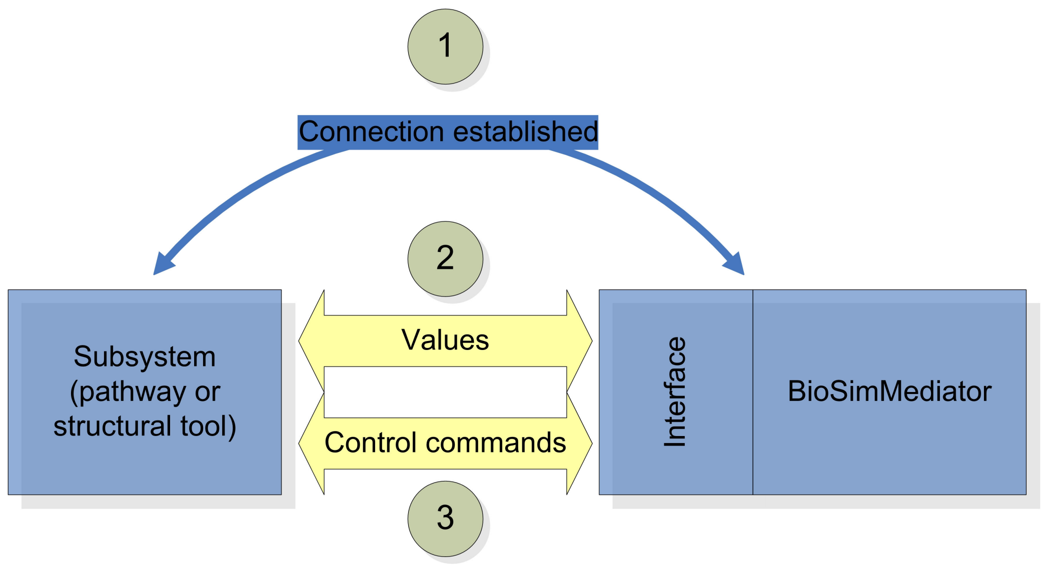
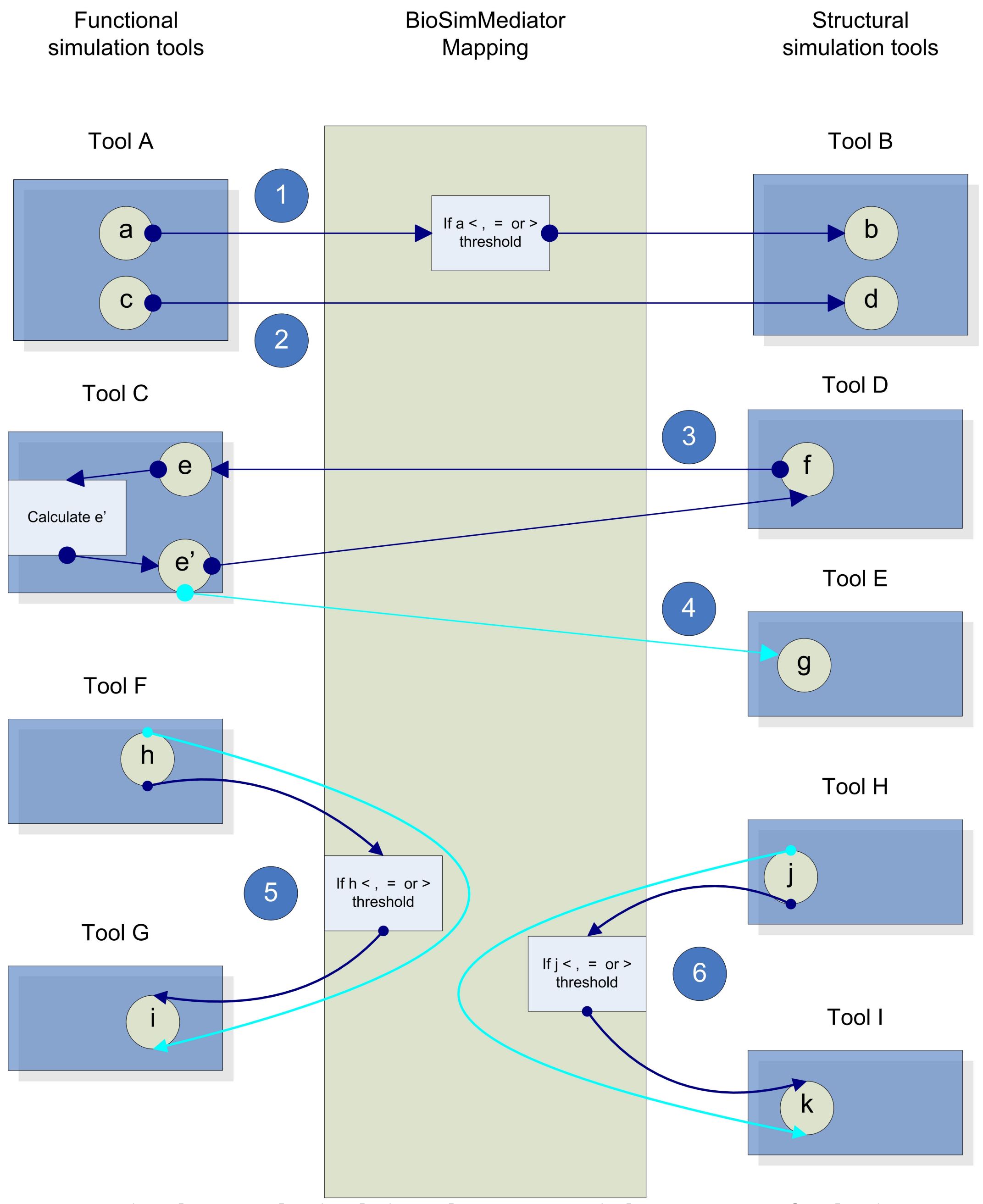


Figure 15: Connection and communication between BioSimMediator and the subsystems.

For an entire simulation, it is necessary to enable the user to define mappings between the subsystems. That means, the user specifies which subsystems are allowed to communicate with each other and under which conditions. For instance, a value from subsystem *A* has to be changed in the next step. Subsystem *B* calculates the value and the mediator knows, with the help of the user defined mapping, that the value has to be sent to subsystem *A*. All possible variations are shown in Figure 16. In principle, the mediator has to distinguish between a direct mapping and a threshold restricted mapping. The figure illustrates six mappings between the subsystems. All mappings are only present in one direction; in reality the reverse mappings are also possible.



Functional simulation tools
BioSimMediator Mapping
Structural simulation tools
Tool A
Tool B
1
a
If a < , = or > threshold
b
c
d
2
Tool C
Tool D
3
e
f
Calculate e'
e'
Tool E
4
g
Tool F
h
Tool H
If h < , = or > threshold
5
j
Tool G
If j < , = or > threshold
6
i
Tool I
k

Figure 16: Mappings between the simulation subsystems. Capitals are synonym for the simulation tools and the lower case letters stand for attribute values inside these systems.

A brief description of the six represented mappings follows (see Figure 16):

1. Threshold restricted mapping: The attribute value of *a* from tool *A* will sent to tool *B*, if the condition is true.
2. Direct mapping: Attribute value *c* will be sent in each mapping step to tool *B* without restriction.
3. Direct mapping: The attribute value of *f* from tool *D* will be sent to tool *C*. This tool recalculates the attribute value (*e* and *e'*) and sends it back to the attribute *f* in tool *D*.
4. Direct mapping: Equal to point three, but the recalculated value will be sent to another subsystem *E*.
5. Direct or restricted mapping: The dark blue arc shows the restricted mapping and the azure picture the direct one. The peculiarity is that both tools *F* and *G* are functional simulation tools.
6. Direct or restricted mapping: in comparison to point six, tool H and I are structural simulation tools.

Accordingly, the mapping unit of the mediator must be generic enough to handle all present mapping types.

The next requirement is a step size control for each subsystem. Step size control means, most of the subsystems work with special simulation methods and all these methods uses its own internal representation for a step in the simulation. Some subsystems use a time step (minutes, second, hours) and others use an abstract step size independent from the time. According to this, the BioSimMediator must be able to save for each subsystem an own step size relative to its internal step size representation. This requires an integrated step size representation in the mediator.

After the end of the simulation, the user requires an overview of the previous system states in each simulation step. Hence, a log unit is required. Therefore, the system must save all attribute names and corresponding values in each simulation step. On top of that, the user should have the opportunity to pre-elect the required attributes. In this case, the system saves only the user defined name value couples. To associate the name value couples to the right step and subsystem, the log file has to contain the respective simulation step and the subsystem's name.

In addition, the system must include a basic statistic calculation unit, which assist the scientist. For instance, the calculation of an arithmetic average of a special concentration or the biomass of the entire biological system is conceivable.

To sum up, the core functional requirements are illustrate in the use case diagram in Figure 17. The main scenario is the 'run' of a simulation, which is extended by the control of the subsystems. To control the subsystems, the mediator has to be configured by the user. With reference to the presented requirements, the system must provide a mapping, a log unit and a statistic unit, which are also used for the control process. Apart from that the figure illustrates, the user's responsibility for the configuration of the mediator. On the other hand, the BioSimMediator can interact with any desired number of subsystems.

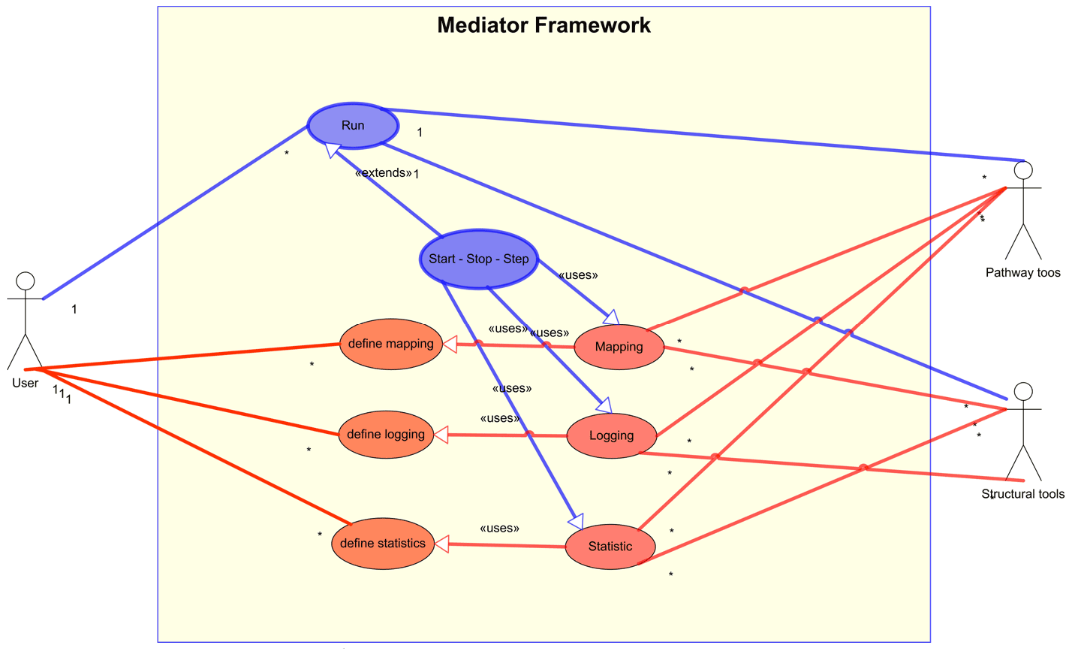


Figure 17: Use case diagram for the BioSimMediator.

Beside the functional requirements, the system must satisfy the following non functional requirements.

To guarantee easy maintenances and further extensions, the implementation should use design patterns and a clear interface structure. Also, the usage of well known *JavaDoc* comments improves the quality of the system.

The reliability of the system must be high; otherwise the complete simulation process will fail. Reliability comprises availability, probability of unavailability and mean time to failure. To guarantee reliability, an easy and well documented program structure is recommended. In the same way, the robustness of the system must be ensured. In this case, a short restart time after a failure and a low percentage of events, which cause failures, are preferable.

The using of the high level programming language Java for the implementation guarantees a platform independent system. The portability of the system and the compatibility to other programming languages must be considered. Most simulation tools are implemented in C or C++, and therefore, an interaction must be possible.

The environment of the system will be in the first place a single processor system. This necessitates no special requirements for the system size or memory usage.

In addition, the software speed is important for the simulation process. The mapping between the subsystems must be fast enough to ensure a successful operation. A fast mapping is procurable with the help of a mapping structure without overhead. For further extension, like running the simulation via a network or on a computer cluster, a suitable data transport format shall be defined.

The last requirement is the usability of the system. Creating a model in the common modelling tools requires good computer skills. Based on this fact, all users of the system should be able to use a high level programming language. Therefore, experienced users should be able to use the system after reading the documentation.

4. Design

This chapter gives an overview of the system design and the primary system parts with an emphasis on a conceptual and abstract level. Implementation details of the core units are presented in the next chapter.

The basic idea of the system is to combine existing modelling and simulation tools and to mediate between these tools (see Figure 14). Also, like mentioned before (see chapter *Fundamentals and Related Work*), a global control of all connected subsystems and a storage of the simulation results is necessary.

The system is based on an object-oriented design. Accordingly, the fundamental components in the system represent objects with their own state as well as operations rather than functions. A precisely defined interface allows the objects to provide their operations for the system. The *Unified Modelling Language* (UML) provides a range of notations to describe the interaction and the dynamic of the objects as well as the system design and behaviour. Experience has shown that objects are often too fine-grained for the entire description of the system design. Consequently, objects are combined to larger-grained abstractions called subsystem or framework (in Java called package). To avoid a mix-up with the names of the simulation subsystems in future, the different subsystems are called simulation subsystem respectively mediator subsystem. Attend that the synonym subsystem is used in two ways. In the first case the description is used for an entire software system (simulation subsystem) and in the second case for a functional unit inside a software system (mediator subsystem).

For a clear and intuitive system design the system functionality is distributed in packages. The mayor functionalities of the system are: connecting to subsystems, handling of simulation values, mapping of the subsystems and the logging and statistic process. To obtain a high-quality software system, which is easy to extend and to maintain, an abstraction layer for each mediator subsystem is defined. In addition, well known design patterns ([Gam95]) for the relationship and interaction between objects and subsystems are used.

The mediator subsystems are described in detail followed by a brief description of the relationships.

4.1. Subsystem connection

The first mediator subsystem I will explain in more detail is the part of the system, which is responsible for the connection to the simulation tools. As mentioned before (Figure 15), the connection includes a direct communication and data transfer with the subsystems. For the simulation of biological systems it is necessary to combine functional and structural simulation tools. Accordingly, the mediator must be able to add and connect to as many subsystems as the user needs for the simulation. Therefore, it should be possible to connect $n\ (n \in N)$ subsystems to the mediator. Also, the interface for the connection must be generic enough to connect to different kinds of simulation subsystems.

Currently, four functional simulation subsystems and two structural simulation subsystems (as shown in Figure 18) are connected to the mediator. The functionality of all these tools, with the exception of the so called *Calculation tool*, is described in the chapter *Fundamentals and Related Work*. The *Calculation tool* is based on the open source Java library *Java math expression parser* (JEP)[25], which allows the handling of mathematical expressions. In the mediator the tool is used as a utility subsystem for basic mathematical calculations.

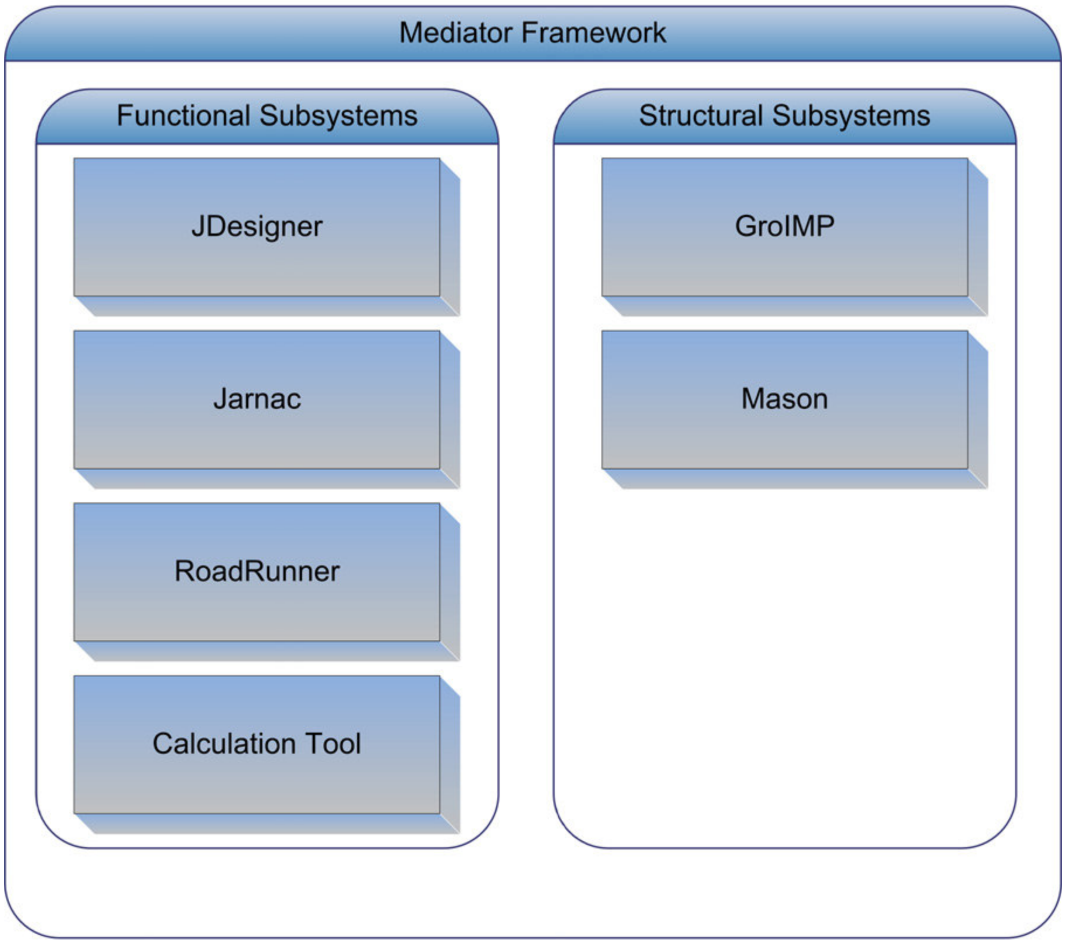


Figure 18: Currently connected simulation tools.

The mayor problem in the interconnection between simulation subsystem and mediator is that nearly all tools provide different interface. Furthermore, the tools are implemented in various programming languages, for instance Java, C/C++, FORTRAN or Perl. Another point is the start up process of the tools. Some tools need special start up parameters, like simulation time-length or simulation step size in advance, whereas other tools need these parameters after the start up.

In reference to these problems, the interconnection between mediator and simulation tools is split in two parts. The first part is only responsible for the connection and identification of the tool during runtime and called subsystem handle. The communication with the simulation subsystem is controlled by the second part and called subsystem control.

[25] http://www.singularsys.com/jep/; accessed 12.07.2007

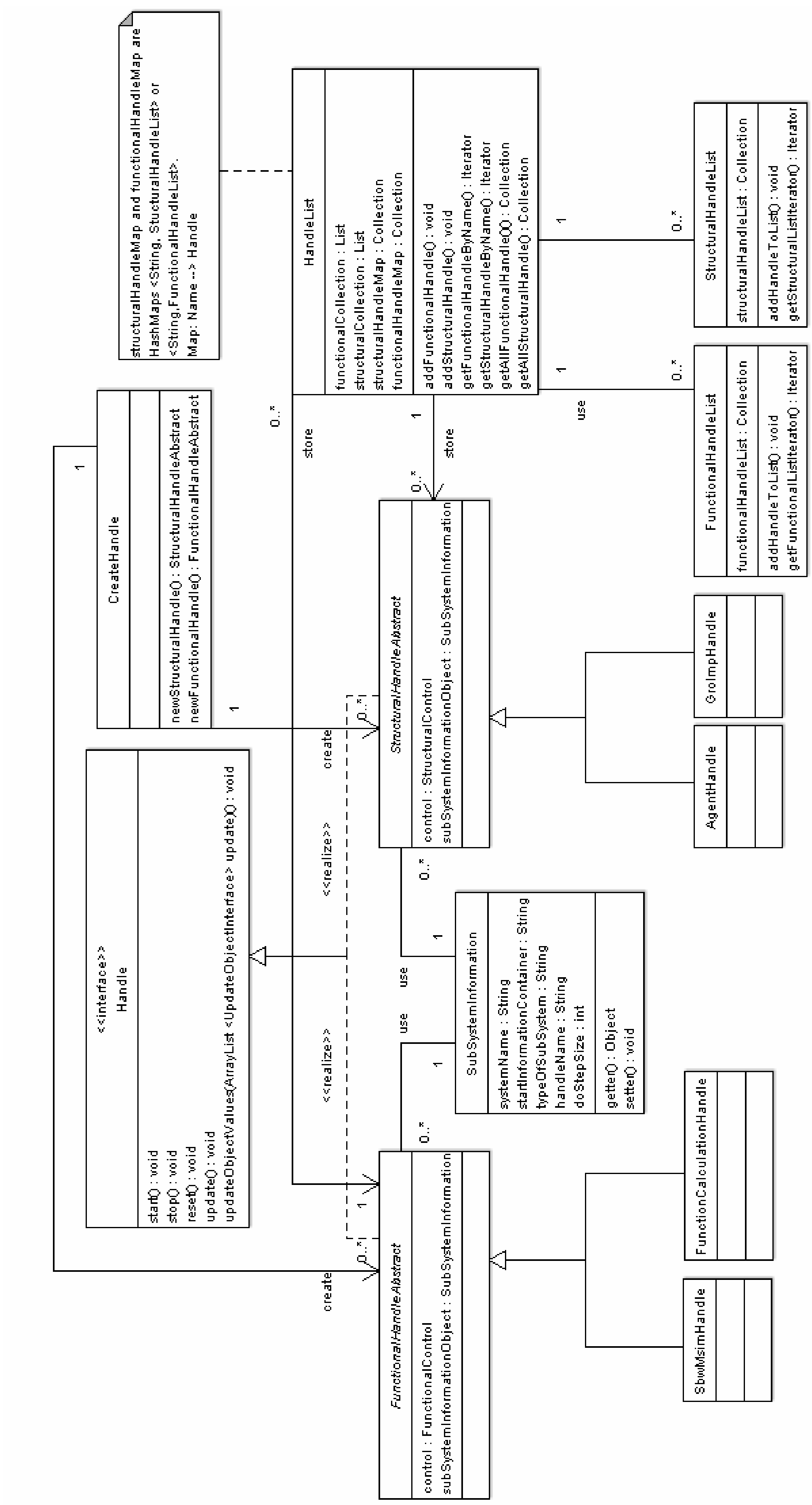
4.2. Subsystem handle

The design of the handle subsystem is illustrated in the UML class diagram in Figure 19 (all class diagrams in this thesis are incomplete. Only those elements which are needed for comprehension are represented). To add a new simulation tool to the mediator a new class must be created, which implements the interface *Handle*. The interface contains five method declarations: *start(), stop(), reset(), update*() and *updateObjectValues(…). Start()* or rather *stop()* starts or stops a simulation, respectively. *Reset()* sets the simulation back to its initial state. The *update()* method is responsible for updating the simulation values in the subsystems to read them (values) in the next step. In contrast to *update()* the method *updateObjectValues(…)* writes new values into the simulation subsystem.

By virtue of similarities between all structural and all functional simulation tools a differentiation between functional and structural handles is nevertheless reasonable. Therefore, two abstract classes *FuntionalHandleAbstract* and *StructuralHandleAbstract*, which both implement the introduced interface *Handle*. The actual simulation classes inherit the corresponding abstract class. Both abstract classes save a corresponding control object, which is responsible for the communication with the subsystem. In the class diagram, four examples for actual simulation handle classes are presented. As convention, all these class names end with the string 'Handle'. *SbwMsimHandle* is the handle class for a multi simulation using Jarnac. Jarnac is not directly connected with the mediator. The System Biology Workbench (SBW) is used as a connection manager. The workbench provides interfaces to various programming languages. In this way, the problem to connect tools which are implemented in different programming languages is solved. For more information see chapter *Fundamentals and Related Work*. The other functional handle class, *FunctionalCalculationHandle* is responsible for the mathematical utility subsystem. Besides, two structural handle classes exist. First of all, *AgentHandle* is the handle class for the multi agent simulation tool Mason. The last example handle class is *GroImpHandle* for GroIMP.

In addition, the abstract classes contain a *SubSystemInformation* object. These objects save all required information about the subsystem. The first attribute of this object saves a unique user defined name for the subsystem. The second attribute saves a string which contains a system path information. Nearly all simulation tools save the model definition in a file. For instance, Jarnac, roodRunner or JDesigner use the SBML format, which is based on XML. GroIMP uses a self defined storage format. Other tools need class path information for loading the models. This attribute is called *startInformationContainer.* It saves the required file path or class path for the start up in a string. The next attribute dedicates the simulation subsystem to the handle type. Only two values are permitted: functional or structural. Attribute four saves a unique handle name. The last attribute includes the step size for the subsystem relative to a mediator step. For instance, a value one means, in each mediator step one subsystem step is executed. A value four means, in four mediator steps one simulation step is preformed.

The creation of a handle class object is controlled by the class *CreateHandle*. Beside the name convention, all handle class has to be saved in the package *uk.ac.rothmasted.mediator.handle*. Dependent on the handle type (functional or structural) *newStructuralHandle()* or *newFunctionalHandle()* search in the package for classes whose names ending with 'Handle' and call the corresponding constructors. The respective return value of the methods is the handle object. For an easy access of these objects, a reference to each new created object is saved in a collection (map) in the class *HandleList*. These lists must be unique in the system. Therefore, the classes must be implemented as a singleton (singleton pattern, see [Gam95]) or must be static. It is possible that more than one instance of a subsystem, which is represented by the corresponding handle class, is created during runtime. Hence, the collection must save a mapping between handle name and a list of object references. These lists are collections of the class *StructureHandleList* or *FunctionalHandleList* dependent on the handle type (functional or structural). *HandleList* offer methods to add a name reference pair to the maps and to return a list iterator over the object reference list for a given handle name.



structuralHandleMap and functionalHandleMap are HashMaps <String, StucturalHandleList> or <String,FunctionalHandleList>. Map: Name --> Handle
HandleList
functionalCollection : List
structuralCollection : List
structuralHandleMap : Collection
functionalHandleMap : Collection
addFunctionalHandle() : void
addStructuralHandle() : void
getFunctionalHandleByName() : Iterator
getStructuralHandleByName() : Iterator
getAllFunctionalHandle()() : Collection
getAllStructuralHandle() : Collection
StructuralHandleList
structuralHandleList : Collection
addHandleToList() : void
getStructuralListIterator() : Iterator
FunctionalHandleList
functionalHandleList : Collection
addHandleToList() : void
getFunctionalListIterator() : Iterator
CreateHandle
newStructuralHandle() : StructuralHandleAbstract
newFunctionalHandle() : FunctionalHandleAbstract
<<interface>>
Handle
start() : void
stop() : void
reset() : void
update() : void
updateObjectValues(ArrayList <UpdateObjectInterface> update)() : void
StructuralHandleAbstract
control : StructuralControl
subSystemInformationObject : SubSystemInformation
FunctionalHandleAbstract
control : FunctionalControl
subSystemInformationObject : SubSystemInformation
SubSystemInformation
systemName : String
startInformationContainer : String
typeOfSubSystem : String
handleName : String
doStepSize : int
getter() : Object
setter() : void
GroImpHandle
AgentHandle
FunctionCalculationHandle
SbwMsimHandle
store
store
create
create
use
use
use
<<realize>>
<<realize>>
1
0..*

Figure 19: Class diagram for the handle subsystem.

4.3. Subsystem control

In contrast to the handle subsystem, which is responsible for the management and start up of the simulation subsystem, the control subsystem comprises the communication and interaction with the subsystems. The subsystem handle class describes only on an abstract level the subsystem functions (start, stop, reset). In contrast to the subsystem control classes, which implement the interaction with the subsystem in a tool specific way. The interaction includes updating the simulation values in the modelling tools as well as in the mediator internal data structures. Also, all necessary configurations after the start up of the simulation tool will perform from the control class. Most of the handle class methods use the methods of the corresponding control class.

In principle, the system consists of two abstraction levels. The mediator interacts with the handle class and uses the provided methods. These methods forward the query to the control class method which includes the tool specific implementation. Therefore, the control class interacts with the real simulation subsystem.

When a new simulation subsystem shall be connected with the mediator, a new control class has to be implemented. For a better overview, the recommendation is to create for each new simulation subsystem a new package which includes all subsystem relevant files.

To obtain a well defined control class design, all control classes must inherit form *AbstractController* and re-implement all five methods. The class diagram for the control subsystem design is shown in Figure 20.

First of all the method *start()* is responsible for the simulation start. Next, *doStep()* performs one simulation step in the simulation subsystem. The third method *reset()* sets the simulation back to its initial state. It must be possible for the mapping between various simulation subsystems to readout values from a subsystem. Therefore, the method *readValues()* reads the values form the simulation tool and updates the internal data structure for the mapping. In contrast to *readValues()* the method *writeValues()* update the simulation values.

Generally speaking, there are three categories of simulation subsystems in respect of the connection with the simulation mediator. Hence, these differences in the interaction with the subsystems require three categories of control classes.

1. The modelling tool provides an interface which allows the mediator to interact directly with the modelling tool. Thus, it is possible to send values and control commands to the tool. Also the mediator is able to send a request and the tool replies the required information.

2. The modelling tool provides no interface which can be used by the mediator. Therefore, a wrapper class for each tool is created and all models must inherit this class. Via the wrapper class an interaction with the simulation is possible. In principle, a third abstraction layer is added between mediator and subsystem besides the handle and control layer.

3. Some simulation tools do not allow to using the wrapper class approach. For example, it is not possible for all required tool elements to inherit from a wrapper class or the tool code is not open source or no changes of the source

code are allowed. The only alternative is to hardcode a communication between mediator control class and simulation tool.

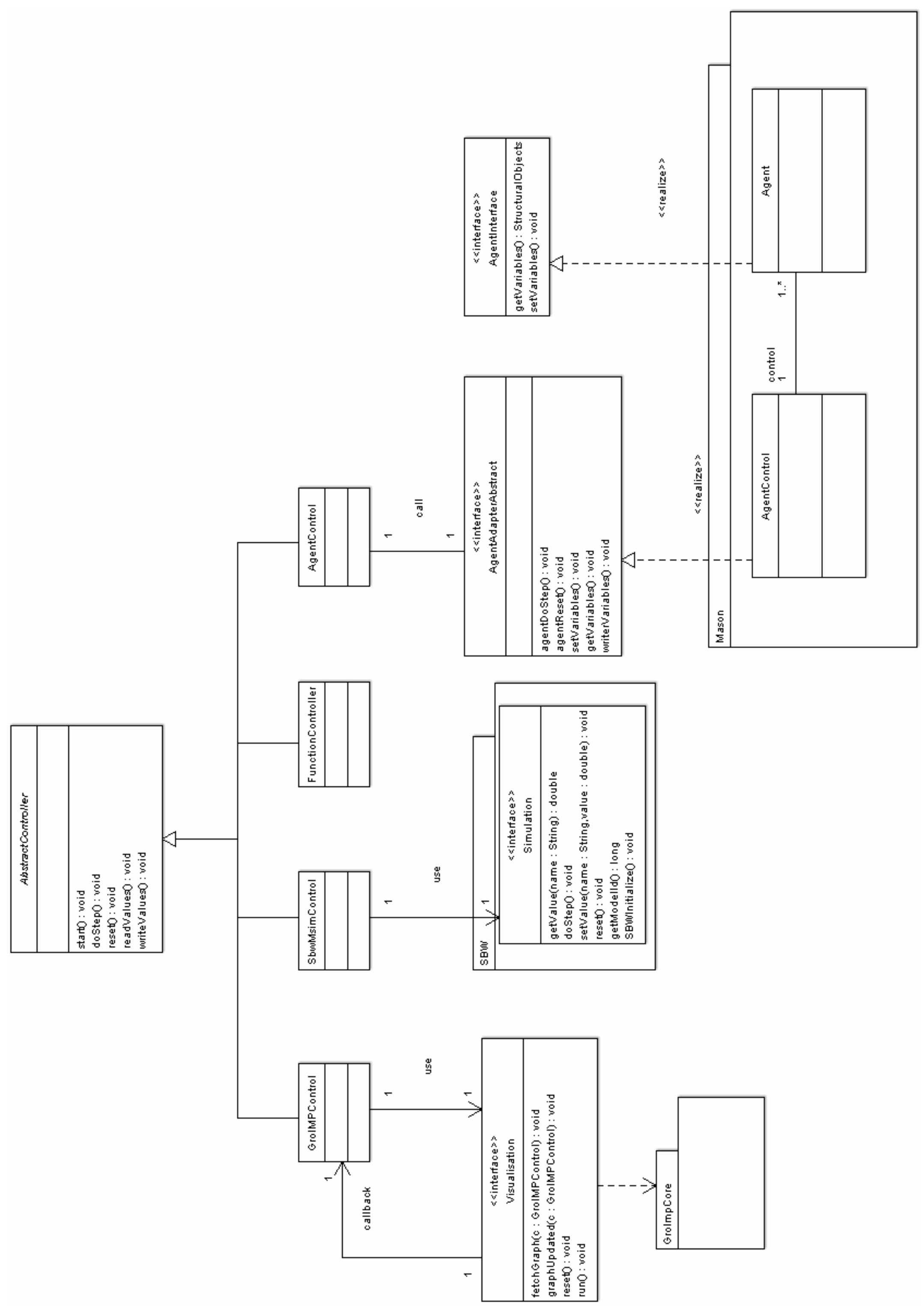


Figure 20: Class diagram: control subsystem.

The class diagram (see Figure 20) shows all three ways to interact with a subsystem. In the bottom left corner (landscape format) the diagram shows the interaction with GroIMP. By virtue of the internal structure of GroIMP it is only possible to use approach three, to hardcode the interconnection between mediator and tool. To guarantee the required quasi parallel execution of the RGG rules, the system uses various threads. A communication is only in a special system state, which is called in GroIMP *lock protected runnable*, feasible. The implementation and a detailed description of the GroIMP interface follows in the next chapter. In principle, the class *GroIMPControl* uses a class which implements the interface *Visualisation*. Via a call-back method call the class sends the required information back to the mediator.

In the middle of the class diagram the interaction with Jarnac is presented. As mentioned before, Jarnac is connected via SBW with the BioSimMediator. The interface *Simulation* shows all provided methods which can directly be called from *SbwMsimControl*. For instance, the method *getValue(name:String):double* returns a double value for the attribute with the identifier *name*. In addition, the interface contains the methods *doStep():void*, *setValue(name:String, value:double):void*, *reset():void* and *SBWinitialize():void.* The characters *Msim* in SbwMsimControl stands for mult simulation. This means that more than one simulation of the same type can be executed in parallel. Therefore, the interface includes a method to call the simulation identifier (*getModelId():long*). This interaction is an example for using a provided interface.

In the bottom right corner the connection with the multi agent simulation tool Mason is shown. Mason consists of a control class for all agents, called *AgentControl*. The functionality of the agents is defined in separate classes (in the class diagram ‘Agent’). For the interaction with the BioSimMediator each agent class implements the *AgentInterface. AgentControl* has to implement *AgentAdapterAbstract*. Now, the mediator can call these implemented methods and in that way an interaction between the agent simulation and the mediator is arranged.

4.4. *Transport of values*

For the interaction of different simulation tools a transport of values between each other is essential. The transport is restricted to attribute values. The challenge is to transport all required values from one subsystem through the mediator, possibly change the values of the attribute in the mediator in the meantime and override the values in another subsystem with the new values. All primitive data types are used in the various simulation tools. Therefore, the mediator must be able to store all primitive data type values. For a clear and intuitive design, the transport is split in two parts.

The first part is responsible for the transport from the subsystem to the mapping unit. Regarding to the differences between functional and structural subsystems this part is split in two sub areas. On the other hand, the second part performs the transport of values from the mapping unit to the subsystems.

An illustration of the transport between mediator and simulation subsystems is shown in Figure 21.

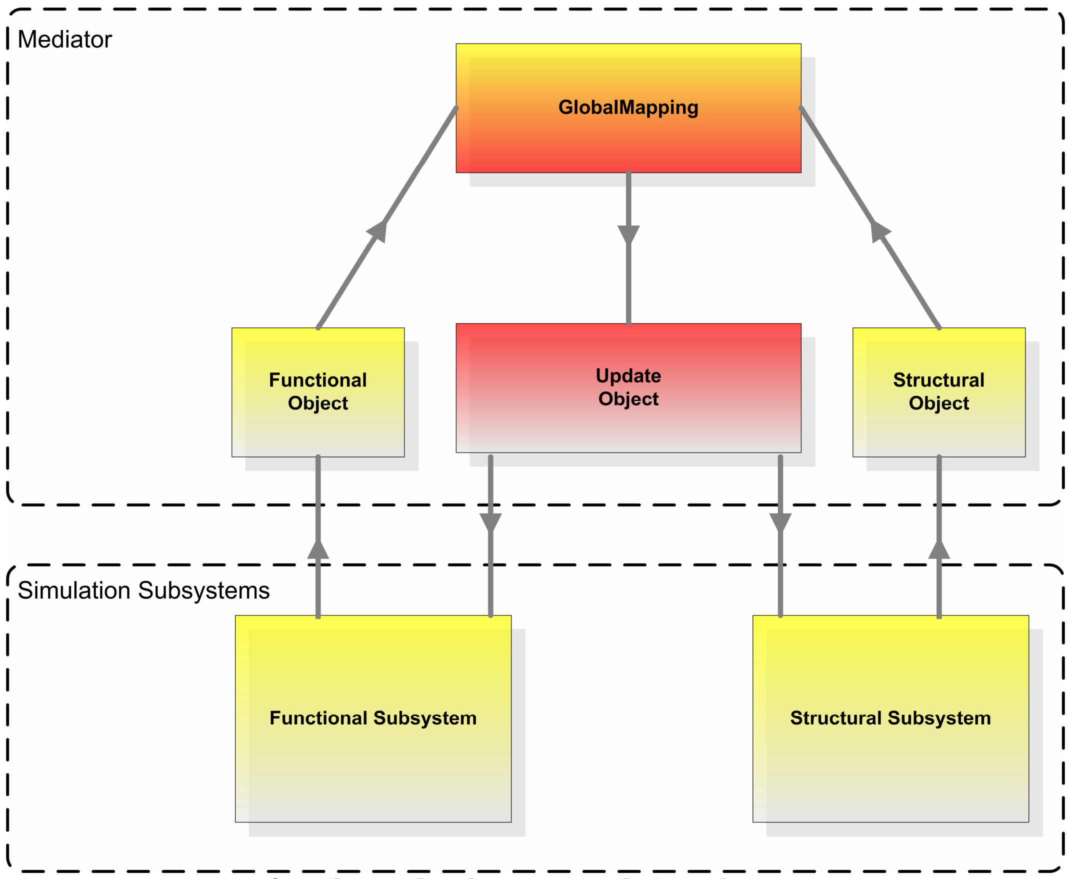


Figure 21: Transport of attribute values between simulation subsystems.

Values from the simulation subsystems are stored in a *Functional* or *Structural Object*, depending on the subsystem, and are transported to *GlobalMapping*. After the mapping process, *GlobalMapping* stores the new attribute values in *Update* objects and redirects these objects to the simulation subsystems.

The class diagram in Figure 22 shows the object classes and, where appropriate, the relationships between these entities. I will start with the description of the class *Attribute*, which implements the interface *AttributeTypeInterface*. The corresponding object saves one simulation attribute value with name and type of the attribute. This object contains three attributes: name, type and value. The first two attributes are from type string. The last attribute must be able to save a primitive numeric data type. In Java the wrapper class *Number* allows saving a numeric value and returns the value in each primitive data format. All attributes in the class are private and can only be accessed or changed via the five methods which are specified in the interface. The first method *getName()* returns the name of the attribute as a string. The second method *getType()* returns the type of the attribute as a string. The value of the attribute is returned as a *Number* object by the third method. To chance the attribute value, the class contains the method *setValue(Number value)*. It is possible that during runtime a copy of the object is required. Therefore, the method *copyAttribute* returns a deep copy of the object form type *AttributeTypeInterface*.

An *Attribute* instance is the smallest storage unit for the transport of values in the mediator. In the simulation tools the simulation attributes are combined in units. For instance, in an agent simulation all attributes belong to one agent or in a biochemical simulation a gene concentration belongs to one gene. To represent this in the mediator design various *Attribute* objects can be combined in an instance of the classes *StructuralObject*, *SimulationObject* or *UpdateObject*. The design of these object classes is also shown in the class diagram in Figure 22. These objects can be associated with the objects shown in Figure 21 in the following way. The class *StructuralObject* in the class diagram represents the *StructuralObject* in the illustration. Also, the class *UpdateObject* typifies *UpdateObject*. The name of the class which represent the *FuncitonObject* is *SimulationObject.* The reason to name the corresponding class *SimulationObject* is that objects of this type are also used for the mapping process.

In the following, I describe at first the interface structure and afterwards the structure of the three transport object classes.

By a closer inspection of the interface *UpdateObjectInterface* bears resemblance to the interface *MediatorObject*. This similarity is intended to get a clear differentiation between objects which are corresponding for the transport of values from the simulation subsystems to the mediator and vice versa. An alternative design solution could be to use one interface and to create two abstract classes which implement this interface.

To avoid confusion between an instance of a class in an object-oriented sense and the name for the objects for the transport of simulation tool values, the last mentioned objects are called in the following transport objects. Both interfaces *MediatorObject* and *UpdateObjectInterface* contain the following four methods. First of all the method *getName()*, which returns the name of the corresponding simulation tool unit (gene name, agent name,…). To identify a transport object in the mediator, each transport object gets a unique identification number (ID). The method *getID()* returns this ID. As mentioned before, in one transport object various *Attribute* objects can be enclosed. All these objects are saved in an internal data structure and the method *getAttributeList()* returns the data structure. Furthermore, it must be possible to add a new *Attribute* object to any existing transport object. Therefore, the interface specifies the method *addAttribute(attribute:AttributeTypeInterface)*.

The class specification of *UpdateObject* and *SimulationObject* are equal. Unlike the principle of object-oriented software design, to reuse existing structures, I recommend two object classes with the same behaviour. There are two reasons for this decision. In the first place, a splitting in two classes enhanced the understandability of the system design. Moreover, during runtime the system can decide on the basis of the object type how to handle the object.

In matter of the equality of the two classes, only the structure of *SimulationObject* will be explained in place of both. Each object of type *SimulationObject* has a name, a global unique identifier and can save one or more *Attribute* objects. *SimulationObject* implements the interface *MediatorObject* and *UdpateObject* implements *UpdateObjectInterface*.

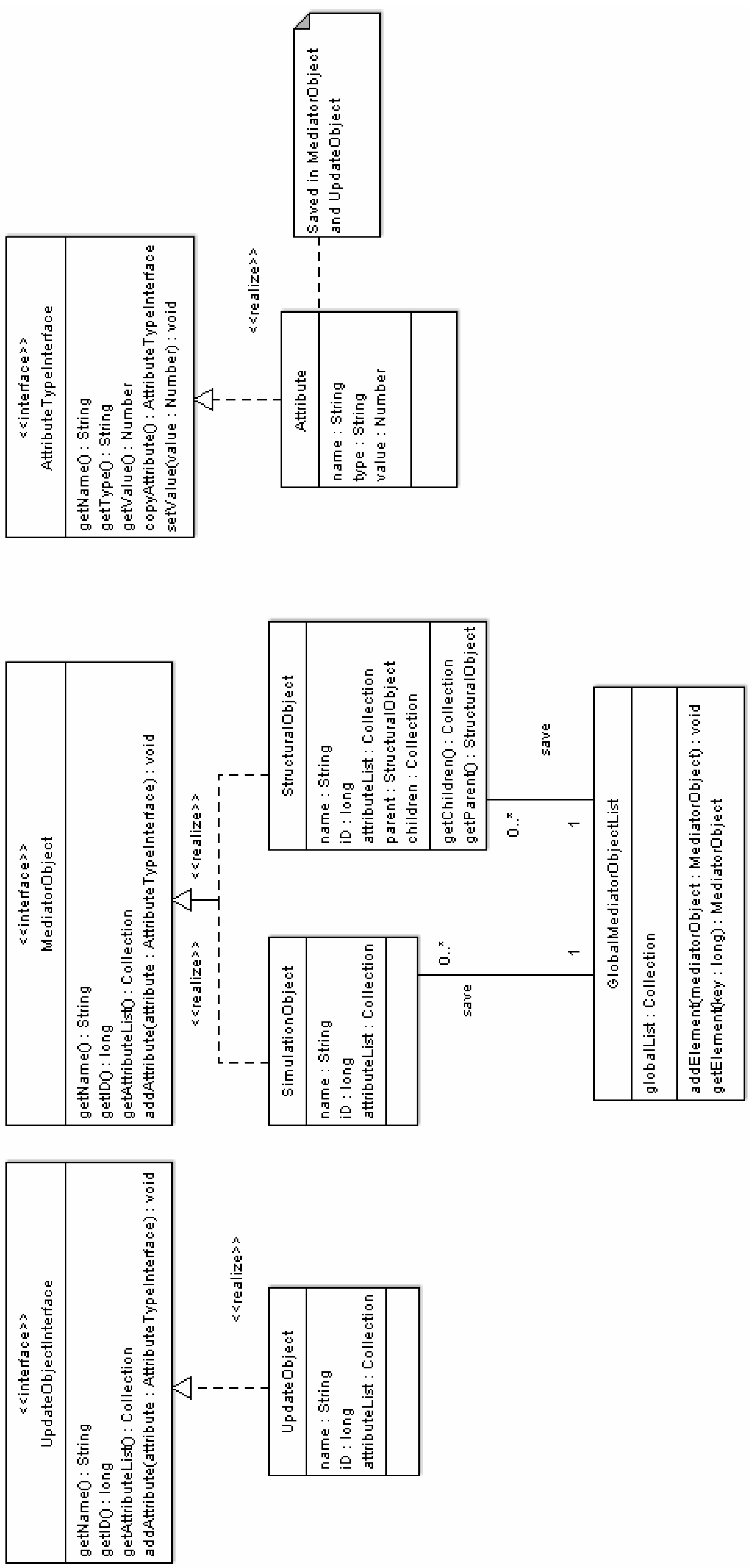

<<interface>>
AttributeTypeInterface
getName() : String
getType() : String
getValue() : Number
copyAttribute() : AttributeTypeInterface
setValue(value : Number) : void
<<realize>>
Attribute
name : String
type : String
value : Number
Saved in MediatorObject and UpdateObject
<<interface>>
MediatorObject
getName() : String
getID() : long
getAttributeList() : Collection
addAttribute(attribute : AttributeTypeInterface) : void
<<realize>>
<<realize>>
StructuralObject
name : String
iD : long
attributeList : Collection
parent : StructuralObject
children : Collection
getChildren() : Collection
getParent() : StructuralObject
SimulationObject
name : String
iD : long
attributeList : Collection
save
0..*
1
save
0..*
1
GlobalMediatorObjectList
globalList : Collection
addElement(mediatorObject : MediatorObject) : void
getElement(key : long) : MediatorObject
<<interface>>
UpdateObjectInterface
getName() : String
getID() : long
getAttributeList() : Collection
addAttribute(attribute : AttributeTypeInterface) : void
<<realize>>
UpdateObject
name : String
iD : long
attributeList : Collection

Figure 22: Mediator object classes for the transport of simulation attribute values.

The object class *StructuralObject* contains additionally two attributes in order to save information about the simulation environment. In this case, I mean with environment other simulation elements which are connected in the simulation with the actual object. Therefore, it is possible two save in each object of type *StructuralObject* a list of objects from the same type. The method *getChildren()* returns all connected transport objects which are in a lower hierarchy class. For instance, objects which save attributes of the simulation subsystem *GroIMP*, which uses a graph for the storage of the simulation objects (values), the method return all successor nodes. On the other hand, the method *getParent()* returns all transport objects which are in a higher hierarchy class. In the case of GroIMP this means the return object is a *StructuralObject*, saving the attributes of the parent node.

During runtime, quite a few objects from type *SimulationObject* or *StructuralObject* are used for the transport of simulation values. For a quick and ease access, all MediatorObject are saved in a hash map. The unique ID of each transport object is used as a key element. The corresponding transport object with the ID is the value element in the map. Therefore, a fast access to each *MediatorObject* via the ID is feasible. As a matter of course, only one hash map can exits in the system to guarantee a unique mapping. To reach this, the class *GlobalMediatorObjectList* has to be implemented as a singleton. Hence, only one instance of the class, and for this reason, only one hash map exists. To add a query for a *MediatorObject,* the class provides two methods:
addElement(mediatorObject:MediatorObject) to add a new object to the hash map and *getElement(key:long)* to query for a object in the map.

4.5. Mapping

Beside the connection of the simulation subsystems to the mediator and the transport of simulation values, the mayor function of the mediator is to allow the users to define dependencies between existing simulation tools. In the following, I present the design of the mapping package and different mapping categories.

In the first place, only the user can create a mapping between simulation tools and the system can only assist him with a simple as well as clear mapping structure. All varied mappings between simulation subsystems are described in the chapter *Requirements* in Figure 16. It is indispensable for the user to know all involved model details before he can create a mapping.

In general, the mediator provides two different types of mappings. I call the first one pre–mapping and the second one main-mapping. Both mappings have the same functionality. The only different is the time of execution. A simplified description of one mediator step illustrates the difference. In first place, all required information for the mapping is transported from the subsystem to the mapping unit. Then, the pre-mapping is executed. Afterwards all connected subsystems are updated. Subsequently the main-mapping is performed. To place emphasis on this difference, a simple statechart diagram clarifies the correlation between mapping and mediator step (see Figure 23). The diagram contains two main states*, Wait for user command* and *Perform mediator step*.

In the first state the system waits in an idle state for a user’s command to perform a mapping step or to end the simulation. The second state is composed of five sub states. After the user triggers a new mediator step the system changes to the state *Update simulation subsystems*. At this point all involved simulation subsystem values in the mediator are updated. This means that existing transport objects are updated, or if so far no transport object exists for the simulation attribute, a new one must be produced. Afterwards, the system switches over to the *Execute pre-mapping* state. Here, the system executes the user defined pre-mapping in the user defined sequence. New instances of type *UpdateObject* saves, after the mapping, the return values for the update process of the simulation subsystems. The update process is performed in the next system state *Update simulation subsystems*. The system behaviour in the next two states is similar to the earlier states. In *Execute main-mapping* all mappings of type main-mapping are accomplished. Just like before, the mapping unit creates new instances of *UpdateObject*. After updating the simulation subsystems, in the state *Update simulation subsystems,* the mediator goes back to the idle state, *Wait for user command*.

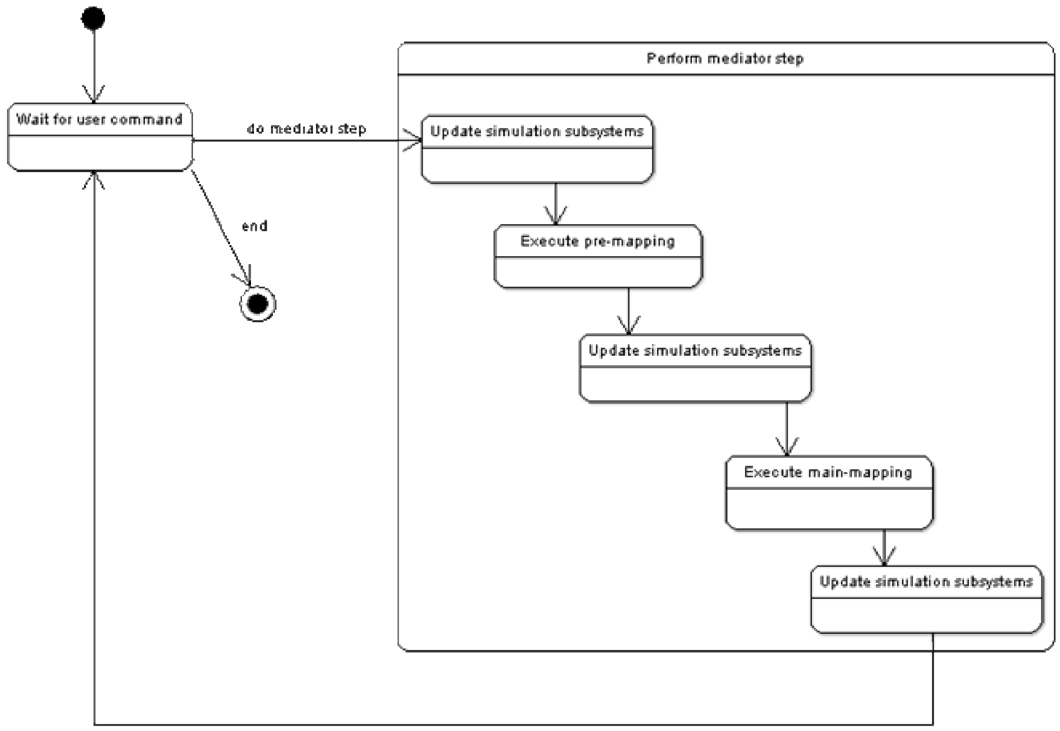


Figure 23: Statechart diagram to illustrate the correlation between one mediator step and the mapping.

Besides the global differentiation between pre-mapping and main-mapping in both categories a further differentiation is necessary. In the simulations two categories of simulation attributes occur. The first category contains the unique attributes inside the simulation. Unique means here that only one attribute of a special type exists in the simulation. In analogy to a programming language these attributes can be called global or independent attributes. The name of the attribute suffices to identify the attribute. The opposite of an *independent attribute* is a so called *dependent attribute*. In a simulation more then one attribute of the same type exists in different subcategories. To identify one of these attributes a set of dependencies to other attributes must be defined. To remain in the analogy of programming language, local attributes in different methods can be named identically. An explicit identification required the name of the method in which the attribute is defined. In the simulation, an attribute can depend on the environment, the position of the corresponding object in the 3D simulation or the position in the used data structure (in GroIMP for instance a graph). It is also possible that a defined dependence fits to more than one attribute. Examples for an independent simulation attribute are an attribute in a structural simulation which saves the global concentration of a hormone, or a constant value in the simulation like a gravitational constant. In addition, a global attribute which saves the branching angle between the stem and branches. On the other hand, an example for a dependent attribute is the concentration of a hormone in a special segment in the plant. For instance, a structural simulation, which is based on the XL approach, separates an entire plant in various small elements. Each element contains a variable for the local gene concentration. In a multi agent simulation a multitude of agents of the same type exists and interacts with their environment. A possible selection criterion for a set of agents could be the concentration of a special element in the environment. Another selection criterion for a simulation attribute in GroIMP is the position in the graph. A possible query can include the parent or child nodes and their current attribute values.

Recapitulate the exigency of a differentiation between dependence and independence simulation attribute is that the mediator must be able to query the right attribute. The query for an independent attribute is quite simple. The mediator has to know only the name of the simulation and the name of the attribute. In the case of a dependent object a query is more complicated. Here, the mediator has to know beside the name of the simulation and the name of the attribute which dependencies between the demanded and other attributes exist. Furthermore, a query for an independent attribute returns zero or one attribute, in contrast to a query for a dependent attribute, which can return zero or unlimited attributes. The mapping unit of the mediator must be able to handle these two kinds of mappings. They are called *dependent* and *independent* mapping. By reason of the multitude of conceivable dependencies between a simulation attribute and its surroundings, the mediator can only provide a well defined structure to query for a dependent object. The actual dependency must be implemented by the user which is endued with the simulations.

It is necessary for the simulation process that the mediator provides the user an opportunity to control the mapping process. Consequently, there are two possibilities. First of all, the mapping is executed in each mediator step. I call this kind of mapping *update dependence*. The reason therefore is that in each mediator step an attribute value is updated (overwritten) by another attribute value via the mapping. In the second place, the user can define a restriction under which the mapping will be executed. For instance, the user defines a threshold and a mapping will only be allowed, if the current attribute value in the pre-image[26] is lesser than the threshold. This kind of mapping is called *threshold dependence*.

In conclusion, for each new mapping between simulation subsystems, the user has to decide which mapping is required. To give an overview of all mappings Figure 24 illustrates all mapping types in a binary tree. A complete mapping between two simulation tools (both tools need not to be necessarily different) enfolds all sub mappings in a path in the tree from the root '*Mapping'* to an end node. For instance, a mapping between a structural and a functional simulation can be pre-mapping, independent and the way of execution is threshold dependence. Alternatively, the mapping can be a main-mapping, dependent and the way of execution is update dependence.

[26] The mapping can be esteemed as a relation between simulation attribute values (pre-image side of the relation) which are send to another simulation attribute values (image side of the relation).

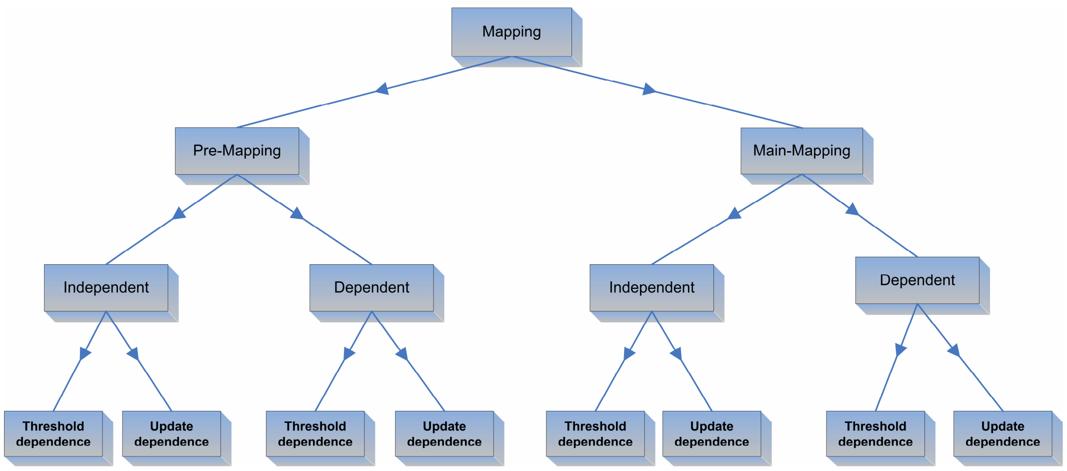


Figure 24: All possible mapping categories. Each path in the binary tree from the root '*Mapping*' to an end node contains one mapping.

With reference to the various mapping combinations, the design of the mapping unit of the mediator is split in three parts. The design of the object classes and the relationship between these entities are shown in Figure 25. Before I will describe the UML class diagram in detail, I will give a short outline of the mapping unit requirements. The user must be able to define, dependent on the simulation subsystems, any possible dependent mapping. Also, the user must be able to define the threshold dependencies with a threshold and a relational operator. It is also feasible that dependences between multiple mappings exist. The execution order is important for the correctness. Therefore, a collection is required which is able to save the mappings in a user defined order.

The mediator is responsible for the management and execution of the mappings. The class *MediatorSupervision* assumed control of all mappings. To guarantee a unique control, the class must be implemented after the singleton pattern. The corresponding object includes a *MappingList* object. In addition, the object includes two methods to execute the pre-mappings and main-mappings, respectively. Before I describe the relation between *MediatorSupervision* and *GlobalMapping*, the storage of the mappings is described. Just like *MediatorSupervision*, the class *MappingList* has to be implemented after the singleton pattern. As mentioned before, the object is responsible to save all pre- and main-mappings in a sorted list. Therefore, the object contains one list for pre-mappings and one list for main-mappings. In addition, *getter* und *setter* methods for each object are provided. The *setter* methods allow the user to add a new mapping at each place in the sorted list. In this way, the scientists can sort the mappings and make sure that the mappings are executed in the right order.

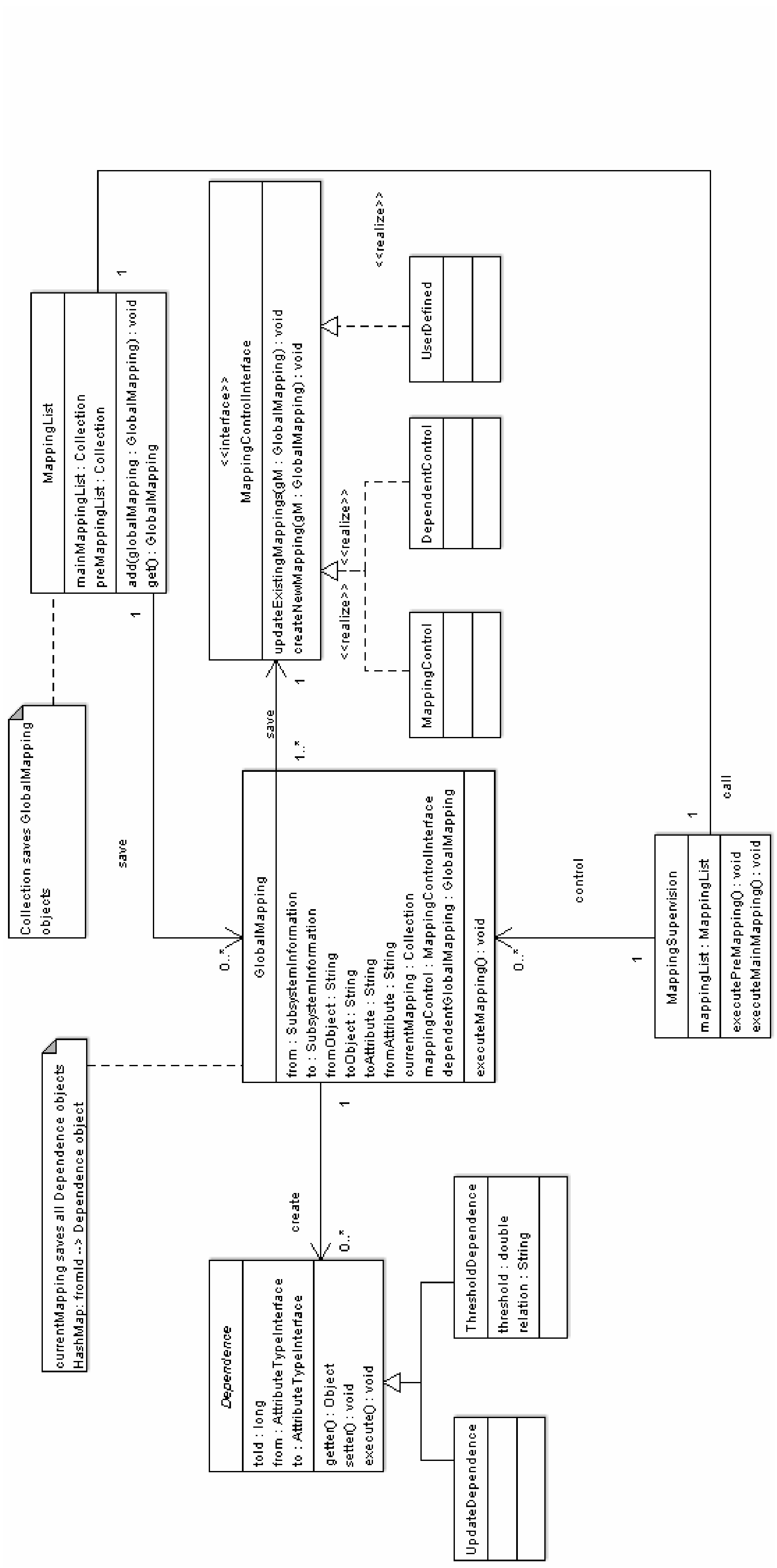

MappingList
mainMappingList : Collection
preMappingList : Collection
add(globalMapping : GlobalMapping) : void
get() : GlobalMapping
<<interface>>
MappingControlInterface
updateExistingMappings(gM : GlobalMapping) : void
createNewMapping(gM : GlobalMapping) : void
<<realize>>
UserDefined
<<realize>>
DependentControl
<<realize>>
MappingControl
Collection saves GlobalMapping objects
save
save
1
1
1
1..*
0..*
GlobalMapping
from : SubsystemInformation
to : SubsystemInformation
fromObject : String
toObject : String
toAttribute : String
fromAttribute : String
currentMapping : Collection
mappingControl : MappingControlInterface
dependentGlobalMapping : GlobalMapping
executeMapping() : void
control
0..*
1
call
1
MappingSupervision
mappingList : MappingList
executePreMapping() : void
executeMainMapping() : void
currentMapping saves all Dependence objects
HashMap: fromId --> Dependence object
create
1
0..*
Dependence
toId : long
from : AttributeTypeInterface
to : AttributeTypeInterface
getter() : Object
setter() : void
execute() : void
ThresholdDependence
threshold : double
relation : String
UpdateDependence

Figure 25: Class diagram for the mapping unit of the mediator.

The main part of the UML diagram is divided in three classes: the abstract class *Dependence*, the class *GlobalMapping* and the interface *MappingControlInterface*. I explain the functionality of these classes briefly beforehand a detail description of each class follows. An object of *GloabalMapping* stores all necessitated information for a mapping. The execution of the mapping is performed in the child classes of the abstract class *Dependence*. All classes, which implement the interface *MappingControlInterface,* are used for the communication with the transport unit of the mediator.

The object class *GlobalMapping* combines all information which are mandatory for the mapping. For each mapping, the following elements of the source and destination simulation subsystem have to be identified. First of all, the name of source and destination system must be defined. Each simulation subsystem is described in the mediator with a *SubSystemInformation* object. In the second place, the objects or system specific abstraction units which save the required simulation attributes have to be known. Furthermore, the names of these two attributes are necessary for the mapping. The object class *GlobalMapping* uses an attribute, starting with the substring ‘from’, for the saving of all elements of the source simulation subsystems. On the other hand, the attributes which save the names of the destination simulation subsystem elements start with the substring ‘to’. Therefore, the class contains the following six attributes: *from*, *to*, *fromObject*, *toObject*, *fromAttribute* and *toAttribute*.

In principle, the *GlobalMapping* objects save only the required information for the mapping. The real execution of the mapping is performed in the classes which are derived from the abstract class *Dependence*. As mentioned before, there are two ways to execute a mapping. In the first place, the execution takes place in each mediator step. This kind of mapping is called update dependence and is realised in the class *UpdateDependence,* which is inherited from Dependence. The other execution way is the so called threshold dependence. In this place, a mapping is only executed if the user defined condition is true. This kind of mapping is mapped in the system design by the class *ThresholdDependence*. Three attributes are contained in the abstract class *Dependence*. The first one saves the ID of the destination simulation subsystem. Attribute ‘to’ and ‘from’ contain an object of type *AttributeTypeInterface*. In principle, the method *execute* calls the attribute ‘from’ and update the values in the attribute ‘to’. In the case of update dependence the method perform this in each method call. Considering a threshold dependence, the defined condition is tested in each method call and if the result is true the ‘to’ attribute is updated. Each *ThresholdDependence* object contains two additional attributes, ‘threshold’ and ‘relation’.

In each *GlobalMapping* object a list of *Dependence* objects are enclosed. A list is required, because in the case of dependence mapping the subsystems can contain more than one simulation attribute with the same name. Therefore, for all of these, a new Dependence object is created and saved in this list. In the case of independent mapping, this list contains only one entry. This attribute in *GlobalMapping* is called ‘currentMapping’. A dependent mapping requires knowledge about other simulation subsystem attributes and objects. The mediator has to identify the other elements in the simulations. Therefore, each *GlobalMapping* object saves a list of other *GlobalMapping* objects in the attribute *dependentGlobalMapping*.

For the mapping process, a synchronisation of the simulation attributes values in the simulation subsystem and the corresponding values in the mediator is necessary. Synchronised means here that after a mediator step the values in the mediator and in the simulation subsystem are equal. After a step in the simulation subsystem all required values are read from the mediator and stored in an *Attribute* object (see chapter *Transport of values*). Also, after each mediator step, all *Attribute* objects are sent to the *Control* object to update the corresponding simulation subsystem attributes. In the case of independent mapping it is quite easy. The class *MappingControl* implements the interface *MappingControlInterface*. This class is responsible for the identification of the required *Attribute* objects and for the interaction with the responsible control objects. Simplified, the application flow is the following: In the first place, the *GlobalMapping* object saves the information. The *MappingControl* object uses this information to find the required simulation objects and attributes in the internal data structure or in the case of a not existing object, the system calls directly the simulation subsystem. These filtered attribute values by the *MappingControl* object are saved in an object of type *Attribute*. By an independent mapping only two attributes are possible. The reason therefore is that the names of the simulation subsystem attributes have to be unique for an independent mapping. Hence, one attribute for the source and one attribute for the destination system should be found. These are saved in the corresponding object of type *UpdateDependence*. After the execution of the update of the 'to' attribute in this object, the *MappingControl* object sends an object from type *UpdateObject* to the corresponding control object to update the simulation. The same process for a dependent mapping is more complex. As mentioned before, more than one attribute with the same name can exits in the simulation subsystem. Only the user knows the dependencies to identify the required attributes. Therefore, only the user can implement a class which can find the attribute in the simulation subsystems. To support the user by the implementation the new class shall realises the interface *MappingControlInterface*. In principle, the user can copy the class definition from *MappingControl* and change only the behaviour of the methods which are responsible to identify the simulation attributes. In this way, the mediator supports all variations of dependent mappings. The only precondition is the user's ability to define the dependency. Beside the identification of the simulation subsystem attributes, the difference between dependent and independent mapping in the mediator are infinitesimal. The method *executeMapping* in *GlobalMapping* calls in the case of an independent mapping only one *Dependence* object. Considering a dependent mapping, the method has to call one or more *Dependence* objects.

4.6. Log and statistic unit

In the first place, I explain the log unit of the mediator and afterwards a short explanation of the statistic unit follows. The crucial point is that the mediator's log unit is completely independent from the mapping. This means, each simulation attribute from a current connected simulation subsystem can be recorded with the log unit independent from the user defined mappings.

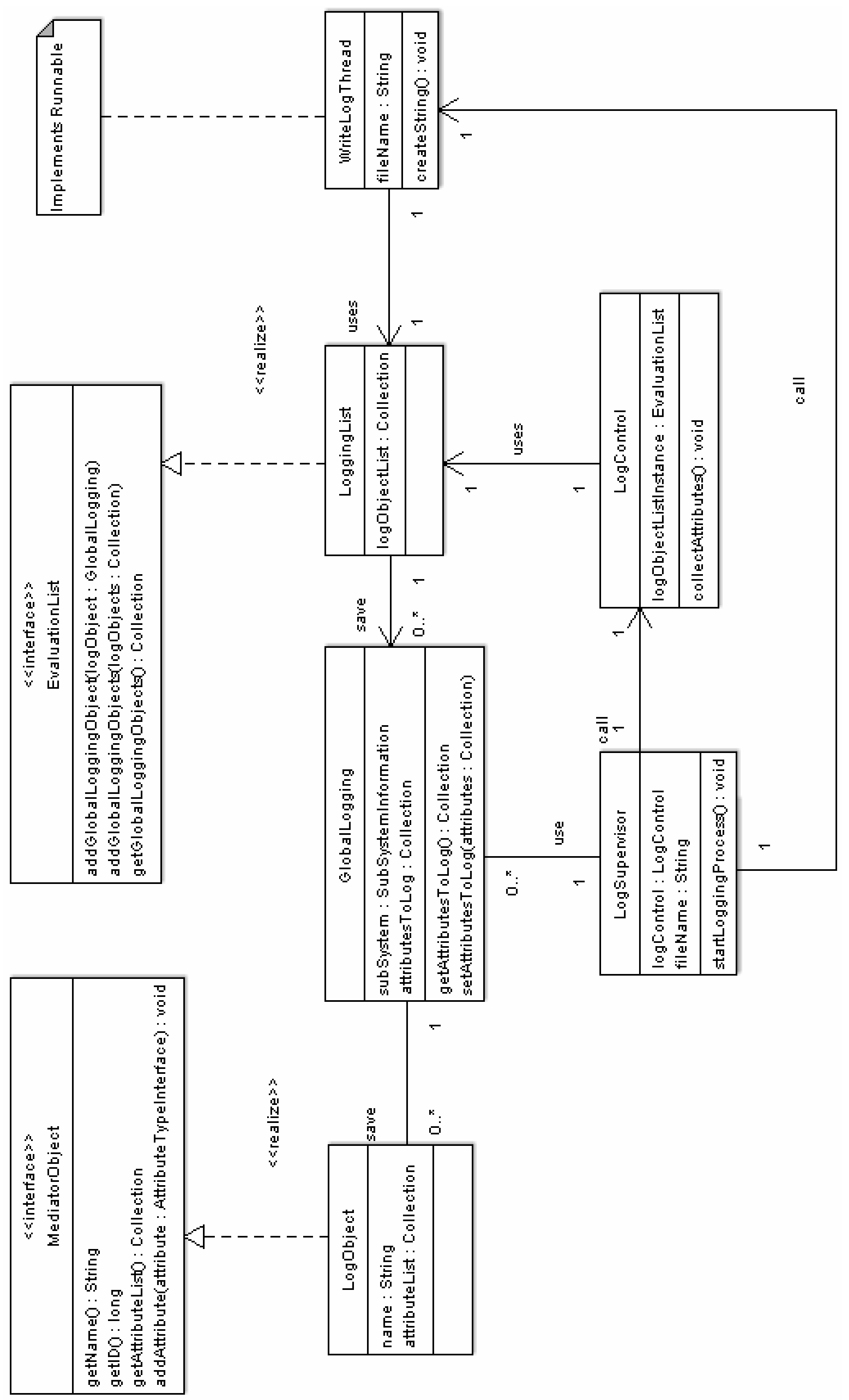

Implements Runnable
WriteLogThread
fileName : String
createString() : void
<<realize>>
uses
<<interface>>
EvaluationList
addGlobalLoggingObject(logObject : GlobalLogging)
addGlobalLoggingObjects(logObjects : Collection)
getGlobalLoggingObjects() : Collection
LoggingList
logObjectList : Collection
uses
LogControl
logObjectListInstance : EvaluationList
collectAttributes() : void
call
save
0..*
1
GlobalLogging
subSystem : SubSystemInformation
attributesToLog : Collection
getAttributesToLog() : Collection
setAttributesToLog(attributes : Collection)
use
LogSupervisor
logControl : LogControl
fileName : String
startLoggingProcess() : void
<<interface>>
MediatorObject
getName() : String
getID() : long
getAttributeList() : Collection
addAttribute(attribute : AttributeTypeInterface) : void
<<realize>>
LogObject
name : String
attributeList : Collection

Figure 26: Class diagram for the log subsystem.

An overview of the log design of the mediator is shown in Figure 26. In order to log an attribute in a simulation subsystem, the mediator needs the *SubsystemInformation* object, and a corresponding object from type *LogObject* with name and type of the simulation attribute. They are stored in an object of type *GlobalLogging*. A simulation subsystem can include more than one object, and therefore, a *GlobalLogging* object can save a list of *LogObjects*. In addition, the object class contains two methods *getAttriubtesToLog* and *setAttributesToLog*. The class *LoggingList* stores all these *GlobalLogging* attributes. To guarantee a unique list in the system, the *LoggingList* class must be implemented as a singleton. This class implements the methods of the interface *EvaluationList* and contains three methods. Method one adds a *GlobalLogging* object to the internal collection, and method two is able to add a collection to the data structure. The third method returns the internal data structure. Simular to the mapping unit, a communication with the control classes for the connected simulation subsystems is necessary to extract the current system values. This functionality is implemented in the class *LogControl*. To get a direct access to the object, which saves the attribute information the user is interested in, the *LogControl* class stores a reference to *LoggingList*. This attribute is called 'logObjectListInstance'. Furthermore, the class includes a method *collectAttribute*. In principle, this class calls all connected simulation subsystem, which include a required attribute for the log process. The necessitated information are stored in the *GlobalLogging* objects in the internal collection of *LoggingList*. If the method finds the attribute in the simulation subsystem, the corresponding *GlobalLogging* object will be updated with the present simulation attribute value. At this point, a differentiation between independent and dependent objects is not necessary. The mediator reads only the values out of the simulation system without changing the original values (systems). In the log file, the name of the corresponding simulation system, the object name and the attribute with name, type and value are listed.

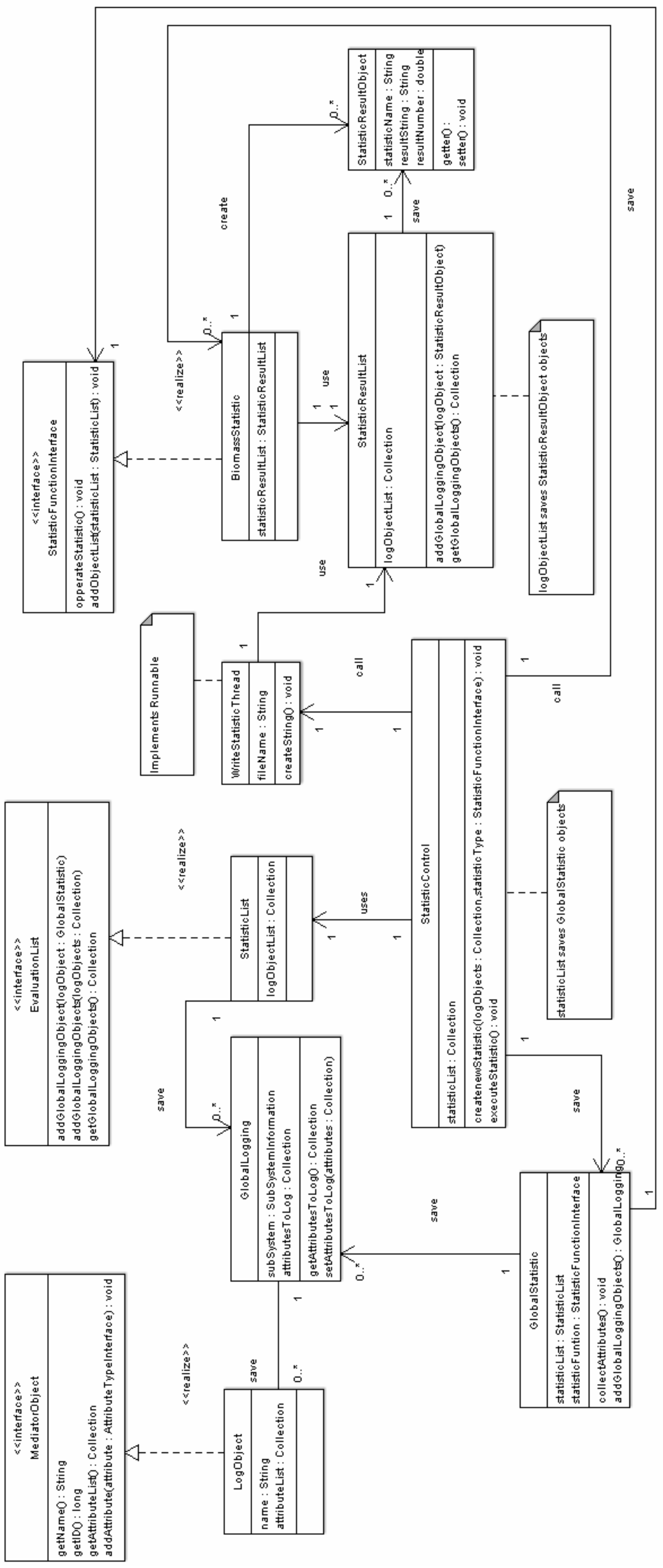
The control over the logging process takes the singleton object of the class *LogSupervisor*. For a direct access to the log process a reference to the *LogControl* object is saved in *LogSupervisor*. To save the log information to a file, the user has to define a log file name. This name is also saved in *LogSupervisor*. Beside these two attributes, the class contains only one method: *startLoggingProcess*. First of all, this method calls the method *collectAttribute* in *LogControl*. As descriped before, this method searches the simulation attributes in the simulations and updates the *logObject* instances. Afterwards, the method of *LogSupervisior* creates a new thread and calls the *run* method in *WriteLogThread*. By then, the new thread writes all information, which are stored in the *GlobalLogging* objects, into a file. Beforehand, these information are sorted in a user defined order. The default sorting sequence is alphabetical for each subsection. A log subsection contains the name of the simulation subsystem, the name of the simulation object and the corresponding attributes, with name, type and value.

The statistic unit of the mediator allows the user to create statistics of the simulation. This enables the user to define his own statistics, and the mediator calculates in each mediator step the results. Statistics can be for instance, the average of a gene concentration in a functional simulation, the average length of all leaves in a structural simulation or the biomass of the simulated biological system. Most of the statistics are simulation specific, and therefore, the mediator can only provide a clear interface structure and basic functionality for the statistic unit.

The calculation of the statistic results dependent on the simulation and the used simulation subsystems. For instance, the calculation to get the biomass of a tree is different in nature from the calculation of wheat or flowers. Accordingly, the user has to create for each new statistic type a new class for the calculation of this statistic. The mediator calculates in each mediator step all users' defined statistics, and saves the results in a file.

Like the log unit, the statistic unit is in the same way completely independent form the mapping unit. I have decided to use elements of the log unit for the identification of the simulation attributes, which are required for the statistic process. An overview over the statistic unit is presented in Figure 27. On the left hand side, the interface *MediatorObject,* the class *LogObject* and *GlobalLogging* are well known from the log unit. Similar to the log unit, the statistic unit includes a class which implements the *EvaluationList* interface. This class is called *StatisticList* and includes a collection to save *GlobalLogging* objects. All these objects are required to identify the necessitate simulation attributes for the statistic process. Therefore, the user defines a set of objects of type *LogObject* and *GlobalLogging,* which are stored in the singleton object of type *EvaluationList*. Beforehand, the user has to create his own statistic calculation class, or he has to choose an existing one. All these classes have to implement the interface *StatisticFunctionInterface*. In the class diagram an example class *Biomasse* is shown. The interface defines two methods *opperateStatistic()* and *addObjectList(statisticList: StatisticList).* In addition, each new statistic function class have to provide an attribute to store the singleton object of *StatisticResultList*.

The application flow for a statistic process is descriped in the following. In the first place, the user creates or chooses a statistic function. Afterwards, he creates the instance of *LogObject* and *GlobalLogging.* They are saved together with a reference to the statistic function class in an object of type *GlobalStatistic*. It is also possible, to execute in one mediator step more than one statistic calculation. Therefore, the class *StatisticControl* is necessary, which takes the control over the statistic process, and can save more than one *GlobalStatistic* instance in its internal collection. At the end of each mediator step, the singleton object of type *StatisticControl* calls, for each *GlobalStatistic* object in its internal collection, the method *opperateStatistic* of the corresponding statistic function class (in the class diagram only *BiomassStatistic* is shown). This method creates for each calculation result an object of type *StatisticResultObject*, which saves the name of the statistic function. The object saves the result in a string and in a double value attribute. All these objects are stored in *StatisticResultList*. After the execution of all statistic calculations the collection in *StatisticResultList* includes all results. In the next step, all these results are output in a file with the help of the class *WriteStatisticThread*. This method implements the interface *Runnable,* and the control object of type *StatisticControl* calls the *run* method. Multithreading is usable at this point without concern. No changes on the *StatisticResultList* objects are allowed before the writing process is completed. Hence, 'no lost update' or other synchronisation errors can occur. In general, the writing process is much faster than the next mediator step, and therefore a delay of the main thread is unlikely. The same technique is used by the log unit.



<<interface>>
StatisticFunctionInterface
opperateStatistic() : void
addObjectList(statisticList : StatisticList) : void
BiomassStatistic
statisticResultList : StatisticResultList
StatisticResultList
logObjectList : Collection
addGlobalLoggingObject(logObject : StatisticResultObject)
getGlobalLoggingObjects() : Collection
StatisticResultObject
statisticName : String
resultString : String
resultNumber : double
getter() :
setter() : void
logObjectList saves StatisticResultObject objects
Implements Runnable
WriteStatisticThread
fileName : String
createString() : void
StatisticControl
statisticList : Collection
createnewStatistic(logObjects : Collection,statisticType : StatisticFunctionInterface) : void
executeStatistic() : void
statisticList saves GlobalStatistic objects
<<interface>>
EvaluationList
addGlobalLoggingObject(logObject : GlobalStatistic)
addGlobalLoggingObjects(logObjects : Collection)
getGlobalLoggingObjects() : Collection
StatisticList
logObjectList : Collection
GlobalLogging
subSystem : SubSystemInformation
attributesToLog : Collection
getAttributesToLog() : Collection
setAttributesToLog(attributes : Collection)
GlobalStatistic
statisticList : StatisticList
statisticFuntion : StatisticFunctionInterface
collectAttributes() : void
addGlobalLoggingObjects() : GlobalLogging
<<interface>>
MediatorObject
getName() : String
getID() : long
getAttributeList() : Collection
addAttribute(attribute : AttributeTypeInterface) : void
LogObject
name : String
attributeList : Collection
<<realize>>
create
use
save
call
uses
1
0..*

Figure 27: Statistic unit design in a UML class diagram.

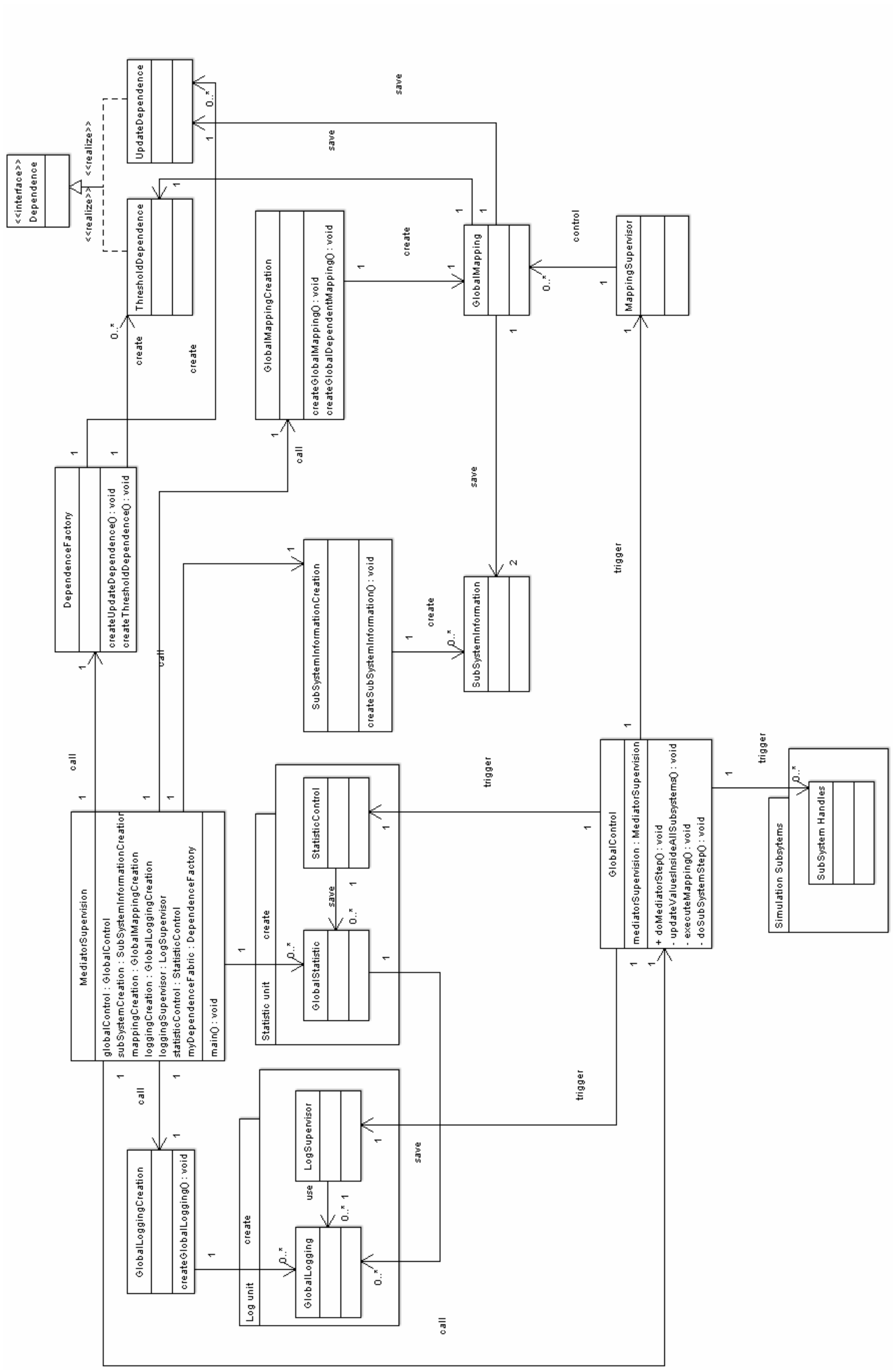
4.7. Global control

In this subsection I will explain the design model of the object classes which are responsible for the control of the mediator. The entrance point in the mediator system is the class *GlobalSupervision*. This class creates and initialises all essential mediator elements. Therefore, the class includes seven attributes. First of all, the class includes an object of type *GlobalControl*. After all initialisation processes the *GlobalSupervision* object gives the system control over to this object. With regard to a unique control of the entire system, the *GlobalControl* class is implemented according to the singleton pattern. The next six attributes of *GlobalSupervision* are used for the initialisation process and will be discussed in detail in the following.

In first place, the attribute 'subSystemCreation' includes the object of the class *SubSystemInformationCreation*. The function of this class is to create new *subSystemInformation* objects. At this point, the fabric pattern is particularly suitable for the design of the class and the relationships between all participants' classes. In second place, the attribute 'mappingCreation' stores the object of the class *GlobalMappingCreation*. According to the fabric pattern, this class includes two methods to create a dependent or an independent mapping. In addition, the class *DependentFactory* uses the same design pattern to create a *ThresholdDependence* or an *UpdateDependence* object. A reference to the *DependentFactory* instance is saved in the attribute 'myDependenceFabric'. In the same line the class *GlobalLoggingCreation* creates new *GlobalLogging* instances. The creation of a new *GlobalLogging* object needs the help of the class *LogSupervisor,* and therefore, a reference is saved in *GlobalSupervison*. In a similar way, the main class of *GlobalSupervision* use the reference to *StatisticControl* to create new *GlobalStatistic* objects.

To initialize the mediator, the main method of GlobalSupervision calls the objects in the following order. First of all, new *SubSystemInformation* objects are created. Afterwards, a set of mappings for the subsystems, which are represented by the *SubSystemInformation* instances, are created. Eventually, the log and statistic objects are defined optionally. The last command in the main method of GlobalSupervision calls the method *doMediatorStep* in *GlobalControl*.

GlobalControl includes the public method *doMediatorStep* and three private methods: *updateValuesInsideAllSubsystems, executeMapping* and *doSubSystemStep*. During the first call of *doMediatorStep* the user can define how often the method *doMediatorStep* is executed in rotation. Therefore, a small graphical control window is used. In each execution, the method *doMediatorStep* calls the private methods in the same class in the following order. First of all, the method *updateValuesInsideAllSubsystems* is called. Afterwards, *executeMapping* is called. In *executeMapping,* first all pre-mappings are executed and subsequent all main-mappings. Between the executions, all subsystems will be updated. A statechart diagram of this application sequence is shown in Figure 23. After the last simulation system update, the program flow jumps back to the *doMediatorStep* method, and the method *doSubSystemStep* is called. The function of this method is to perform one simulation subsystem step in each connected subsystem. Eventually, the *doMediatorStep* method starts from the beginning until the user defined iteration number is reached. In the class diagram the global control position of *GlobalControl* become visible. The class is connected with all control classes of the mediator subsystems, and with all handle classes for the control of the connected simulation subsystems.



<<interface>>
Dependence
<<realize>>
<<realize>>
UpdateDependence
ThresholdDependence
GlobalMappingCreation
createGlobalMapping() : void
createGlobalDependentMapping() : void
GlobalMapping
MappingSupervisor
DependenceFactory
createUpdateDependence() : void
createThresholdDependence() : void
SubSystemInformationCreation
createSubSystemInformation() : void
SubSystemInformation
MediatorSupervision
globalControl : GlobalControl
subSystemCreation : SubSystemInformationCreation
mappingCreation : GlobalMappingCreation
loggingCreation : GlobalLoggingCreation
loggingSupervisor : LogSupervisor
statisticControl : StatisticControl
myDependenceFabric : DependenceFactory
main() : void
Statistic unit
StatisticControl
GlobalStatistic
GlobalControl
mediatorSupervision : MediatorSupervision
+ doMediatorStep() : void
- updateValuesInsideAllSubsystems() : void
- executeMapping() : void
- doSubSystemStep() : void
Simulation Subsytems
SubSystem Handles
Log unit
LogSupervisor
GlobalLogging
GlobalLoggingCreation
createGlobalLogging() : void
save
create
call
control
trigger
use
1
0..*
2

Figure 28: UML class diagram shows the global control object classes of the mediator.

To sum up the software design, Figure 30 shows all packages of the mediator. The diagram shall give an overview of all packages and not in the first place a description of the relations between the different packages. All arcs in the illustration are used only to highlight the central position of the control package (*uk.ac.rothamsted.mediator.control),* which includes the classes *MediatorSupervision*, *MediatorControl* and all fabric classes as described in the last subsection. In reality, a connection and dependence of nearly all packages exists. For a better overview, only the relationships between the central control package and the subpackages are shown. All packages below *MediatorControl* are packages for the interaction with the simulation subsystems. Above *MediatorControl* all packages, which are used for the internal functionality of the mediator, are shown. For instance, a package for the mapping unit, one for the log unit as well as for the statistic unit or a package with utility classes, like file writer or file reader.

In general, the software architecture of the system is a mixture of a component based architecture and a variation of a three tier architecture. The first layer (Layer 0) includes the simulation subsystems. The communication and the internal data storage of the simulation attribute values are placed in the middle layer (Layer1). The third layer (Layer 2) includes the control unit and the mapping units. The software system is split in different components such as the mapping unit, the log unit as well as statistic unit and all simulation subsystems. Adding a new simulation subsystem or a new functionality is quite easy, and can be performed with only a few changes or settings in the existing code. A simplification of the system architecture is presented in Figure 29.

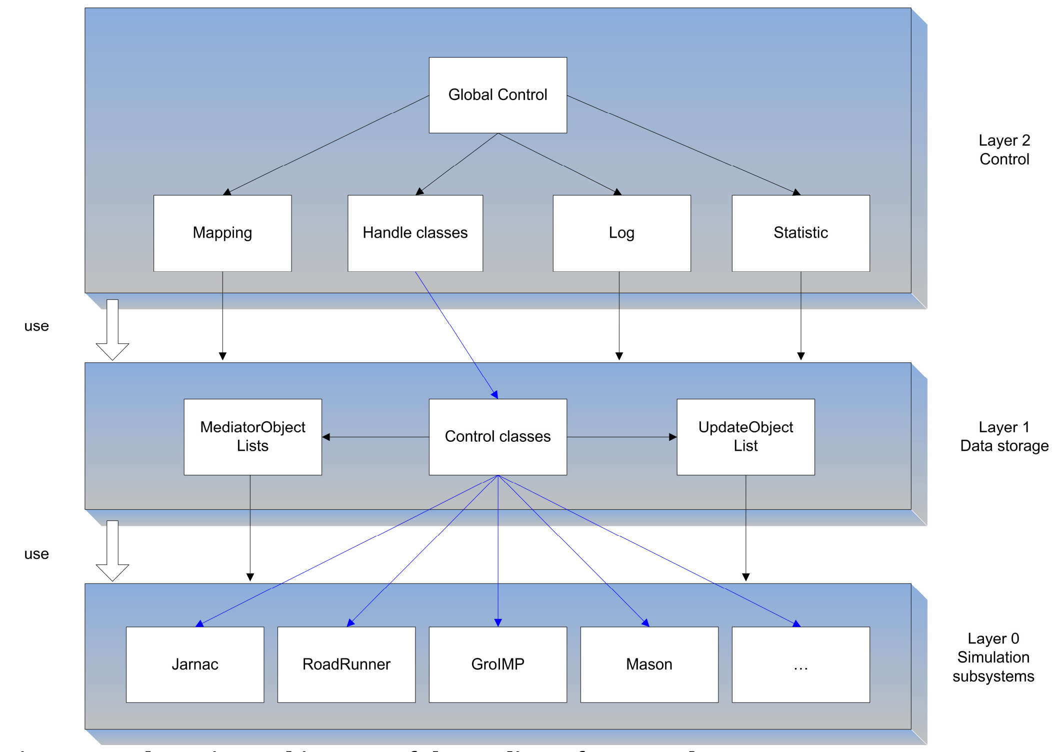


Figure 29: Three tier architecture of the mediator framework.

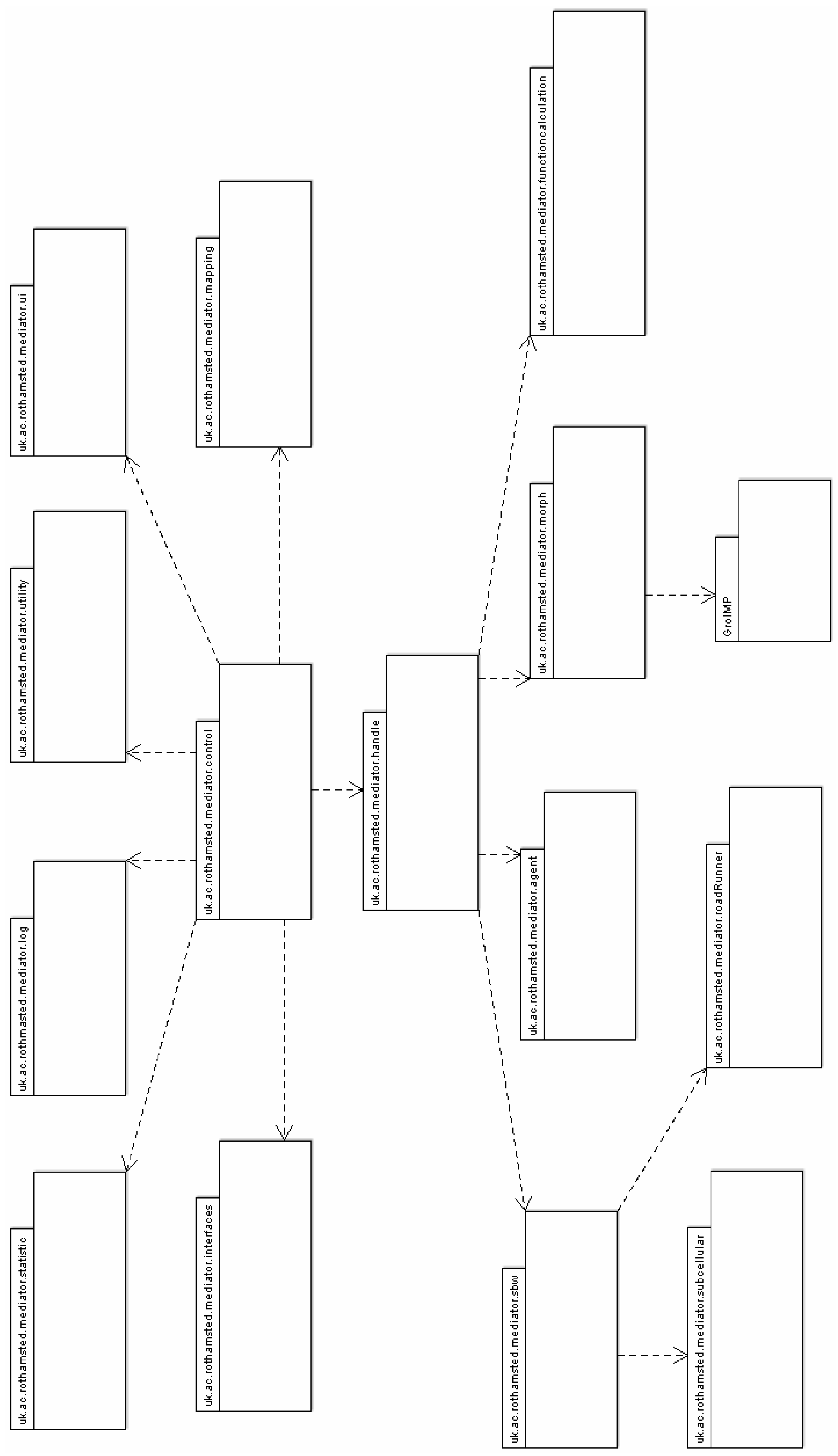

uk.ac.rothamsted.mediator.ui
uk.ac.rothamsted.mediator.mapping
uk.ac.rothamsted.mediator.functioncalculation
uk.ac.rothamsted.mediator.utility
uk.ac.rothamsted.mediator.morph
GroIMP
uk.ac.rothamsted.mediator.control
uk.ac.rothamsted.mediator.handle
uk.ac.rothmasted.mediator.log
uk.ac.rothamsted.mediator.agent
uk.ac.rothamsted.mediator.roadRunner
uk.ac.rothamsted.mediator.statistic
uk.ac.rothamsted.mediator.interfaces
uk.ac.rothamsted.mediator.sbw
uk.ac.rothamsted.mediator.subcellular

Figure 30: Package diagram of the mediator framework.

5. Implementation

The implementation chapter highlights the used programming techniques and gives a more detailed view of the key elements in the mediator framework. I start with general information about the used programming language and extension to optimize the performance. Afterwards, the control process of the class *Global-Control* is explained in more detail than in the previous chapter. Finally, a detailed description of the GroIMP interaction follows.

5.1. General implementation details

The mediator framework is completely implemented in Java version 1.5.0.9. The Java programming language allows creating a robust and platform independent software system. Furthermore, Java uses typed data types, which mean that all data types are independent from the used operation system and having the same value ranges and fixed lengths. In addition, the programming language is safe in reference to the memory management. Moreover, the language allows multi-threaded programs. Nearly for all existing programming language a Java interface or wrapper class exists so that Java can interact with programs in these languages. Nevertheless, the mediator framework has not used these Java extensions. The System Biology Workbench is based on fast and reliable broker architecture, with interfaces for nearly all existing programming languages. For more information about the SBW see the chapter *Fundamentals and Related Work*. This framework allows the mediator to add simulation tools, which are implemented in the different programming languages.

As a consequence of the numerousness transport objects instances in the mediator, a fast and effective data storage is required. Therefore, in the fist step each object in the mediator contains a unique identification number. The ID is from type long, with a value range of 2 * 9,223*10^{18} values. The Java Collection-API offers various data structures for the storage of objects. All these collections differ in their complexity. In comparison to data structures like *Vector* or *LinkedList*, a *HashMap* includes and extracts new values in the data structure in a constant time. For example, the complexity to find a specific element in an unsorted linked list is O(n). With reference to these variations in the access times the choice of the used data structure has a mayor influence of the system performance. In general, it is not possible to use only one type of data structure in all parts of the system. For instance, for a small amount of entries an *ArrayList* or a *LinkedList* have performance vantages in comparison to a *HashMap* data structure. The other way round, the optimal storage for a huge amount of objects with unique identifiers can be a *HashMap*. In addition, a type specific collection enhances the performance of the system. Therefore, the mediator framework does not use the standard Java collections. *Fastutil* is a collection of type-specific Java classes, which extend the original Java Collection Framework. More information about the *Fastutil* collection framework can be found at http://fastutil.dsi.unimi.it/ (accessed 28.07.2007). Besides the performance advantages, *Fastutil* collections provide additional features (like bidirectional iterators) that are not available in the standard classes (see http://fastutil.dsi.unimi.it/;accessed 28.07.2007).

Finally, I will point to the fact that in the mediator, especially in the log and statistic unit, a few lists have to be sorted during runtime. Therefore, objects of type *Attribute* have to be compared. Each object in Java that should be compared to other objects of the same type has to implement the *Comparable* interface. Therefore, the class *Attribute* implements the interface *Comparable* and overwrites the method *compareTo(Object o)*. The method compares the current object (this) to the object '*o*' (that), and will returns an integer value less than zero if *this* is smaller than *that*, zero if *this* and *that* are equal and a integer value bigger than zero if *this* is bigger than *that*. The method uses the attribute values of the objects to decide which return value will be returned. Each time the compare method is overwritten, the object has to overwrite the *equals()* method of *Object* to guarantee a error free compare method. The *equals(Object o)* method tests the equality of the current object (*this*) and the object o (*that*). For example, the method can use an integer attribute to test the equality. If both objects save the same value in the integer attribute the *equals()* method returns true otherwise false. A new *equals()* and a new *compareTo()* method requires a new method *hashCode()*, which calculates for an object a unique integer value (so called 'hashcode'). Hence, each object has to overwrite the public method *hashCode()* from Object. The implementation of the method is not as simple as at the first glance. It is indispensable to follow some rules by the implementation of this method. A detailed description of the right implementation of a *hashCode()* method is represented by Angelika Langer at her webpage[27].

5.2. Global control details

As mentioned in the previous chapter, the instance of *GlobalControl* is responsible for the global control of the mediator framework. After the initialisation of the system in *GlobalSupervision* the subsystem handling, the execution of the mappings and the log as well the statistic process is controlled by the *GlobalControl* object. A dynamic UML diagram is used to explain the behaviour of the system after the initialisation process. For this purpose, I use a sequence diagram. For a better overview, I split the interaction between *GlobalControl* object and the other mediator objects in two parts. The first diagram shows a global view of a mediator step. In the second diagram the mapping unit is illustrated. Additionally, a picture of the small graphical user interface that is used for the user control of the step size is shown.

First of all, we concentrate on the sequence diagram in Figure 31. At the beginning of the diagram the object of type *GlobalControl* is active and all simulation subsystems are started as well as connected to the mediator. In the sequence diagram two alternative sections are shown. Which section is used depends on the users' input. The user can define how many mediator steps shall be performed in a row. Also the user can terminate the entire system. For an easy control of the mediator during runtime a small graphical user interface allows the user to enter a mediator step number, start the execution of the next mediator step(s) or to terminate the simulation. Figure 32 shows an illustration of the GUI. This Java swing component includes a text field, where the user can enter the mediator steps, and two buttons ('Step' and 'Exit'). Also, a small control element is shown on the

[27] http://www.angelikalanger.com/Articles/JavaSpektrum/03.HashCode/03.HashCode.html; accessed 28.07.2007

right side of the window. A red dot indicates that currently a mediator step is running. A green dot shows that the mediator is waiting in an idle state and can perform the next mediator step when designated. If the user presses the button with the label ‘Step’ the mediator performs the method calls which are shown in the first section of the sequence diagram. If the user presses the button ‘Exit’ the mediator system performs the commands that are illustrated in the second section of the sequence diagram.

The first section includes a loop. The user defines the number of iterations of this subsection with the input in the control GUI. For instance, the user enters four in the user interface the mediator performs four mediator steps in a row, which is equal to four iterations of the subsection in the sequence diagram. In the following a single iteration is explained in detail.

First of all, the *GlobalControl* object calls all functional and structural subsystems to update the internal data structure. The object uses for the interaction with the simulation subsystems the handle objects that are stored in the *HandleList*.

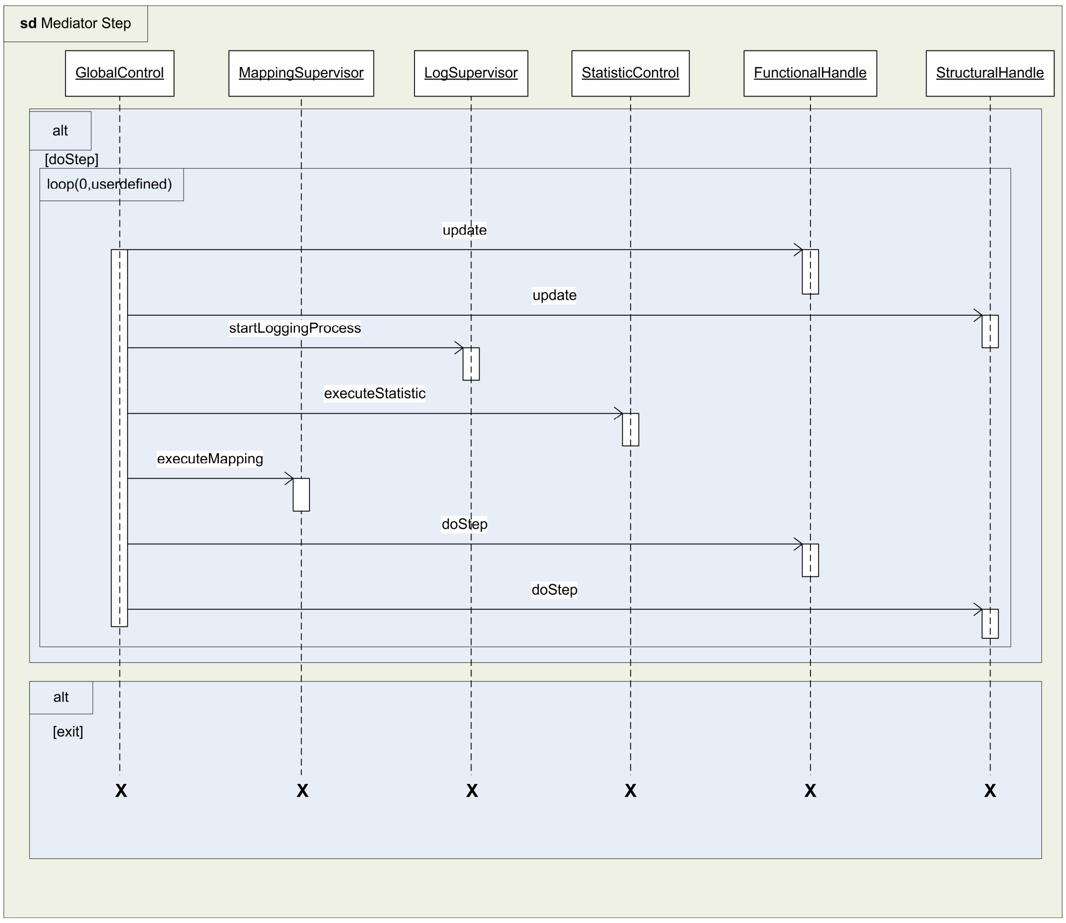


Figure 31: Sequence diagram illustrate the method calls during a mediator step.

Afterwards, the *GlobalControl* instance calls the method *startLoggingProcess* from the class *LogSupervisior*. All users’ desired simulation attributes are identified in the simulation subsystems and stored in a comma separated file. In the following iterations the log unit uses the same file for the consistent storage of the log elements and adds the new elements to the end of this file. Following the lifeline of *GlobalControl* the next method call is *executeStatistic*. Similar to the log process, all desired statistics are calculated, and the statistic unit store the results by adding them to a CSV file. After the log and statistic process, the method *executeMapping* in *MappingSupervisor* is called and all mappings are executed. A detailed description of the mapping process follows later. Finally, the *doStep* method in each handle object is called and all simulation subsystems perform a system step, if the user defined condition is true. The user can specify during the initialisation process of the mediator in which ratio between subsystem simulation system and mediator step a subsystem simulation step is performed. The implementation of this condition is quite simple. The system saves for each simulation subsystem an integer value and the current step value of the mediator. For instance, a simulation subsystem should perform one step every four mediator steps. In this case the mediator saves for this subsystem the step size four. The method *doStep* uses the operator *modulo* to calculate the current mediator step value *modulo* the step size of the subsystem. If the result is zero the subsystem will perform a step, otherwise nothing will happen.

In the second alternative section of the sequence diagram all lifelines end, which is equivalent to the termination of the mediator objects. Consequently, the entire simulation process will be terminated and the system is closed when the user presses the 'Exit' button.

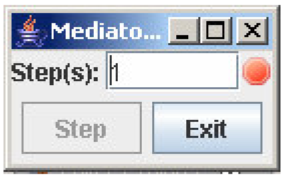


Figure 32: GUI for the mediator steps size control.

The sequence diagram in Figure 33 illustrates the system dynamic during a mediator step and after the calling of the method *executeMapping*. For a better orientation in the application flow, the sequence diagram repeats the mediator step loop section with the method call *executeMapping*. All previous interactions are indicated by the three dots in the graphic. After the method call, the instance of *MappingSupervisor* executes first all pre-mappings and afterwards all main-mappings.

To execute all pre-mappings the *MappingSupervisor* instance sends a request to *MappingList,* to get a reference to all *GlobalMapping* objects in the pre-mapping list. *MappingList* uses a type specific *ObjectArryList* out of the *Fastutil* collection framework to save the *GlobalMapping* objects. The reason to use an *ObjectArrayList* instead of a *HashMap* implementation at this point is that in each mediator step all mapping have to be executed. Therefore, a total iteration over all elements is required and not a specific element has to be called. Each *GlobalMapping* object contains a *MappingControl* object. *MappingControl* is responsible for searching the corresponding *SimulationObject* and *StructuralObject* instances for the mapping in the internal data structure. Each *MappingControl* object saves for an already existing mapping a reference to the required *SimulationObject* and *StructuralObject* instances. The next method call in the sequence diagram, *updateExistingMappings,* uses these references to test the already existing mappings. For instance, a simulation subsystem deletes the corresponding object in its simulation; the reference object in the mediator is deleted as well. Therefore, a mapping which uses this object is not valid anymore and the method *updateExisting-Mappings* deletes the internal existing mapping that is saved in a *Dependence* object. In the other way round, a simulation subsystem creates a new object. This object is used in a mapping definition in *GlobalMapping*. The method *creat-eNewMapping* queries all simulation subsystem objects, which are not already mapped of this type, creates a new mapping and saves it in a new *Dependence* object. In this way, it is guaranteed that not valid mappings are deleted automatically and a new mapping is created automatically when possible. Here it is important to mention that there is a difference between functional and structural subsystems. A new mapping can only be created, if an instance of a source and destination object, which is defined in *GlobalMapping,* exists in the internal data structure. If one of these objects is missing, the mediator creates autonomously a new instance of the defined simulation subsystem. Therefore, it is possible to simulate a system which creates new objects during the runtime. In one case the mediator is not allowed to create a new instance of a simulation subsystem. If the destination mapping object is from type *StructuralObject* a creation of a new instance of a simulation subsystem can adulterate the whole simulation. For example, the structural simulation subsystem simulates a tree, which is composed of small sub-elements. Each of these sub-elements is included in a mapping to update a hormone concentration in the structural simulation. A potential mapping can be hormone concentration in a functional simulation is mapped to the corresponding attribute value in the structural simulation. Now, it is possible that four instances of the functional subsystem exist, but only three sub-elements in the tree. The mediator detects that the difference between source and destination objects is one, and he creates a new instance of a structural simulation. This is obviously false. Now two trees exist; one tree with three sub-elements and one tree with only one sub-element instead of one tree with four sub-elements. Therefore, an autonomous creation of instances, when the destination object of the mapping is from type *StructuralObject,* is forbidden. Only the structural simulation itself can create a new instance of an internal object. In the example above, the mediator creates only three mappings and if the structural simulation creates a new tree sub-element on its own, the fourth mapping is created automatically from the mediator.

After the creation of new mappings and the update of existing mappings, the instance of *MappingSupervisor* can call the method *executeMapping* for each *GlobalMapping* object. Depending on the defined *Dependence* object the mapping is executed. All these updates produce new instances of *UpdateObject,* or if the mapping already exists longer than one mediator step, an existing instance is updated. The reason for using another abstraction layer instead of writing the new values directly in the corresponding simulation subsystems is that most of the subsystems allow a change in their internal data structure only in special system states. Because of performance reasons, it would be out of the question to wait after each single mapping of this special subsystem state to update the values. Therefore, all mapping results are stored in an instance of *UpdateObject,* and after all pre- or main-mappings all values update at once. This update is performed in the methods *updateObjectValues* in the *Handle* objects.

The next two method calls are optional. Some simulation subsystems require an internal step after an update of the data structure. Otherwise, during the next update the previous one is lost. For each simulation subsystem, the user uses a *Boolean* attribute to declare the necessity of a system step after an update. If this attribute value is *true* the *MappingSupervisor* instance calls *doStep* in the corresponding *Handle* instance.

Now exactly the same procedure is performed for the main-mapping. As described in the last chapter, a differentiation between pre- and main-mapping is essential to realise all required mappings between simulation subsystems. Specifically, a pre- and a main mapping are required in the following example. In a structural simulation a transport from one object to another should be realised in one mediator step, and the transport process should be defined via a mathematical formalism (an example can be a fluid dynamic transport). The user defines two mappings in this case. The first mapping is defined from the source object in the structural simulation to another simulation subsystem, which calculates for instance the fluid dynamic. This mapping is called in the mediator pre-mapping. The main-mapping starts at the result attribute from the calculation unit and links to the destination attribute in the structure simulation. Without the differentiation between the two mapping classes such a mapping would require two mediator steps.

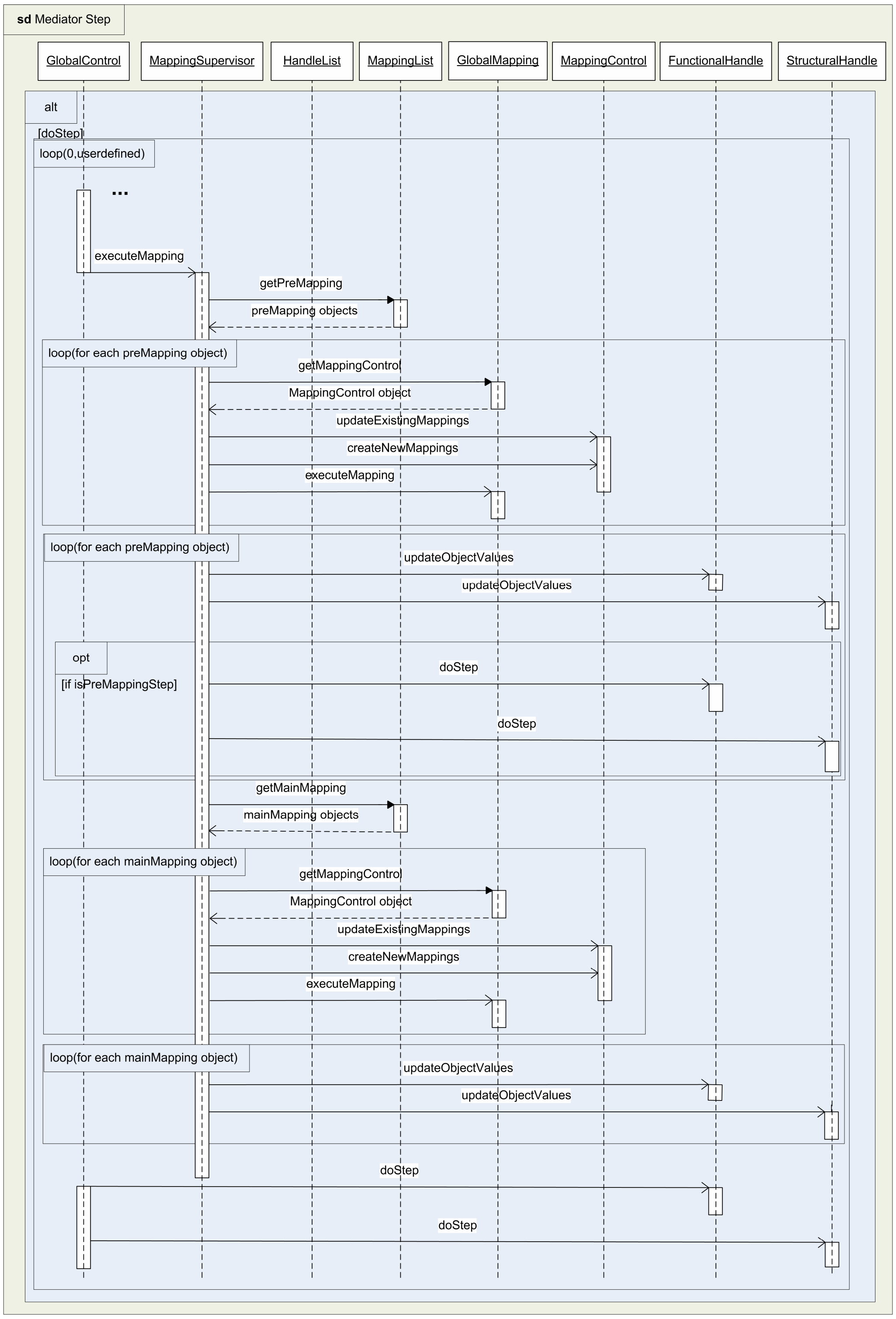


Figure 33: UML sequence diagram illustrate the system dynamic during the mappings.

5.3. GroIMP interface

To connect GroIMP with the mediator a new interface is implemented. In principle the system was not designed to share its functionality with external applications. In the first place, GroIMP is designed as 3D-modelling-plattform using the potential of growth grammar. A new modelling language, called XL is defined for GroIMP. The whole system is implemented in Java and also the XL language is an extension of the Java programming language. Using a normal wrapper or adapter class approach to create an interface to GroIMP is not feasible by reason of the used XL language. The GroIMP framework includes a compiler, which transfers the user defined XL model description into a Java compliant byte code before the simulation starts. Beside the predefined classes, which relate to the basic L-systems commands and basic 3D shapes, the user can define new classes. To define a new simulation an integrated editor in the GroIMP front-end is used. Accordingly, to provide an interface to the existing classes in the system is insufficient. Together with Tully Yates I decided to use a call-back method approach and to use the Java Reflection API [28] to create an interface for the GroIMP system. Additionally, GroIMP does not allow changing internal values at any time. To simulate the quasi parallel execution of the RGG commands a complex thread handling is required. Only in a save system state the internal graph data structure can be read or modified.

The design of the GroIMP interface is shown in Figure 34. This approach requires in each new simulation model in GroIMP the insertion of a Java class definition in the model. After the compile process the object instance of this class allows a communication with GroIMP. The class in GroIMP must implement the interface *Visualisation*. In the class diagram this class is called *GroIMPVis*. In the first place, after the compile process the model creates an object of *GroIMPVis* and calls the static method *register* in the static class *GroIMPControlStart*. With this method call a reference to the *GroIMPVis* instance is sent to the *GroIMPControlStart* class, which creates a new *GroIMPControl* instance. Now, each instance of *GroIMPControl* saves a reference to an object inside the *GroIMP* simulation. In this way, GroIMPContol can call the methods *run*, step, *fetchGraph* and *graphUpdate* in the *GroIMPVis* class. To get the simulation attribute values out of the simulation *GroIMPControl* calls the method *fetchGraph* in *GroIMPVis*. Internal thread handling processes and synchronisation processes are required to access the graph structure in *GroIMP*. This access is only permitted in a class which implement the *LockProtectedRunnable* interface. An instance of *GraphReader`*, which implements this interface, queries the graph and creates for each node in the graph a new object from type *OrgNod*. Afterwards, this object is saved in a collection in *GlobalControl*. Each *OrgNode* object saves a set of child nodes and its parent node as *OrgNode* instances. Furthermore, the object saves an *OrgAttribute* instance, which includes for each simulation attribute in the node an object of type *Attribute* to save the name, type and value of the simulation attribute. This *OrgNode* object is transferred in *GlobalControl* to an object of type *StructuralObject*. To sum up, updating the internal data structure in the mediator with

[28] Detailed information about the Java Reflection API can be found at the webpage: http://java.sun.com/j2se/1.5.0/docs/guide/reflection/index.html; accessed 01.08.2007. Java Reflection API enables Java to discover objects during runtime to collect information about these objects. Information can be attribute names and values, method names and information about the object constructor.

the values out of the *GroIMP* simulation requires the following steps. Firstly, *GroIMPControl* calls *fetchGraph* in *GroIMPVis*. At this point, it is necessary to stop the execution of the mediator thread until the GroIMP thread, for the graph query, terminates. In the next step, *GroIMPVis* calls the *run* method of *GraphReader* which creates the *OrgNode* objects. The last command in the *run* method is to send a notification to the mediator thread. Now, the mediator can transfer the *OrgNode* objects into *StructuralObject* instances.

To update the simulation objects in *GroIMP* after a mediator step, *GroIMPControl* calls the method *graphUpdate* in *GroIMPVis*. Similar to the method call of *fetchGraph*, the mediator thread has to wait until the *GroIMP* update process is terminated. Beforehand, the *GroIMPControl* class changes the *OrgNode* objects regarding to the mediator changes and sets a *Boolean* flag *true*. Now, the method *graphUpdate* updates the internal data structure in *GroIMP* using all *OrgNode* objects with a set flag. The update process is performed in the *run* method of *GraphWriter*. To guarantee a thread save update, this method implements the *LockProtectedRunnable* interface. After the update process this method sends a notification to continue the mediator execution process (main thread).

The Java Reflection API is used in this process in the class *OrgFields*. Using reflection to identify *GroIMP* objects and the attributes' names and values is required at this point by reason of the *GroIMP* modelling structure. In each *GroIMP* model, predefined and user defined classes are used to describe the model behaviour. Therefore, it is only possible to identify the used objects in the simulations during runtime. The performance disadvantages of the reflection approach afford to use the entire modelling power of *GroIMP*. A restriction, which allows only predefined classes for a model, restricts the modelling power significant. Also, a strict exception handling is required to catch all possible exceptions, which can be occur during the usage of the reflection API. A combination of exception handling and predefined conditions guarantee a save usage of the reflection classes and methods.

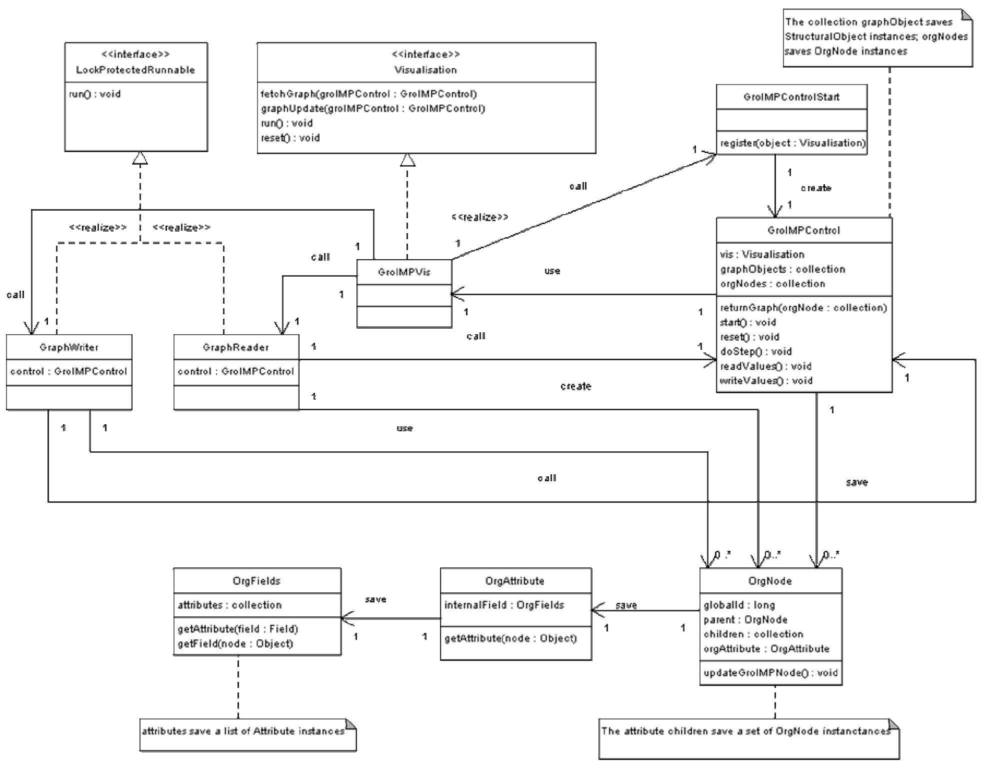


Figure 34: GroIMP interface design.

6. Results

The introduced mediator approach allows different independent simulation tools to interact with each other. The combination of structural and functional simulation tools allow the scientist to simulate biological systems in a more realistic way. This chapter includes a short recapitulation of the mayor system functions and illustrates the general applicability with the help of two examples.

The mediator software system is used for the combination of existing simulation tools and to control those, to perform a combined simulation. Figure 35 shows the necessary steps to perform a simulation with the mediator. It should help to clarify the interaction between mediator and existing simulation tools. The illustration is split in a vertical and a horizontal section. On the left side in the vertical segmentation all functional simulations tools are combined. In the middle, the control section composed of the user and the mediator framework is placed. Right aside the control components all structural components (simulation mediator and abstraction layer) are combined. The horizontal alignment includes bottom-up: the initialisation process, the user control and the framework with all connected simulation tools itself.

In the first place the user has to select the simulation tools. Afterwards, he creates in each tool a model with the help of the tool internal editors. For the creation process, the scientists use different sources (data integration tools, databases, literature or personal knowledge). In the figure this process is numbered with number one. Next, the user has to define the relationships (mappings) between the beforehand created models and to initialise the mediator framework (see label two in the illustration). Now the user controls the entire simulation process. He decides when the next simulation step is performed and when the simulation will be terminated. In each simulation step the mediator communicates with all structural simulation systems and all functional simulation systems which are necessary for the current simulation process. The figure highlights the bidirectional connection between the simulation tools and the mediator (number three – eight).

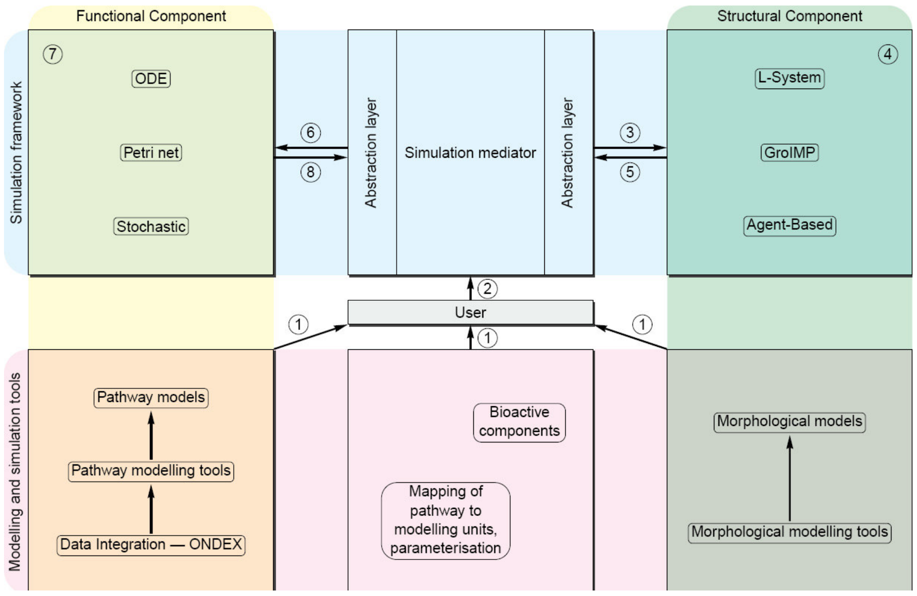


Figure 35: Mediator framework (illustration by Tully Yates).

Regarding to the clear interface structure and the various abstraction layers, the system is easy to maintain and further extensions are easy to realise. Different interfaces allow adding nearly all existing simulation tools to the mediator. Also, the subsystem handle design does not restrict the number of simulation subsystems which can be connected simultaneously to the mediator. Basic mapping types between simulation systems are predefined. For each complex mapping the mediator assists the user with a clear interface design and examples. Log and statistic files provide the user an insight into all simulation values after the simulation to evaluate the results.

During the implementation, a score of test simulation implementation has shown good results for small and medium sized simulations. The mediator software is able to control the subsystems in the right way, and also the communication between the simulation subsystems and the control software is fast enough for the simulation process. All required mapping combinations between the simulation tools are assisted. Different tests with large simulation models in the simulation tools adduced partly unsatisfactory results. One simulation step and the update of the subsystems require a too long time. The mediator has only a small influence to this long execution time. In the first place, the existing simulation tools and the interaction with these tools are responsible for the delay times. Additionally, the implemented examples show the advantages of the usage of different simulation tools. Each simulation tool can concentrate on it own model, and with the help of the mediator, each required additional functionality is provided by other tools. Accordingly, the several models are easier and the performance of each simulation can be improved in this way.

The following two examples show the interaction between functional and structural simulation tools on the basis of two very simple biological systems. The systems raise no claim of completeness or true to original. They should only illustrate the applicability of the mediator framework. In addition, the examples show the flexibility of the framework. Not only mappings between structural and functional simulation tools or vice versa are allowed. A mapping between a functional and another functional tool as well as between a structural and another structural tool is feasible.

6.1. Example: ABC model

This model uses the mediator framework to simulate the so called ABC model of flower development. To define and visualize the structure of the flower the modelling tool GroIMP is used. On the other hand, for the simulation of the gene concentrations in the flower, Jarnac is used. Consequently, this simulation is an example for a mapping between a functional simulation tool (Jarnac) and a structural simulation tool (GroIMP).

The ABC model of flower morphology indicates that different classes of transcription factors, in different parts of the flower, are responsible for the specification of the different flower organ cells. Each class of genes and their combination is responsible for the development of the flower elements. This model illustrates the flower development and different mutations in a clear and easy way. Figure 36 shows an illustration of a flower with all flower organs: carpel, stamen, petal and sepal. In the third part of the illustration, the symmetry of the different flower organs is visible. In the next figure (Figure 37), the dependences between classes of genes and produced flower organs are shown. Genes in class *A* produce the sepals. A combination of *A* and *B* specifies the petals, *B* and *C* the stamens and an occurrence of genes only of class *C* the carpels. More details about the ABC model of flower development can be found in [Alb02].

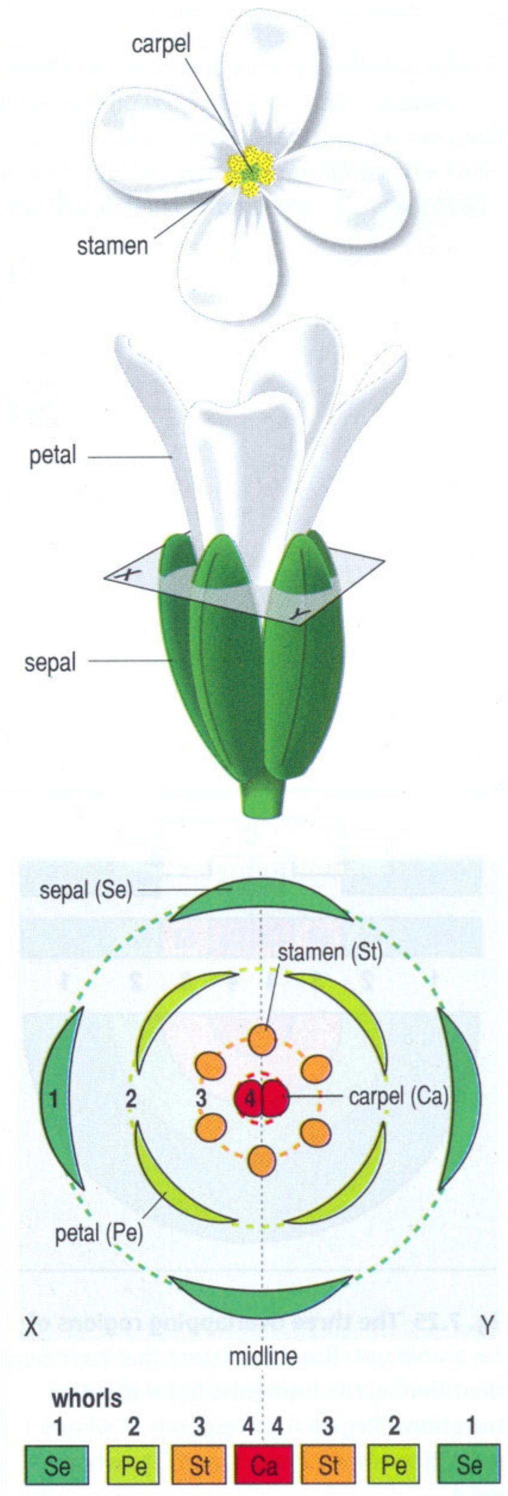


Figure 36: ABC model of flower development (from [Wol02]).

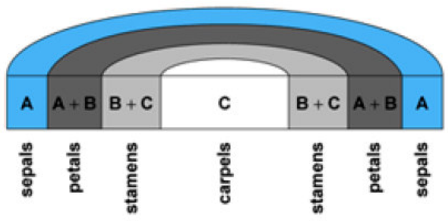


Figure 37: Classes of genes are responsible for the development of the different flower organs (from [Alb02]).

To highlight the advantages of the mediator approach, the model is based on an already existing example in GroIMP. The GroIMP package includes an ABC model example, defined in the XL modelling language. This model uses a regulatory network to simulate the transcription factors and a visualisation of the corresponding flower organs. The model of the regulatory network is presented in detail in [Kel01]. To get an impression of this model, Tully Yates and I created a continuous Petri Net model with inhibitor arcs. Figure 38 shows an abstract view of this model, without tokens and without transition functions. All green places save the gene concentration and the blue places save the concentration after the transcription process. For the simulation the places *a*, *b* and *c* are interesting. They store the concentrations, which are responsible for the development of the flower organs (see Figure 37). Attend, the Petri Net uses small letters and the transcription factors are labelled in Figure 37 with capital letters.

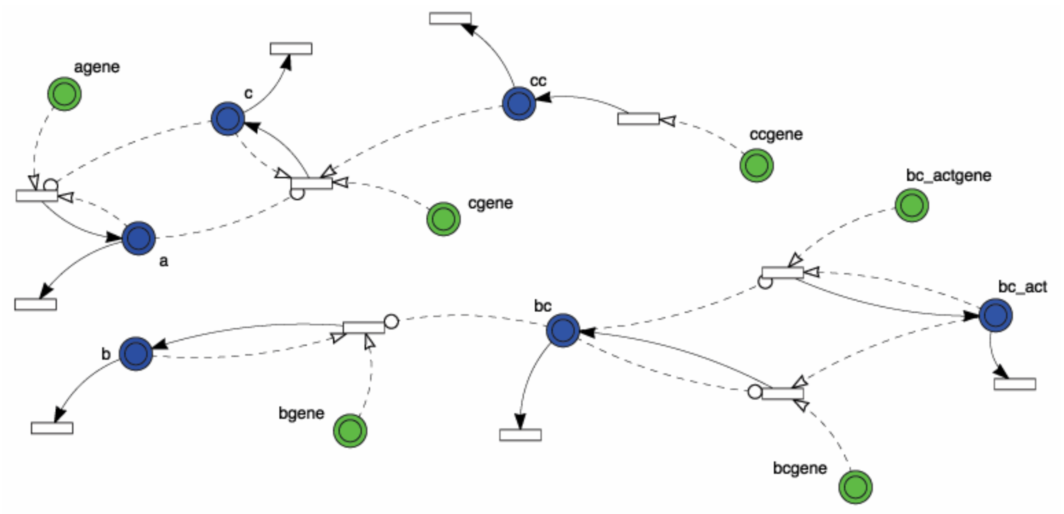


Figure 38: Petri Net model for the ABC model (by Tully Yates, Rothamsted Research, United Kingdome)

To reuse the existing model in GroIMP for a later comparison, I separated the definition of the regulatory network and the part of the model which is responsible for the 3D structure (visualisation). Consequently, the new GroIMP model is only responsible for the visualisation of the simulation. On the other hand, the introduced regulatory network is translated in a Jarnac compatible notation, and due to some mathematical solving limitations of Jarnac, simplified in some parts. After the separation of the model into two models in different simulation tools, the mediator framework is used to combine these two simulations again. To map the current concentration of transcription factor *A*, *B* and *C* (in Jarnac) to the structural simulation (GroIMP) three independent mappings are created. GroIMP requires (for the simulation) only the current concentration and decides with simple if-statements which flower organ should be created. Therefore, all mappings are from type update dependence and will be updated in each mediator step. The entire calculation of the transcription factors are performed in one simulation instance and only a data flow in one direction (functional simulation to structural simulation) is necessary. Because of this, all mappings are defined as main-mappings. To illustrate the log functionality of the mediator, each transcription factor value is saved during the simulation in each time step.

Three mappings are defined in the mediator. The functional simulation includes three attributes *A*, *B* and *C*. On the other side, in the GroIMP model three attributes *a*, *b* and *c* exists. Figure 39 shows the dependences between the functional simulation attributes and the structural simulation attributes.

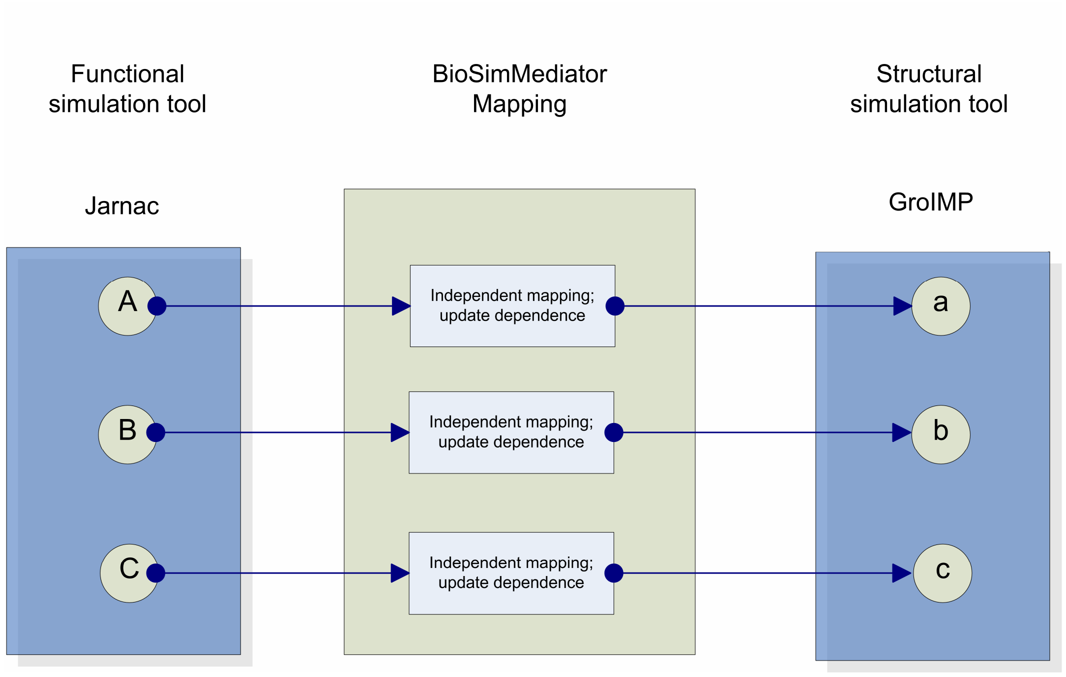

Functional simulation tool
BioSimMediator Mapping
Structural simulation tool
Jarnac
GroIMP
A
B
C
Independent mapping; update dependence
Independent mapping; update dependence
Independent mapping; update dependence
a
b
c

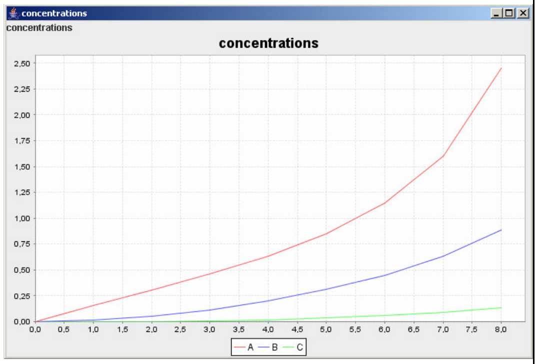
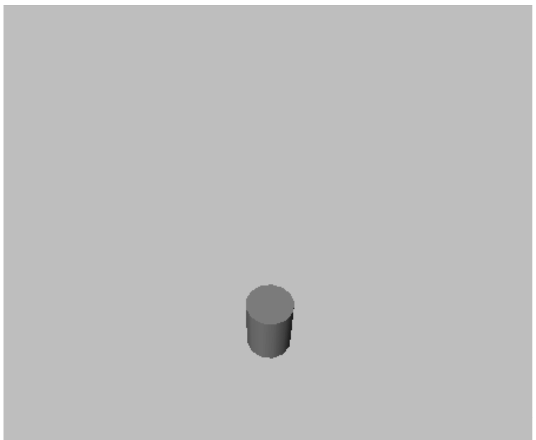
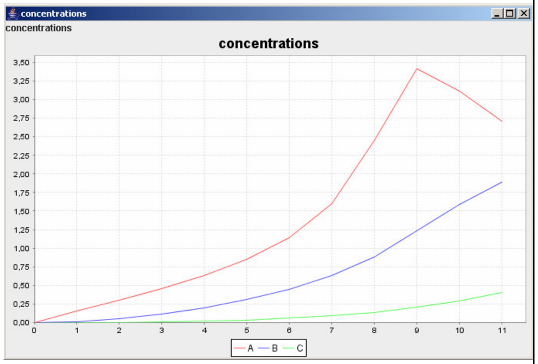
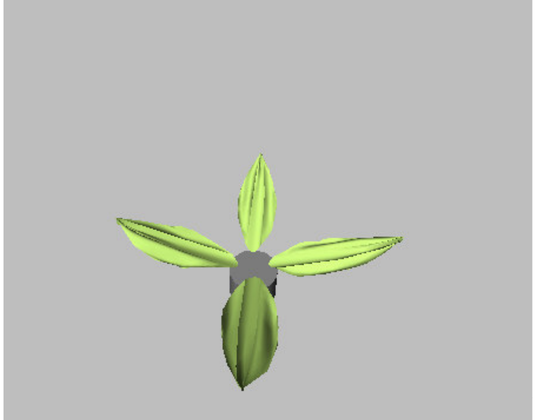
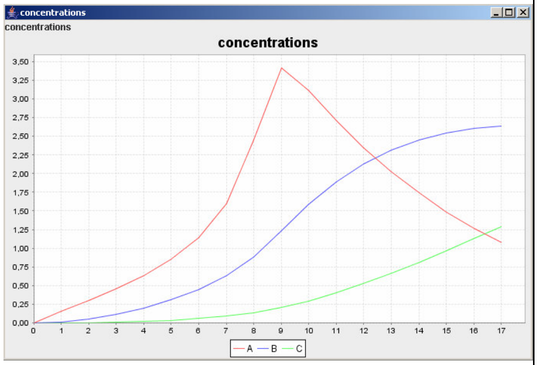
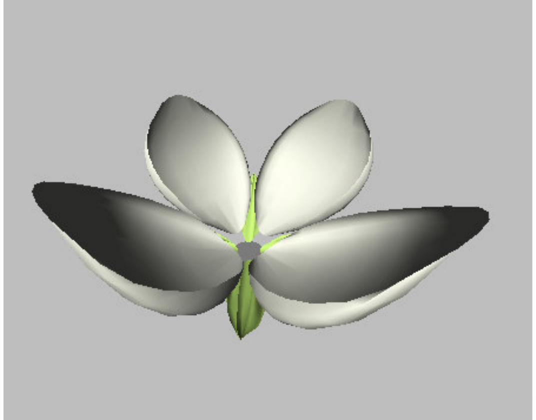
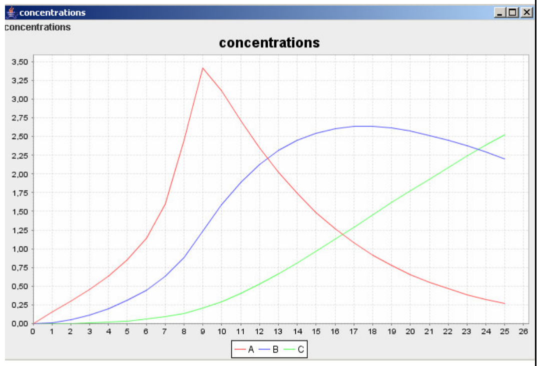
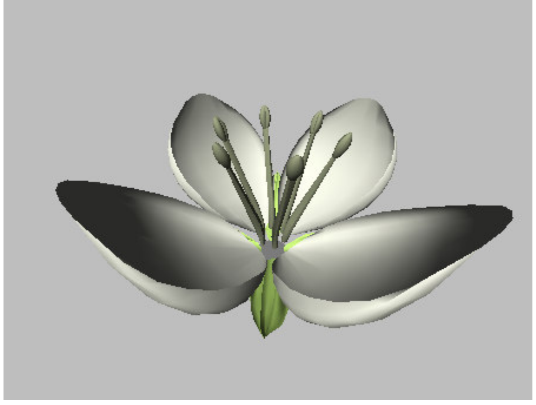
Figure 39: Mapping of the ABC model.

After each mediator step the simulation attributes *a*, *b* and *c* saves the current transcription factor concentration of the biochemical simulation in the GroIMP model. The structural simulation uses these values to decide which 3D structures have to be created. As mentioned before, the creation of the different flower organs depend on these values (concentrations). The following dependences are used in the example:

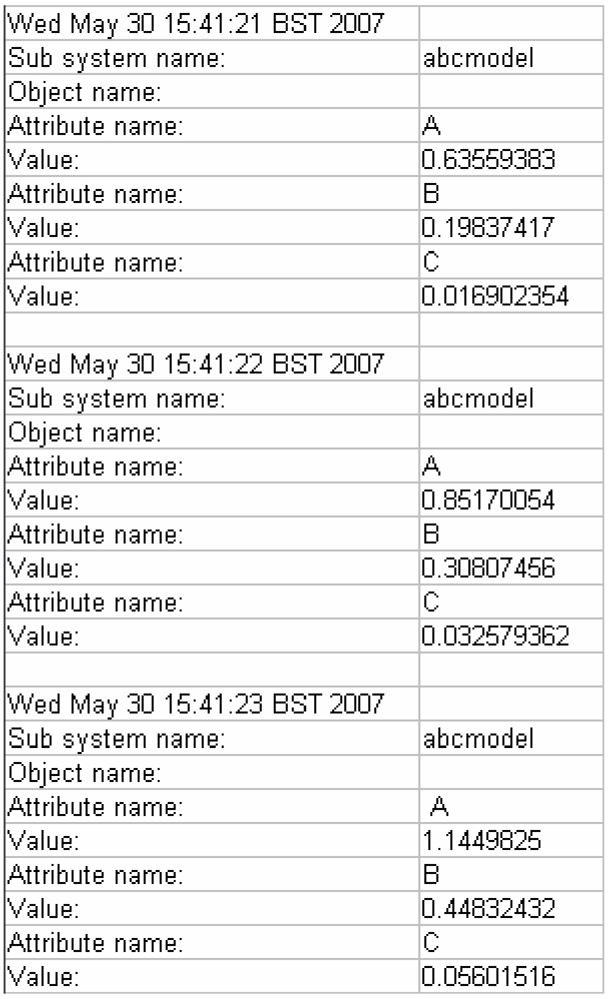
1. *if* c > 2.4 *then* terminate simulation *else 2.*
2. *if* b> 1.7 *then* (*if* c>a create stamen *else* petal) *else 3.*
3. *if* a> 1.8 *then* (*if* c>1.8 create shoot *else* sepal) *else 4.*
4. *if* c > 1.8 create carpel

Table 5 illustrates the results of the simulation at different time points.

Table 5: Simulation results: ABC model of flower development.

Gene classes concentration	3D structure
	
	
	
	

To explain the log unit results briefly, Figure 40 shows the log values in three mediator steps. The log file saves for each mediator step the current system time. In the next line, the name of the used model is presented. Here, the file contains the name ‘*abcmodel’*. The model does not differ between different objects in the simulation. Therefore, the line object name is empty. The next six lines include the names of the attributes and the corresponding attribute values. A blank line separates each mediator step blocks in the log file. For instance, in the first mediator step the attribute *A* contains the value 0.63559383, *B* contains 0.198,7417 and *C* contains 0.016902354. If a log file contains more than one simulation subsystem, each block is sorted in an ascending alphabetical order using the subsystem names.



Wed May 30 15:41:21 BST 2007	
Sub system name:	abcmodel
Object name:	
Attribute name:	A
Value:	0.63559383
Attribute name:	B
Value:	0.19837417
Attribute name:	C
Value:	0.016902354
Wed May 30 15:41:22 BST 2007	
Sub system name:	abcmodel
Object name:	
Attribute name:	A
Value:	0.85170054
Attribute name:	B
Value:	0.30807456
Attribute name:	C
Value:	0.032579362
Wed May 30 15:41:23 BST 2007	
Sub system name:	abcmodel
Object name:	
Attribute name:	A
Value:	1.1449825
Attribute name:	B
Value:	0.44832432
Attribute name:	C
Value:	0.05601516

Figure 40: Log file of the ABC model example.

This example shows how to use an existing model for a simulation with the mediator. In comparison to the original model in GroIMP a differentiation of the functional and structural part of that model is used in the presented example. A direct comparison between both model results is not feasible, because of the mathematical solving problems of the used functional simulation tool. Therefore, an adaptation of both models (structural and functional) was necessary, which results in a slightly different simulation course. In this example, the major advantage of splitting a simulation is not obviously. Scientists can work together in different tools on one simulation, following only some basic interface agreements and naming convention. Hence, each scientist can work on his area of expertise, and an administrator can map these single simulations together to an entire simulation. Furthermore, the models can be implemented or designed in a model specific formalism. The author of the ABC model example in GroIMP uses the XL language to model the regulatory network. Using a Petri Net approach for the biochemical model should be much easier and familiar for the scientist instead of transferring the entire process to object oriented formalism.

In conclusion, this example shows the applicability of the mediator framework to combine structural and functional simulations to an entire simulation. The next example illustrates the combination of two structural simulation tools. In addition, a dependent mapping is used and the statistical unit of the mediator is presented.

6.2. *Example: Root development*

Creating a realistic simulation of a root is quite difficult and complex. The root structure differs from plant to plant. Some roots grow deep into the soil and some grow near the surface to the sides. Also, the branching behaviour of the root depends on the species and the soil. For instance, different densities or water concentration in the soil produces different root structures. In literature a multitude of experiments and descriptions of the root development process can be found (see [Gas01]).

To focus on the interaction between different simulation tools, the presented root example is quite simple. The branching factors are defined in a GroIMP model and the interaction with the soil is implemented in an agent based model in Mason. The agent model simulates the soil density and the gravity. Hence, the agent simulation is also responsible for the structure of the root. The default grow direction of the root is downwards. If the density in this direction is too high, the root will have to change its direction and to grow sideward until it can grow again downwards. This default growing direction is specified in a gravity value in the agent simulation. A high gravity value affects the root to grow downwards whenever possible. A smaller value results in a rather sideward growing root. First of all, I will explain both models separately and afterwards the interaction of both tools with the necessary mappings.

The multi agent simulation tool Mason uses different types of matrixes to save the agent and the environment. To simulate the soil, I use a three-dimensional matrix, which can save 64.000 integer values. All these fields include an integer value between zero and six and represent the density of the soil. Whereas, the integer value zero signifies a negligible small density in the soil. Furthermore, a density value of six indicates an area in the soil where the root is not able to go through. All integer values between zero and six encode the soil density in an ascendant order. Each of these fields are visualised by a cube with the side length of one. Regarding to this, the simulation simulates a soil area of length, weight and height of forty cubes. During the initialisation process of the simulation all fields get randomly a density value. In addition, the agent simulation includes a double variable to save the standard gravity. The well known standard gravity of the earth is about 9.81 m/s^2 and is used as the default value for the simulation. A higher value indicates a stronger gravity and a lower value a lesser gravity. Furthermore, the agent simulation includes two types of agents: *Root* and *RootSegment*. Each *RootSegment* can save one or more *Root* agents. In principle this kind of agent is used to combine various *Root* agents into a category. Each *Root* agent in this category is able to interact with its environment, but only the last *Root* agent, who is added to the category, is able to ‘grow’. Interact with the environment can mean, a transport of water, a nitrogen uptake or a density ‘test’. Each *RootSegment* represents in the simulation a part of the root which has not branched so far.

To sum up, all agents of type *Root* are responsible for the growing process and for the interaction with the environment. The agents of type *RootSegement* are only used to combine various *Root* agents, and to control which agent is ‘active’ (can grow and interact with the environment) and which one are ‘passive’ (can only interact with the environment). Figure 41 shows a part of the simulation area in 2D. Grew rectangle illustrates the soil with different density values (light grew indicates a low density and a dark grew a high density). In addition, two *Root-Segment* categories are visualised. The first, are bounded by the red line, includes four *Root* agents and the second blue bordered one includes three agents.

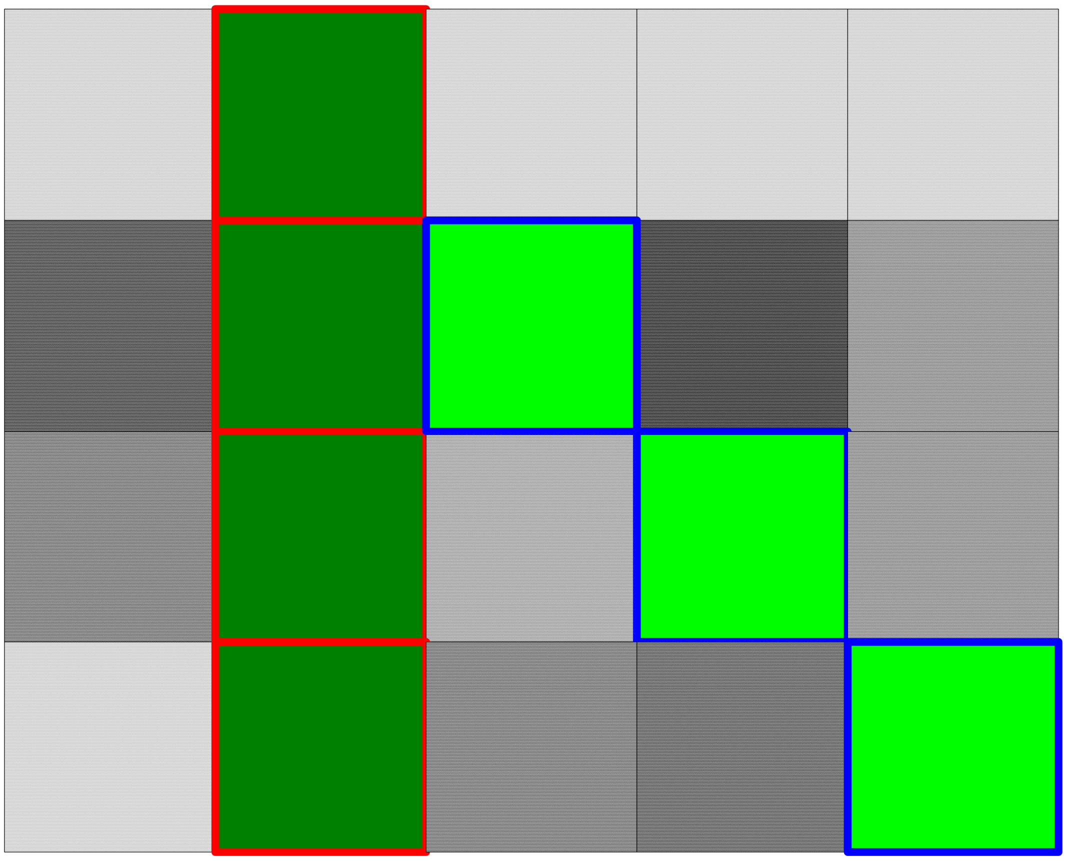


Figure 41: Root and RootSegment agents and the density of the soil in 2D.

During a simulation step each *RootSegment* tries to grow. Nevertheless, only the active *Root* agents in the *RootSegment* are allowed to grow. Each of them evaluates its environment and decides afterwards in which direction it grows. Of course, it is not possible to grow backwards. Therefore, each agent has to test twenty three cubes (in 3D and five in 2D). The order in which the agent tests the cubes depends on the gravity constant value. Figure 42 illustrates this process in a 2D visualisation. With the default gravity value, the active agent (with the red border) tests in the first place the rectangle directly downwards (number *one*). If the density in this field is lesser than the maximum density, a new *Root* agent will be created by the system and placed to this position. Also, this new agent is added to the *RootSegment* agent and becomes the new active *Root* agent in this category. If the density in this area is to high too grow in this direction, the active *Root* agent will test the fields left and right downwards (number *two*). If one of these fields has a density lower than the maximum density, the root will grow in this direction. In the case of an equal density in multiple sections, the simulation decides randomly in which direction to grow. If the growing process downwards is impossible, the active agent will try to grow sideward (number *three*). A maximum density in each direction results in a stagnation of the growing process of the *RootSegment*.

A future extension of the model can also include water or mineral concentration in the soil, which influence the growing process as well.

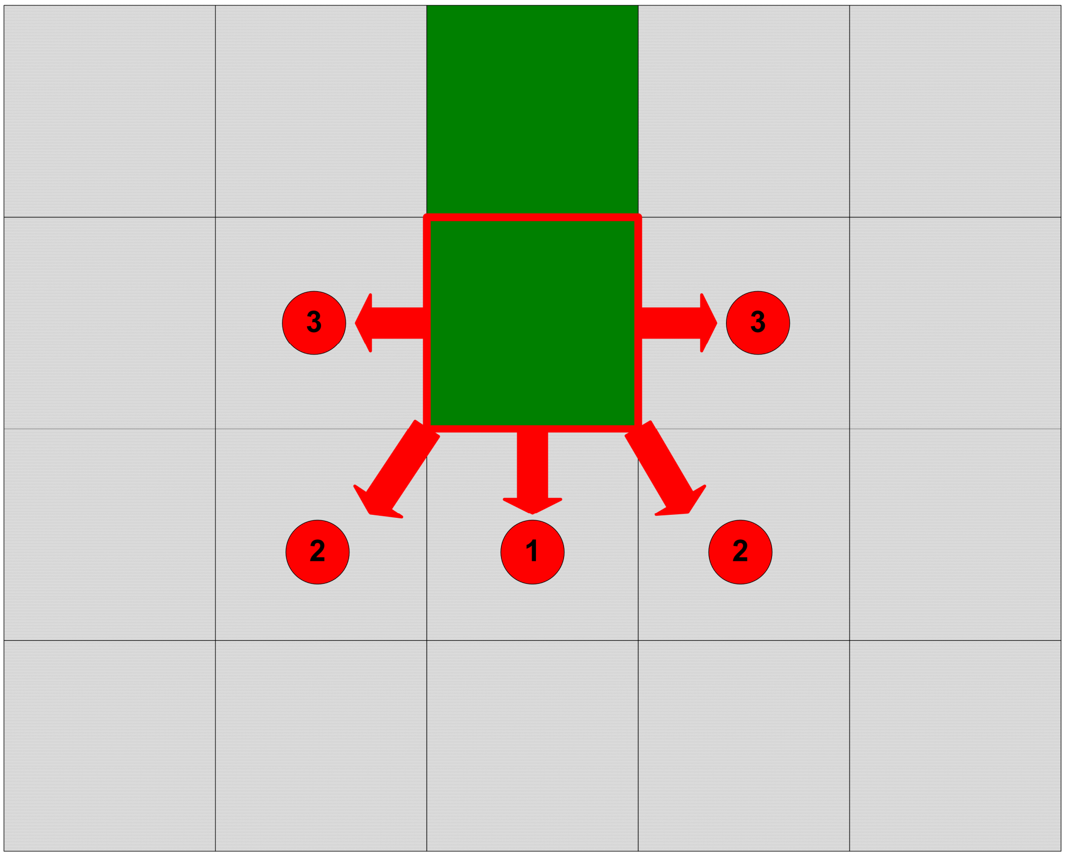


Figure 42: Agent root growing direction.

The agent simulation can only control the growing process and the interaction of the root with the environment. As mentioned before, different kinds of plants have a characteristic branching structure of the root. This branching structure is encoded in the GroIMP model. In this way, the agent model decides in which direction the root can grow and the GroIMP model decides at which place the root has to branch. With the help of the 3D matrix in the agent simulation it is easy to access each position in the grid directly. If the GroIMP simulation sends the command to branch at a special position, the existing *RootSegment* at this point is split into two *RootSegment* agents and a new side branch starts growing at this place.

The GroIMP model uses basic L-System elements to create the root structure. In this basic example, the model uses the length of each root branch to decide at which time (position) a new branch has to be created. The branching angle and the position are predefined and depend also on the agent simulation. With the help of these values, the user can change easily the structure of the root and simulate different kinds of root structures.

To combine both models to an entire root simulation, six mapping definitions in the mediator are necessary. In contrast to the first example, all these mappings are from type *Dependent* mapping. All these mappings are defined between one *RootSegment* agent in the agent simulation and one root segment (simple L-System element) in the GroIMP simulation. During the complete simulation process, the mediator has to ensure that always the same *RootSegment* agent and GroIMP root segment are mapped together. Each instance in the simulation includes a unique identifier and allows the mediator the identification. After the initialisation process of the simulation, the agent simulation contains one *RootSegment* agent with one active *Root* agent. The GroIMP simulation starts with one root element. Therefore, in the first mediator steps six mappings between those both elements exists. During branching, a new *RootSegment* agent is created and also a new root segment in the GroIMP simulation occurs. For each of these new elements, the mediator creates six new mapping instances and saves the element identifiers. A mapping from type *Dependent* is mandatory to ensure that the mappings are always exists between the same *RootSegment* agent and the root segment in the GroIMP simulation. The six mappings are split into five pre-mappings and one main-mapping. In Figure 43 all mappings for the simulation are shown.

All mappings inside the blue rectangle are pre-mappings and the mapping inside the red rectangle is a main-mapping. The execution order during the mapping process is equal to the order in the illustration. In the first place, the pre-mapping between the attribute *branch-distance* in the GroIMP simulation and the attribute *branch-distance* in the agent simulation is from type update dependence. The attribute values are used to specify at which point a new side branch can grow. The GroIMP simulation defines the value in each simulation step for each root segment, and with the help of the mediator the value is transferred to the corresponding *RootSegment* in the agent simulation. The next three mappings are responsible for the correct orientation in the 3D visualisation in GroIMP. The orientation of the different root segments is defined in the agent simulation. For instance, the root has to grow sideward to avoid a high soil density, and therefore, the orientation of the entire *RootSegment* changes. To transfer the new orientation from the agent simulation to the coordination system in GroIMP, each *RootSegment* calculates the new orientation dependent form the origin point as an angular misalignment to the coordinate axis. With the help of three mappings, the angle values (x, y and z) are transported to the GroIMP simulation. The last pre-mapping is used to inform the agent simulation to create a new side branch. In the GroIMP model, an integer attribute saves for each root segment the value zero for no new branch and one for a new branch. If the simulation decides to create a new branch, the main-mapping will not be executed directly. Before the next main-mapping can be performed, six new mappings instances have to be created for the new (a total of two) root segments in both simulations. Afterwards, the pre-mappings of this new segment have to be performed, and then both main-mappings. The main-mapping only transports the current length of the *RootSegment* from the agent simulation to the corresponding root segment in GroIMP. In this way, both root elements, in Mason and GroIMP, have always the same length. The root grows in both simulations simultaneously.

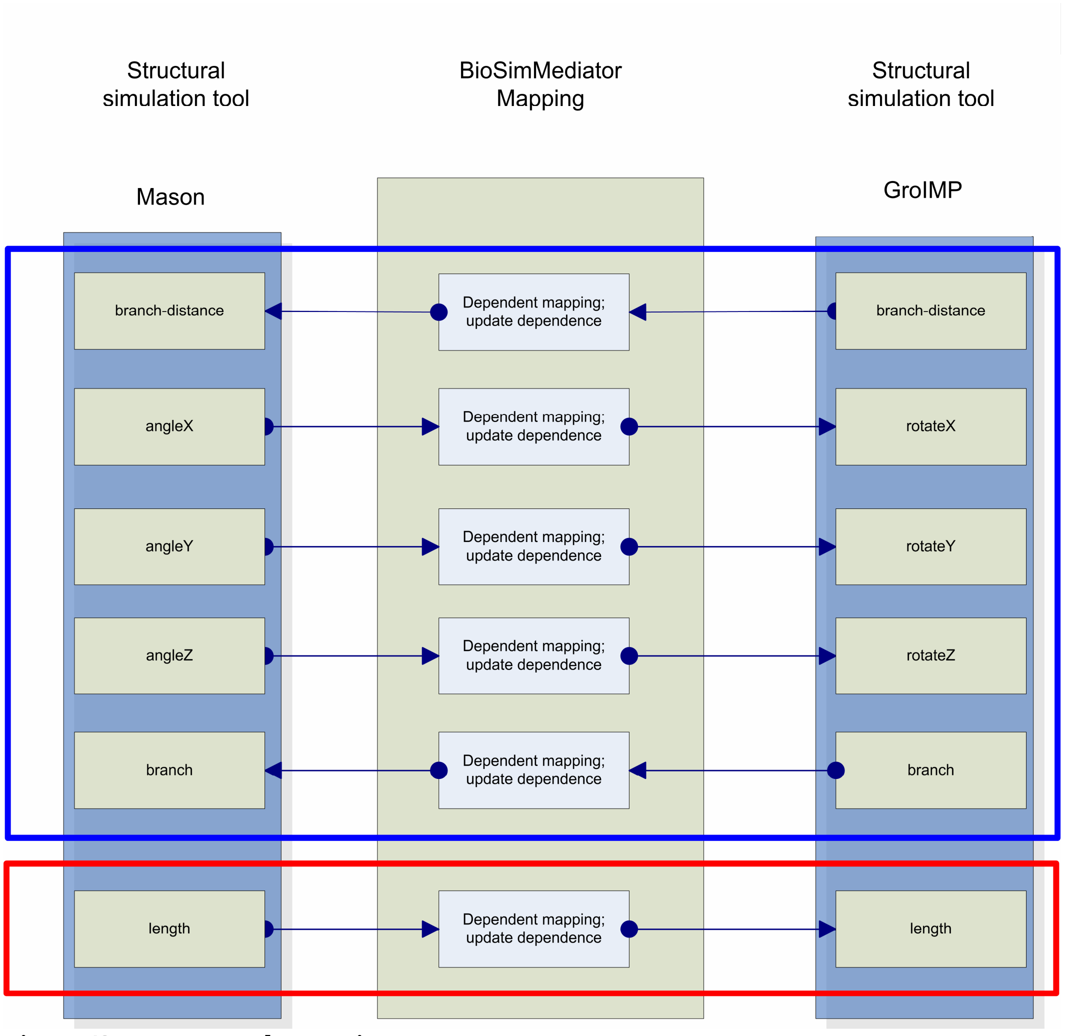


Figure 43: Root example mappings.

Figure 44 shows the visualisation of a root simulation in the agent simulation tool Mason. The brown-grey globes illustrate areas in the soil with the maximum density value of six, which cannot be penetrated by the root. All other densities are not visualised in the illustration. Each blue cylinder represents a *Root* agent in the simulation. Currently, two *RootSegment* are shown in the figure. The branching point is dark blue. As a result of the used standard gravity in this example, each root segment grows directly downwards. Furthermore, the current soil density has no effect on the root growing process. In contrast, the next figure (Figure 45) shows the influence of the soil density on the root growing process. The root starts growing sideward to avoid the high density area. Afterwards, the root starts growing downwards, until the next high level density areas occur and a sideward growing direction is necessary.

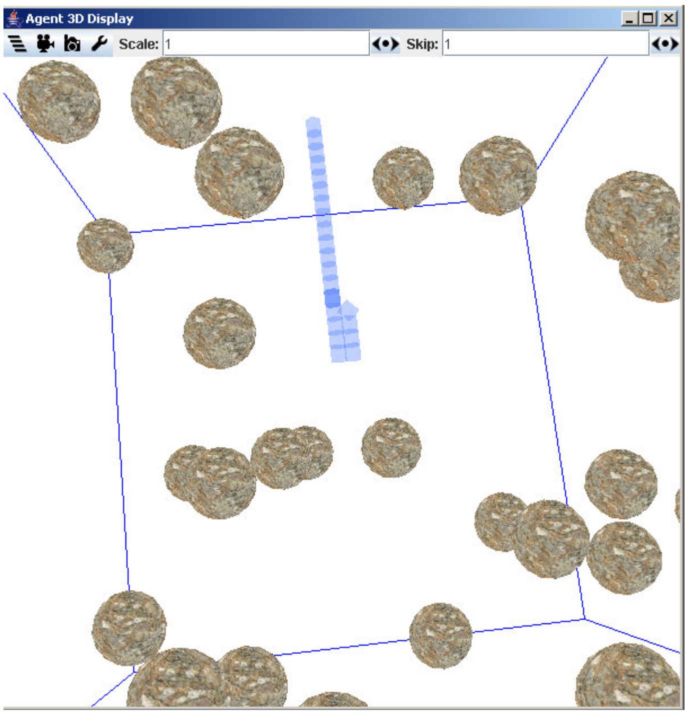


Figure 44: 3D visualisation of the root example in the agent simulation Mason. The globes highlight soil with the maximum density value of six. All other soil densities are not shown in the visualisation. The blue cylinder illustrates the root. Each of them is one *Root* agent.

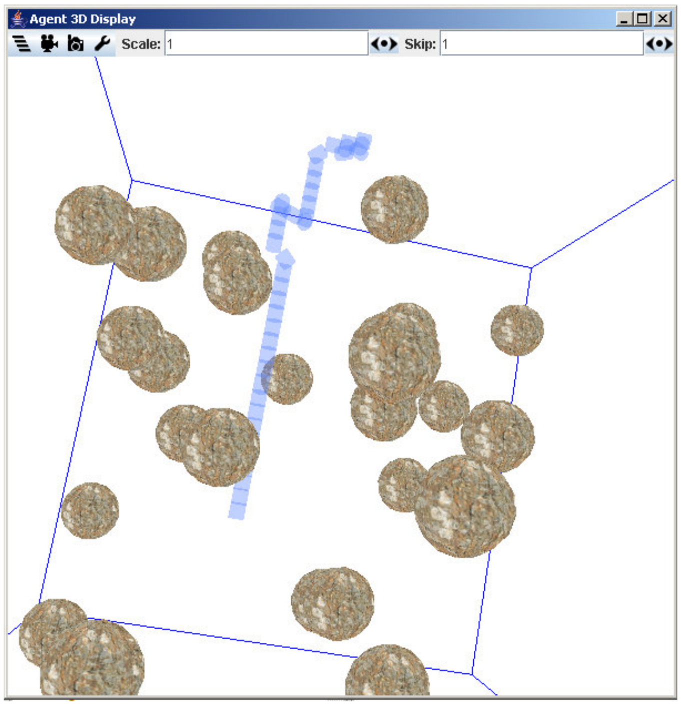


Figure 45: Example visualisation of another root simulation. Here, the root growing is affected by the soil density.

In the following figures the same simulation state is represented. The first figure (Figure 46) shows the visualisation of the agent simulation. In the current simulation state, the simulation consists of a main root segment and various side branches. Furthermore, all root segments avoid the high density areas and grow around these areas. Figure 47 hides the high level density visualisation to clarify the root growing process around these areas. To compare the simulated root structure in the agent simulation with the GroIMP simulation, Figure 48 shows the visualisation of the GroIMP simulation in the same simulation state. The root structure in both simulation tools looks quite similar. Only the orientation of both illustrations is slightly different. First of all, this is affected by the problem that both tools work with different coordination systems and the user has to write its own transformation between both. In the second place, the root elements in Gro-IMP are only visualised by a straight line. Only the start and end point of these elements are the same than in the agent simulation. Finally, the GroIMP image is rotated around the y-axes by a few degrees to get a better overview of all root segments in the simulation.

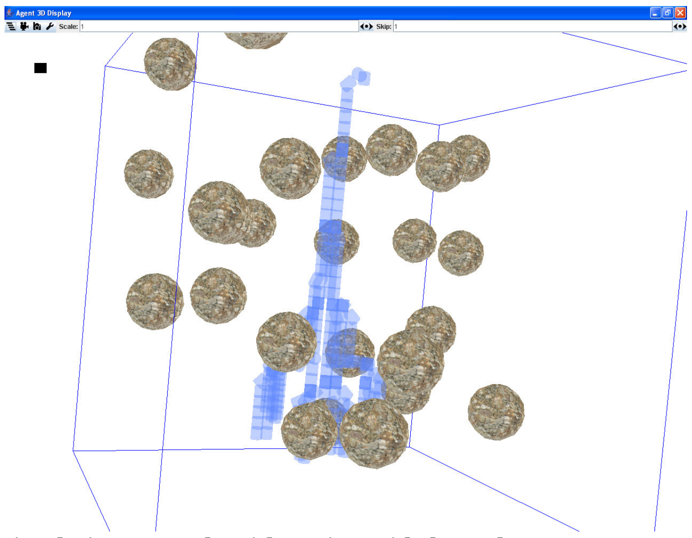


Figure 46: Root simulation example with various side branches.

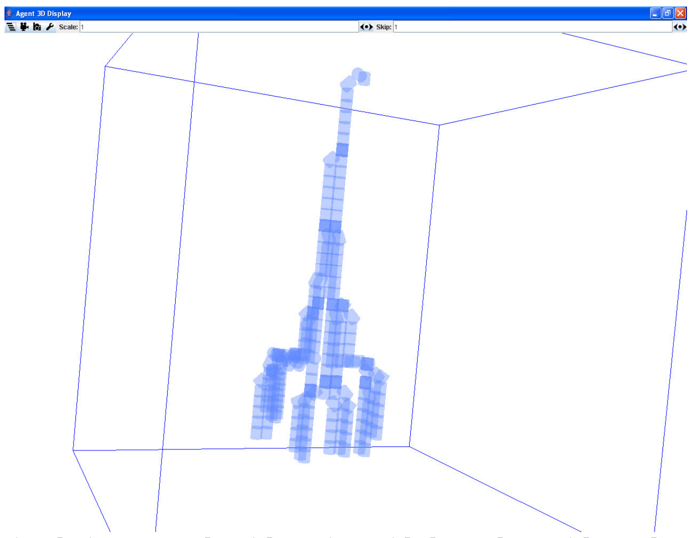


Figure 47: Root simulation example with various side branches, without the density visualisation.

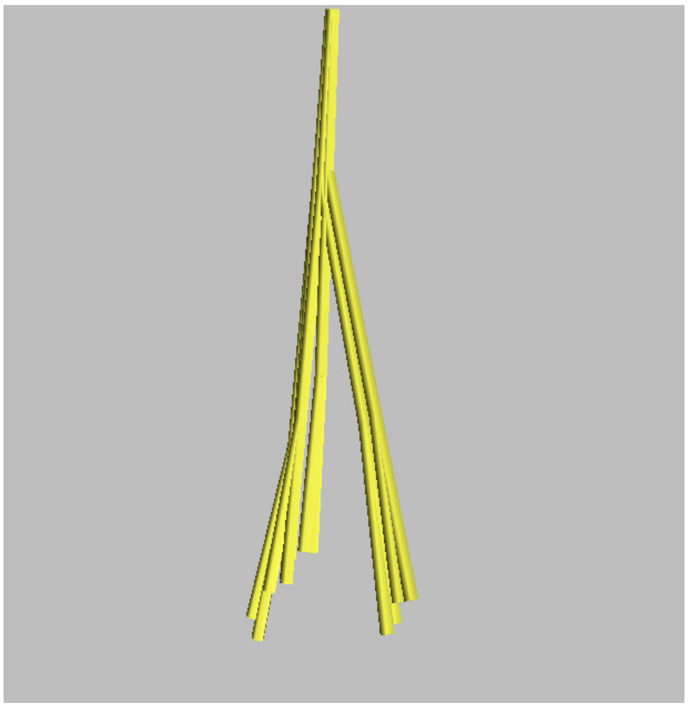
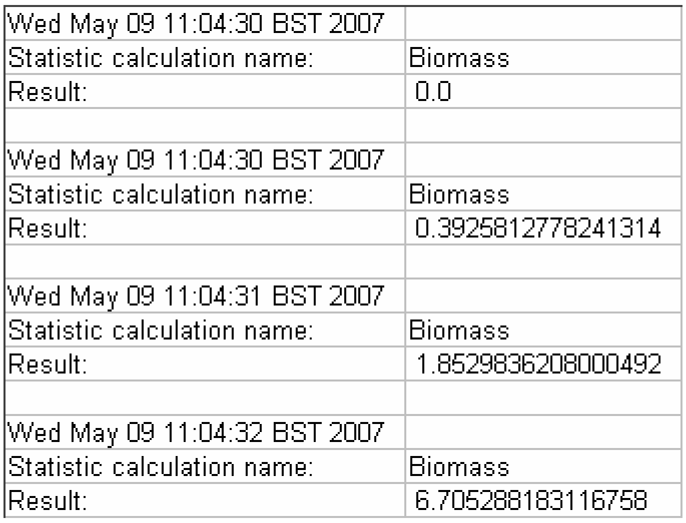


Figure 48: Root structure visualised with GroIMP.

The quite simple root model in GroIMP allows the use of the predefined biomass static function of the mediator in this simulation. This function calculates in each simulation step the volume of the entire root and multiplies the result with the density of water by 10 degree Celsius (0.9997 g /cm^3). The result of the statistic function is presented for four simulation steps in Figure 49.



Wed May 09 11:04:30 BST 2007	
Statistic calculation name:	Biomass
Result:	0.0
Wed May 09 11:04:30 BST 2007	
Statistic calculation name:	Biomass
Result:	0.3925812778241314
Wed May 09 11:04:31 BST 2007	
Statistic calculation name:	Biomass
Result:	1.8529836208000492
Wed May 09 11:04:32 BST 2007	
Statistic calculation name:	Biomass
Result:	6.705288183116758

Figure 49: Biomass calculation of the root example.

In conclusion, the root example shows the applicability of the mediator framework to more complex examples. During the simulation process a multitude of mapping instances are used for each root segment pair. The *Dependent* mappings allow the correct mapping between belonging root segments. In addition, the example highlights the exigencies for the differentiation between pre-mapping and main-mapping. Furthermore, the statistic unit of the mediator framework is used in the example.

7. Discussion

In this chapter, I discuss the advantages and the disadvantages of the invented framework, as well as problems which have been occurred during the system tests. In addition, I point out the possible optimisation potential. At the end, I give an outlook over planned increments.

The goal of the thesis is to introduce a new software framework which combines existing biological simulations tools. The thesis shows the design and the implementation details of the system. In the previous chapter Results, the applicability of the system by means of two examples is shown. The focus is put on the presentation of the different mediator functionalities and not on the biology. Therefore, the examples have no qualify for biological correctness. However, the examples show the different mapping types and the log unit as well the static unit of the mediator in use. Also, the independent control of each connected simulation tool is presented in the examples. As mentioned before, the mediator is able to interact with each simulation tool independently, and saves for each of them separately information about the simulation step size or the required log and statistic functions.

7.1. In comparison to existing tools

The mediator framework is the first software system which allows combining a multitude of different existing simulation tools and defining dependencies between those in the presented way. Furthermore, the framework offers a log and a statistic unit. In contrast to the introduced tools in chapter *Fundamentals and Related Work*, like the SBW workbench or the Bio-Spice framework, the mediator framework is able to control all connected tools and define mappings between different simulation attributes. With the help of the different mapping types, the system is nearly able to handle all possible occurring dependencies between the different models. Also, the user can, to avoid unintentional dependences between the mappings, define an order in which the mediator has to perform the mappings. For the mapping process and the internal storage of simulation attribute values, the mediator system uses performance optimised collections, based on the *Fastutil* collection API. Jan Taubert (see [Tau05]) shows in his diploma thesis the performance advantage of the *Fastutil* collections in comparison to the standard Java collections. Furthermore, the number of possible connected simulation tools to the mediator framework is (theoretical) unlimited. The entire software system is implemented in the Java programming language to ensure platform independence. A connection to simulation tools which are implemented in other programming languages is easy possible by using the SBW workbench. The SBW provides different kind of interfaces for nearly all known programming and scripting languages. Via this software system the mediator is able to interact with those tools.

In contrast to all current developed hard coded simulation systems, the mediator approach allows different kinds of scientists to work on the same simulation and to concentrate on their area of expertise. For instance, one scientist can model the biochemical process with the help of Petri Nets or similar graph based approaches, and another scientist can focus on the structural models of the biological system. All developers work with familiar tools and a supervisor can map all those tools with the mediator system to an entire simulation together. It is not necessary to re-implement all functionalities (modelling abstractions, mathematical solvers, etc.), which already exist in tools, like in hard coded simulations. Nearly all of them (like the virtual heart simulation, see chapter *Fundamentals and Related Work*) re-implement for instance, fluid dynamic simulation parts and algorithms for the visualisation. All those re-implementations are unnecessary by using the mediator framework. Instead of them, the user has only to define the mappings between all involved and connected simulation tools.

Additionally, the clear design and interface structure of the system allows the users to implement their own new mappings or statistic functions to the mediator. In principle, the framework only provides a set of predefined functions and regulations, and each user can add new elements quite easily to the system. The interface structure, quite a few guidelines and examples help the users with the implementation. With reference to the system architecture (three tier architecture and component based architecture), a complete new system unit can be added to the framework quite easily. Furthermore, the system architecture and the usage of object instances, for the storage and transport of the simulation attributes inside the mediator, allow easy further changes.

The disadvantages of the mediator approach in comparison to existing approaches are, besides the performance (which is discussed in the next subsection), are limited control over the connected simulation tools, and the lack of a time correlation.

Most of the current existing simulation tools or frameworks allow the user to run their simulation for x time steps and to store the current system state persistent (for instance, in a file). Next time, the user can start the simulation at this time point, and in that way, a time intensive simulation can be interrupted and restarted later on. The problem with this functionality in the mediator framework is the missing functionality in some of the used simulation tools. For instance, the agent simulation tools Mason offer this function, but functional simulation tools like RoadRunner or JDesigner do not offer such a function. Therefore, the current version of the mediator is not able to save the entire system state and to restart the simulation later on at this point. To avoid this problem, the used simulation tools have all to provide such functions and the mediator framework has to be modified.

Currently the mediator framework uses for the control of the simulation progress an abstract step size unit. The user can define, during the initialisation process, how many simulation steps in the simulation tools shall be performed in relation to the mediator steps. In biological simulations the time plays nearly always an important role. With this abstract step size a correlation to the real time is nearly impossible. Similar to the control problem of the mediator, each simulation tool uses a different internal simulation time or step size unit. With regard to this, it is not possible for the mediator to use a realistic time unit for the simulation progress control.

7.2. Performance

The mayor and most obviously disadvantage of the mediator framework, in comparison to hard coded simulation tools, is the performance. In a hard coded simulation system the designer and programmer can choose for the interaction of all parts an optimised system design and implementation. The mediator framework has to be generic enough to communicate with the different simulation tools and for the mapping between them. Only the transport of values in the mediator, the mapping process and the interaction of the mediator with the simulation tools can be optimised. The optimisations at this point are limited to a good system design and the usage of fast collections. During the implementation tests, the performance bottle neck was always the interface to the simulation tools. In contrast to the GroIMP system, all tools which are connected via the SBW framework interact quite fast with the mediator. As mentioned before, the interaction with GroIMP is not so easy, because the system has not been designed for a communication with other tools in that way. The mediator and other simulation systems have to wait until the GroIMP internal process is finished. For more details of the interaction with GroIMP see chapter *GroIMP interface*.

The mediator framework is applicable for small and medium complex simulations. The limitation at this point is not the number of mappings or the number of connected subsystems, but the speed of the connected subsystems itself and the interaction with them. Simulations which use a combination of only functional simulation tools can be much more complex than simulations which use one or more structural simulation tools. All used structural simulation tools are limited in the simulation performance. In addition to highly calculation intensive 3D visualisations, the performance problem is based on disadvantageous system designs and on the complexity of the simulations.

As mentioned in the last subchapter, the mediator uses optimised collections for the storage of simulation attributes. Furthermore, a unique identifier is dedicated to each attribute and mapping. With the help of optimised HashMaps the mediator can find each of them in nearly constant time.

Beside the performance problem, the mediator is nearly completely implemented as a single thread system and optimised for a single processor system. Only the log and statistic unit uses different threads to accelerate the execution. With the help of the SBW workbench some simulations can be relocated to other computer systems, but the most calculation intensive mapping and storage of the simulation attribute values processes are executed all at one computer. The mayor problem with a multi thread system is the synchronisation of the different mappings. Most of the mappings depend on each other. A parallel execution of the reading and writing process in each simulation subsystem can improve the system performance.

7.3. Outlook

The current version of the mediator system is applicable for small simulations. At the moment, six simulation tools (RoadRunner, Jarnac, JDesigner, Mason, GroIMP and the mathematical utility tool) are connected with the mediator. One future task will be to add more simulation tools to the mediator. More simulation tools offer the users a wider range of tools, and they would be able to choose the optimised tool for a specific simulation part. With respect to the interface design of the mediator, new tools can be added quite easily. Especially, a connection to a mathematical tool like Mathematica or MatLab would be preferable. These tools offer optimised toolkits for different use cases. For instance, fluid dynamic processes can be calculated with the help of the fluid dynamic tool kit in MatLab, or equations can be solved with a higher precision than with other tools. With the help of these new features, the examples can be more realistic. For example, with the fluid dynamic tool kit it is possible to simulate the water concentration in the soil and its dynamic, and use this for a more detailed root model (see Example: Root development). Furthermore, the fluid dynamic can also be used for the simulation of nutrition transport in plants or organism.

So far, only small and simple examples are tested with the mediator. The next step would be to create a more realistic example, and comparing the results of a hard code simulation with a distributed simulation using the mediator. Afterwards, the model complexity should increase. Tully Yates, from the Rothamsted research institute in Harpenden, is planning to use the mediator framework to simulate Arabidopsis thaliana with RoadRunner and GroIMP.

Additionally, the interaction and communication with the GroIMP system has to be revised. Together with the developer of the system the interface has to be changed, and the mediator must be able to get access to the internal thread handling process of GroIMP to avoid the busy waiting structure. In the first place, the mediator's internal copy of the GroIMP RGG graph has to be replaced by a direct access to the RGG graph in GroIMP. Also, the possibility to disable the visualisation and to use the XL language separately would bring a big performance benefit.

The current (mediator) version is optimised for a single processor system. To obtain a better performance, the system must be able to be used on a cluster or a grid computer system. In the first place, all simulations should be executed at different systems and only the mapping process should be performed at a master computer. Consequently, each subunit in the mediator must use its own thread. Also, a new system unit has to be created, to control all threads and guarantee the synchronisation for the right execution order (extension of *GlobalControl*). The current system design allows quite easy to include a new layer or a kind of information (object) transport between the existing layer zero (simulation subsystems) and layer one (data storage) (see Figure 29). Different kinds of approaches can be used. Each of them has vantages and disadvantages in respect of performance, complexity and reliability, and therefore, the decision has to be elaborated.

The first possibility is to use the Java object serialisation and to transport the object and control commands via a standard socket connection. The second approach is to use the Java remote method invocation (RMI). RMI distinguishes between local and remote objects. The calling method is not able to distinguish between a local and a remote object and handle both equally. COBRA (common object request broker architecture) uses, similar to the RMI approach, also an object spreading. This approach, in comparison to RMI, is not mandatory bounded to the Java programming language. The last possible approach uses the Java Message Service (JMS) to communicate via messages between the different subsystems. A detailed introduction to all this methods can be found in the book, 'Middleware in Java' [Hei05].

Finally, a graphical user interface has to be created, to allow the user an easy configuration of the mediator. The configuration includes the definition of the mappings, the used log unit as well as the statistic unit, and also the required simulation subsystems and models.

8. Conclusion

The mediator framework is a new approach to combine existing simulation tools. With the help of the framework, scientists can split a biological simulation into parts, and use for each of them a specific and optimised simulation system. The main advantage, in comparison to existing simulation tools (hard coded tools), is the reusability of existing tools and the generic mapping between them. A number of features, like the log and statistic unit, or the various mapping types, allow the scientists to use the mediator framework for realistic simulations of biological systems.

Currently, the mediator framework is more or less a prototype implementation, but completely usable. With all the proposed extensions, the mediator approach could play an important role in the system biology in future. However, each time a very complex and calculation intensive simulation are necessary, a hard coded and specific optimized system is the first choice. The application area for the mediator should be small groups of scientists, which concentrate on small and medium complex models.

Beside the invention of new tools and models, the main goal of biology and especially the system biology, is to close existing knowledge gaps in order to increase the quality of the existing models. Also, data mining helps to order and to categorise existing knowledge and assists the scientists in future research.

The quality of computer models and simulations depends on the known knowledge. Without the entire knowledge of the real system (structure, biochemical processes, dynamic of the system and interaction with the environment), the fastest computer system and the best simulation system is not able to simulate the real system correctly.

Appendix

A. CD-ROM

The CD-ROM contains:

- all required program sources as JAR archive files
- short installation manual
- the complete thesis in digital form

Acknowledgment

I am grateful to my supervisors Dr. Jacob Köhler, Prof. Dr. Kurt Lautenbach and Dr. Stephan Philippi. I thank Tully Yates and Artem Lysenko for the fruitful discussions and their help. I also thank Prof. Dr. Chris Rawlings, Jan Taubert, Paul Verrier, Matthew Hindle, Rainer Winnenburg, Nathalie Brooke and Sabrina Holland. Finally, I thank Rothamsted Research and Leonardo Da Vinci (Trier, Germany) for financial support.

List of figures

Figure 1: Iterative model creation process. 4
Figure 2: OpenCI: GA20 oxidase simulation with time course simulation via SBW and roadRunner. 6
Figure 3: The individual models communicate via the SBW broker (from [Huc07]). 7
Figure 4 Cellular automata after Ulam [Ula66]. 8
Figure 5: Basic elements of MASON (from [Luk05]) 9
Figure 6: L-system and turtle interpretation. 10
Figure 7: Relation between the Chomsky hierarchy and L-System classes. OL stands for the class generated by context-free L-Systems and IL for the class generated by context-sensitive L-systems. 11
Figure 8: Mycelis muralis, based on a context-sensitive L-System from [Pru90]. 12
Figure 9: Lilac inflorescences from [Pru90]. 12
Figure 10: General syntactic structure of an RGG rule. Simplified, the rule replaces *L* by *R and executes P* (from [Kur07]). 13
Figure 11: Relational growth grammar rule (upper part) and corresponding graph (lower part) (from [Kur07]). 14
Figure 12: RGG rule for genetic crossing-over (from [Kur07]). 14
Figure 13: Koch construction step. Replace the middle part with two lines. 15
Figure 14: Basic principle of the mediator concept. The connections between the software system and the simulation tools are bidirectional. 20
Figure 15: Connection and communication between BioSimMediator and the subsystems. 20
Figure 16: Mappings between the simulation subsystems. Capitals are synonym for the simulation tools and the lower case letters stand for attribute values inside these systems. 21

Figure 17: Use case diagram for the BioSimMediator............................. 23
Figure 18: Currently connected simulation tools. 25
Figure 19: Class diagram for the handle subsystem. 28
Figure 20: Class diagram: control subsystem. 30
Figure 21: Transport of attribute values between simulation subsystems.32
Figure 22: Mediator object classes for the transport of simulation attribute values. 34
Figure 23: Statechart diagram to illustrate the correlation between one mediator step and the mapping. 36
Figure 24: All possible mapping categories. Each path in the binary tree from the root '*Mapping'* to an end node contains one mapping. 38
Figure 25: Class diagram for the mapping unit of the mediator................ 39
Figure 26: Class diagram for the log subsystem. 42
Figure 27: Statistic unit design in a UML class diagram.......................... 45
Figure 28: UML class diagram shows the global control object classes of the mediator... 47
Figure 29: Three tier architecture of the mediator framework. 48
Figure 30: Package diagram of the mediator framework. 49
Figure 31: Sequence diagram illustrate the method calls during a mediator step.. 52
Figure 32: GUI for the mediator steps size control.................................. 53
Figure 33: UML sequence diagram illustrate the system dynamic during the mappings. 55
Figure 34: GroIMP interface design. 57
Figure 35: Mediator framework (illustration by Tully Yates). 58
Figure 36: ABC model of flower development (from [Wol02]). 60
Figure 37: Classes of genes are responsible for the development of the different flower organs (from [Alb02]).. 61
Figure 38: Petri Net model for the ABC model (by Tully Yates, Rothamsted Research, United Kingdome).. 61
Figure 39: Mapping of the ABC model. 62
Figure 40: Log file of the ABC model example... 64
Figure 41: Root and RootSegment agents and the density of the soil in 2D. 66

Figure 42: Agent root growing direction. 67
Figure 43: Root example mappings. 69
Figure 44: 3D visualisation of the root example in the agent simulation Mason. The globes highlight soil with the maximum density value of six. All other soil densities are not shown in the visualisation. The blue cylinder illustrates the root. Each of them is one *Root* agent. 70
Figure 45: Example visualisation of another root simulation. Here, the root growing is affected by the soil density. 70
Figure 46: Root simulation example with various side branches.............. 71
Figure 47: Root simulation example with various side branches, without the density visualisation. 71
Figure 48: Root structure visualised with GroIMP. 72
Figure 49: Biomass calculation of the root example................................. 72

List of tables

Table 1: Turtle geometry: Geometrical interpretation of characters in L-systems. 10
Table 2: Illustrate the relation between iteration step, string and resulting geometrical form for the example. 11
Table 3: Geometric outcome after n durations for the snowflake example 16
Table 4: Vantages and disadvantages of the embedded and the mediator approach. Plus sign connote vantage and minus sign connote disadvantage. 19
Table 5: Simulation results: ABC model of flower development.......... 63

Bibliography

[Abe82] Abelson, H.; DiSessa, A. A.: Turtle Geometry. Cambridge, MIT Press. (1982)

[Alb02] Albert, Victor A.: Genetics of flower morphology; http://folk.uio.no/victoraa/MH_Ybk_Albert_2002.pdf, accessed 0.1.08.2007

[BIB07] Biobase: Biological Databases; http://www.biobase.de/; accessed 22.06.2007

[BN+07] Centre for Bioinformatics Saar: BN++; http://fred.bioinf.uni-sb.de9180/BNPP/; 22.06.2007

[Cal06] Calder, Muffy; Vyshemirsky, Vladislav; Gilbert, David and Orten, Richard: Analysis of Signalling Pathways using Continuous Time Markov Chains. *Lecture Notes in Computer Science* 4220:pp. 44-67. (2006)

[Che97] Chen, S.G.; Impens, I. and Ceulemans, R.: Modelling the effects of elevated atmospheric CO2 on crown development, light interception and photosynthesis of poplar in open top chambers, Global Change Biology, Volume 3, Number 2, April 1997 , pp. 97-106(10), (1997)

[Cho03] Cho, K.H.; Shin, S.Y.; Kim, H.W.; Wolkenbhauer, O.; McFerran, B. and Kolch, W.: Mathematical modelling of the influence of RKIP on the ERK signalling pathway. Lecture Notes in Computer Science, 2602:127-141, (2003)

[Coh67] Cohen, D.: Computer Simulation of Biological pattern generation processes, Nature, 216, 246-248, (1967)

[Coh96] Cohen, S.D. and Hindmarch, A.C.: CVODE, a stiff/nonstiff ODE solver. C. Computer in Physics, 10.2: 138-143, (1996)

[Fin03] Finney, A. and Hucka, M.: Systems Biology Markup Language: Level 2 and Beyond. Biochem. Soc. Trans. 2003, 31: 1472-1473,(2003)

[Gam95] Gamma, Erich; Helm, Richard; Johnson, Ralph E.: Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, Amsterdam; Edition: 1st ed., Reprint (1995)

[Gar03] Garvey, Thomas D.; et.al.: BioSPICE: Access to the Most Current Computational Tools for Biologists; OMICS: A Journal of Integrative Biology, Dec 2003, Vol. 7, No. 4 : 411 -420, (2003)

[Gas01] Gasparíková, O.; Ciamporová, M.; Mistrík, I.; Baluska, F. (Eds.): Recent Advances of Plant Root Structure and Function
Proceedings of the 5th International Symposium on Structure and Function of Roots held August 31-September 4, 1998, Stará Lensná, Slovakia, Springer Netherlands, (2001)

[Gib99] Gibson, M. A. and Bruck, J.: Efficient exact stochastic simulation of chemical sytems with many species and many channels. Journal of Computional Physics, A 104, 1876-1889, (1999)

[Gil76] Gillespie, D. T.: A general method for numerically simulating the stochastic time evolution of coupled chemical species. Journal of Computational Physics 22: 403-434, (1976)

[Gor99] Goryanin, I.; Hodgman, T.C.; Selkov, E.: Mathematical simulation and analysis of cellular metabolism and regulation. Bioinformatics 15, (1999).

[Hei05] Heinzl, Steffen; Mathes, Markus: Middleware in Java, 1. edition, Vieweg Verlag, Wiesbaden, (2005)

[Hin83] Hindmarch, A. C.: ODEPACK: a systematized collection of ODE solvers. Scientific Computer, (1983)

[Huc07] Hucka, M.; Finney, A.; Sauro, H.; Bolouri, H. and Bergmann, F.: Biology Workbench Java^{TM} Programmer's; http://128.208.17.155//caltechSBW/sbwDocs/docs/api/Java/Java_API.pdf; (2004) accessed 03.07.2007

[Kel01] Kelemen, Jozef; Sosik, Peter: Advance in Artificial Life (Proceedings of the 6th European Conference of Artificial Life), 242-251, Springer-Verlag, Berlin-Heidelberg, (2001)

[Kit05] Kitano, Hiroaki; Funahashi, Akira; Matsuoka, Yuhiko and Oda, Kanae; Using process diagrams for the graphical representation of biochemical networks. Nature Biotechnology 23.8, 961-966, (2005)

[Kle02] Klein, J.: breve: a 3D simulation environment for the simulation of decentralized systems and artificial life. Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems. The MIT Press. (2002)

[Kli05] Klipp, Edda; Herwig, Ralf; Kowald, Axel; Wierling, Christoph; Lehrach, Hans (2005): Systems Biology in Practice. Weinheim; Wiley-VCH, 1. Edition, ISBN-10: 3-527-31078-9

[Koe06] Köhler, Jacob; Baumbach, Jan; Taubert, Jan; Specht, Michael; Skusa, Andre; Rüegg, Alexander; Rawlings, Chris; Verrier, Paul and Philippi, Stephan: Graph-based analysis and visualization of experimental results with ONDEX; Bioinformatics 22(11) (2006)

[Koh00] Kohl, Peter; Noble, Denis; Winslow, Raimond L.; Hunter, Peter J.:Computational modelling of biological systems: tools and visions,The Royal Society, Volume 358, Number 1766/January 15, 576-610, (2000)

[Kon07] Konrad-Zuse-Zentrum; Kober et al.: Anisotropic simulation of the human mandible; http://www.zib.de/Publications/Reports/ZR-04-12.pdf; accessed 05.07.2007

[Kur07] Kurth, Winfried: Specification of morphological models with L-systems and relational growth grammars, Image, Journal of Interdisciplinary Image Science, Vol. 5, (2007)

[Luk05] Luke, Sean; et al.: MASON: A Multiagent Simulation Environment, *SIMULATION*, 81: 517-527, (2005)

[Man06] Manninen, T.; Makiraatikka, E.; Ylipaa, A.; Pettinen, A.; Leinonenm, K. and Linne, M. L.: Discrete stochastic simulation of cell signaling: comparison of computational tools, Engineering in Medicine and Biology Society, (2006). EMBS '06. 28th Annual International Conference of the IEEE

[Min04] Ming, P.; Albrecht, J.: Integrated Framework for the Simulation of Biological Invasions in a Heterogeneous Landscape, Transactions in GIS, Volume 8, Number 3, June 2004 , pp. 309-334(26), (2004)

[Nag04] Nagasaki, M.; Doi, A.; Matsuno, H. and Miyano, S.: Genomic Object Net:I. A platform for modeling and simulating biopathways, Applied Bioinformatics 2:181-184,(2004)

[Nag04+] Nagasaki, M.; Doi, A.; Matsuno, H. and Miyano, S.: A Versatile Petri Net Based Architecture for Modeling and Simulation of Complex Biological Processes, Genome Informatics, 13:180-197, (2004)

[Per07] Perttunen, Jeri: The Functional-Structural Tree Model LIGNUM; www.sal.hut.fi/Personnel/Homepages/JariP/thesis/summary_perttunen, accessed 10.07.2007

[Pru90] Prusinkiewicz, Przemyslaw; Lindenmayer, Aristid: The Algorithmic

Beauty of Plants. New York, Springer, (1990)

[Ram05] Ramsey, S.; Orrel, D. and Bolouri, H.: Dizzy: stochastic simulation of large-scale genetic regulatory networks. Journal of Bioinformatics and Computational Biology 3.2: 415-436, (2005)

[Red96] Reddy, VN; Liebman, MN; Mavrovouniotis, ML: Qualitative analysis of biochemical reaction systems; Comput Biol Med. (1996) Jan; 26(1):9-24.

[Roz97] Rozenberg, Grzegorz: Handbook of Graph Grammars by Graph Transformations, Vol.1, Foundations. Singapore, World Scientific, (1997)

[Saur00] SAURO, H. M.:Jarnac: a system for interactive metabolic analysis. (2000) Ch. 33, 221- 228, Animating the Cellular Map 9th International BioThermoKinetics Meeting (eds: Hofmeyr, J-H. S, Rohwer, J. M, Snoep J. L) Stellenbosch University Press, ISBN 0-7972-0776-7

[Saur07] Sauro Lab, University of Washington:JDesigner: A Biochemical Network Layout Tool;http://128.208.17.155//software/jdesigner.htm; accessed 02.07.2007

[Tau05] Taubert, Jan: Database Integration and Analysis of Biological Neworks Methods and Optimisation of ONDEX, Diploma thesis, University Bielefeld, (2005)

[Ula66] Ulam, S.: Pattern of growth of figures: Mathematical aspects; Module, Proportion, Symmetry, Rhythm, 64-74. Braziller, New York, (1966)

[Ven01] Venter, J. Craig; *et al. :* The Sequence of the Human Genome; *Science* 291, 1304 (2001)

[Vos03] Voss, Klaus; Heiner, Monika and Koch, Ina: Steady state analysis of metabolic pathways using Petri nets; *In Silico* Biology 3, 0031 (2003).

[Wei98] Weimar, Jörg R: Simulation with Cellular Automata, Logos-Verlag, Berlin, (1998). ISBN 3-89722-026-1

[Wil06] Wilkinson, Darren James (2006): Stochastic Modelling for Systems Biology: Mathematical and Computational Biology; CRC Press Inc, 2006.

[Wol02] Wolpert, Lewis: Principles of Development, Oxford University Press, Oxford, (2002)

[Zus07] Zuse Institute Berlin: Affin-invariant Newton Techniques NLEQ2; http://www.zib.de/Numerik/numsoft/ANT/nleq2.de.html; accessed 22.06.2007