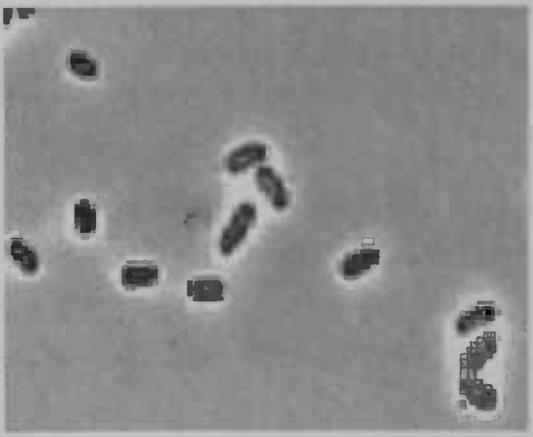
Model Intestinal Microflora In Computer Simulation:

MIMICS-J

Java Simulation Code Generator



Roeland Werring
Student nr. 0962694
University of Groningen 2003: MSc-thesis
Email: roeland@werring.net
Supervisors: dr. M.H.F. Wilkinson & prof. dr. J. Bosch

Table of contents

1. Introduction

The Centre for High Performance Computing of the University of Groningen has developed a project called MIMICS (Model Intestinal Micro-flora In Computer Simulation) [1]. The aim of this project is to develop simulation tools that can be used to examine the interaction between the host and its intestinal micro-flora.
The main tool of MIMICS is a large scale cellular automation, initially designed to examine the interactions in the lumen of the intestine [2]. This program was written in the program language FORTRAN 90. It has been optimized for parallel computation and runs on the Cray SV1e supercomputer.

Main quality that was envisaged in the development of the current tool was speed of computation whilst the ease of use or modifiability was considered to be less important. This means that with the current tool it is difficult to add new interactions. Therefore addition of a wall in the simulation [4] did already require considerable redesign of the tool.

In order to make it easier to carry out new simulation tasks, a tool should be developed that provides more flexibility. A graphical user interface should be added for improvement of the ease of use. However, not too much speed should be sacrificed, especially when the model becomes more complex and computationally intensive.

In this context a new project has been started up, which is subject of this report.
First aim of this project is to explore which design strategy can provide a tool that has sufficient modifiability, ease of use and speed. For that purpose, the applicability of the existing software engineer methods [5, 6] will be investigated.
On the basis of the chosen architecture, a new tool will be developed. Finally this tool will be implemented in order to demonstrate its functionality.

Chapters 2 and 3 of this report are the result of a literature study into the current MIMICS project and evaluation tools of software architecture of programs. In chapter 4 the terms of reference for the new software tool are defined, including the evaluation method that will be chosen to assess notably its modifiability. In chapter 5 and 6 the development and implementation of the new software tool is described. Chapters 7 and 8 explain the user interface of the new tool, and how to create a simulation. Finally, chapter 9 gives an analysis of the new the tool.

2. The current MIMICS project

2.1 General description

The intestinal micro-flora forms a highly complex community of an estimated 400 species. The exact number of these species, and even their nature and role in the ecosystem, are not really known because the major part (60-85%) of the microscopically visible bacteria in faecal content cannot be cultured [1].

There is an urgent need to understand better the bio-chemical reactions of intestinal micro-flora, because of the rapid increase of antibiotic resistance of pathogenic bacteria [1]. The intestine ecosystem is considered to be a first line of defense against invading pathogens. The better understanding of this very complex effect, called colonization resistance, may contribute to the development of alternatives for antibiotics, such as priobotics (live bacteria taken orally).

In order to provide alternatives for laboratory experiments, a quantitative simulation tool was developed, which is the main tool of the MIMICS project. This tool is a large scale cellular automaton which can simulate both metabolic and transport processes in the human intestine [1]. The tool has been written in the program language FORTRAN 90, and optimized for parallel computations on the Cray SV1e supercomputer of the Center for High Performance Computing. Initially, only the interactions in the lumen of the intestine were included in the program. The structure of this tool is described in the next sections.

2.2 The conceptual model as used in MIMICS

The simulation can be divided in five (somewhat interrelated) parts:

A. The bacterial metabolism
B. The chemistry of the environment
C. The geometry of the environment
D. The mechanics of transport
E. The interaction with the immune system

At this stage, the immune system is left out of the model. This is because of its complexity, and the fact that the majority of bacteria in a healthy intestine does not seem to evoke an immune response. Therefore it is realistic to keep the immune system out of the model.

Bacterial metabolisms & the chemistry of the environment

If the focus is on distinction between aerobic and anaerobic metabolisms, then bacteria can be classified in to six types.

Four of the types of bacteria grow in completely oxygen free conditions:

Strict anaerobe:	Can not survive with any oxygen
Moderate anaerobe:	Can survive low concentrations of oxygen
Tolerant anaerobe:	Are unaffected by oxygen
Facultative (an)aerobe	Grow better with oxygen, but also in absence of oxygen

The two others need oxygen to survive:

Microaerophile:	Need low concentrations of oxygen to survive, but will not survive at moderate concentrations
Strict aerobe:	Require oxygen to grow; no toxic effects at normal levels

Using the Monod model [5], the metabolisms can all be modeled with differential equations of the same general form. This is the equation used to model the growth of the bacteria:

$$\mu(S,O_2)=\left(\mu_{an}+\mu_{O_2}\frac{O_2}{K_{R,O_2}+O_2}\right)\frac{S}{K_S+S}-\kappa_{O_2}\frac{O_2}{K_{T,O_2}+O_2}-\mu_{basal} \qquad (1.1)$$

μ	growth rate per unit of bacterial biomass	(s^{-1})
S	concentration of food substrate	$(^{mol}/_{l})$
O_2	concentration of oxygen	$(^{mol}/_{l})$
μ_{an}	maximum growth rate of anaerobic metabolism	(s^{-1})
μ_{O_2}	maximum growth rate of aerobic metabolism	(s^{-1})
K_S	food uptake saturation constant	$(^{mol}/_{l})$
K_{R,O_2}	respiratory oxygen uptake rate constant	$(^{mol}/_{l})$
K_{T,O_2}	toxic oxygen uptake rate constant	$(^{mol}/_{l})$
κ_{O_2}	maximum oxygen kill rate	(s^{-1})
μ_{basal}	Minimal metabolic requirement	(s^{-1})

The first term of (1.1) is the oxygen dependent growth through food uptake. The second term is the cell destruction due to oxygen, the third the maintenance energy cost.

All the above mentioned types of bacteria can be described with this equation by adaptation of the values of the growth- and kill-rates:

Table 2-1: Aerobe and Anaerobe bacteria

	μ_{an}	μ_{O_2}	κ_{O_2}
Strict anaerobe	>0	<0	>0
Tolerant anaerobe	>0	0	0
Facultative (an)aerobe	>0	>0	0
Microaerophile	0	>0	>0
Strict aerobe	0	>0	0

The differential equation for concentration X_k of the k^{th} specie of bacteria associated with growth rate μ_k is as follows:

$$\frac{dX_k}{dt}=\mu_k(S,O_2)X_k \qquad (1.2)$$

For an ecosystem with P species of bacteria, the food usage is described with the following differential equation:

$$\frac{dS}{dt}=\sum_{k=1}^{P}\left(-\left(V_{an,k}+V_{O_2,k}\frac{O_2}{K_{R,k}O_2}\right)\frac{S}{K_{S,k}+S}+Y_{\kappa,k}\kappa_{O_2,k}\frac{O_2}{K_{T,k}+O_2}\right)X_k \qquad (1.3)$$

$V_{an,k}$	maximum specific food uptake rate of the anaerobes	(s^{-1})
$V_{O_2,k}$	maximum specific food uptake rate of the aerobes	(s^{-1})
$Y_{\kappa,k}$	fraction of oxygen killed bacteria returned as food	(s^{-1})

The differential equation for oxygen usage is:

$$\frac{dO_2}{dt}=-\sum_{k=1}^{P}\left(\beta_{R,k}\frac{O_2}{K_{R,k}+O_2}\frac{S}{K_{S,k}+S}+\beta_{T,k}\frac{O_2}{K_{T,k}+O_2}\right)X_k \qquad (1.4)$$

$\beta_{R,k}$	maximum oxygen uptake rates due to aerobic metabolism	(s^{-1})
$\beta_{T,k}$	maximum oxygen uptake rates due to toxic effect on anaerobes or microaerophiles	(s^{-1})

Bacteria are capable of producing toxins, called bacteriocins, which are harmless to the host, but highly toxin to their competitors. If a bacterium X_k produces toxin T at a constant rate $Y_{T,k}$, then its growth is reduced according to this equation:

$$\frac{dX_k}{dt}=(\mu_k(S,O_2)-Y_{T,k})X_k \qquad (1.5)$$

$Y_{T,k}$	toxin production rate constant	(s^{-1})

If specie X_l is sensitive to the toxin, then its growth is reduced by a term proportional to the toxin concentration:

$$\frac{dX_l}{dt}=(\mu_l(S,O_2)-\kappa_{T,l}T)X_l \qquad (1.6)$$

$\kappa_{T,l}$	toxin kill rate constant	(s^{-1})
T	toxin concentration	$(^{mol}/_{l})$

It is assumed that the toxin is lost in that reaction (1.6), so the toxin usage is described with the following differential equation:

$$\frac{dT}{dt}=Y_{T,k}\cdot X_k-\beta_{T,l}\cdot T\cdot X_l \qquad (1.7)$$

$\beta_{T,l}$	Toxin uptake rate constant due to toxic effect on X_l	(s^{-1})

The geometry of the environment

The complex geometry of the intestine is simplified in this simulator to a single, asymmetric, tube. The tube is subdivided in axial direction and in radial direction. The radius of the tube varies along the intestine.
A typical set of parameters for the simulation is: Total intestinal length of 6 m with a varying diameter; the first 4.98 m are the small intestine with a radius of 1 cm, the next 18 cm are the "caecum" with 5 cm radius, followed by a colon of 84 cm and 3 cm radius. After various tests, it appeared that 10 radial subdivisions and 100 axial subdivisions are precise enough for the simulation. The time scale of the simulation is in the order of days for the flora as a whole. The time steps of the simulation are in the order of 20 min, since this is the fastest doubling time of the bacteria, so smaller time steps do not change the result in any significant way.

The mechanics of transport

The transport mechanics are modeled separately from the metabolism and are divided in two parts:

- *Diffusion* Transport trough diffusion applies to all volume elements. For this kind of transport, the time steps must be so small, that only diffusion between immediate neighbors needs to be considered.
- *Laminar Flow* This transport is laminar, and can be computed for every radial subdivision of the intestine independently.

The transport functions do not have to be the same for all the substances of the simulation. For instance, in the implementation of an intestinal wall [4] some elements do not move at all.
A detailed description of the transport mechanics and its equations can be found in the MIMICS Technical Report [2].

Boundary conditions

Food and oxygen are always present in the epithelium of the intestine and they can diffuse through the intestinal wall. For all other substances the wall has to be considered as sealed hermitically.

2.3 The program structure

The program structure can be described in a (C-like) pseudo language:

```
main()
{       initialize_exp();
        for (t=0;t<=tmax; t++) {
                metabolism();
                laminar_flow();
                diffusion_and_bounds();
                store_data();
                report_progress();
        }
        final_report();
}
```

Figure 2-1: program structure

initialize_exp	Read experiment conditions from file, and initialize all data structures
metabolism	Compute the metabolism interactions for every grid point
laminar_flow	Compute the laminar flow through the tube for every radial layer
diffusion_and_bounds	Compute the diffusion and boundary conditions for every grid point
store_data	Store the new data
report_progress	Report the progress after a certain number of iterations
final_report	Generate a final report.

For the *diffusion_and_bounds*, a sub loop of smaller time steps may be needed to guarantee that only diffusion between direct neighbors needs to be considered.

2.4 Results and conclusion

The current state of the MIMICS project shows that it is possible to simulate the intestinal micro flora and transport processes. Various hypotheses have already been supported by the outcomes of simulations made by the MIMICS tool. This means that the first steps have been made towards creating a realistic simulation environment.

3. Software architecture evaluation methods

When a new software tool is developed for the MIMICS project then such a tool should be evaluated.
The software architecture of a program can be defined as *the structure or structures of the system, which comprise software components, the external visible properties of those components, and the relationships among them* [6].
Analysis of software architectures is mainly carried out by the assessment of system qualities, like performance, reliability, security, etc. There exist various methods to evaluate the software architecture of a project. The next sections consist of brief overviews of four existing evaluation methods of which one or more will be chosen for the evaluation of the new tool.

3.1 SAAM – Software Architecture Analysis Method

The Software Architecture Analysis Method (SAAM) is the first documented architecture analysis method. It was originally developed for the evaluation the architectures' modifiability and functionality, but has proven to be useful to asses many other system qualities as well.

This method consists of six steps:

1. **Development of scenarios.** The system's stakeholders are asked to develop various scenarios that illustrate the kinds of activities the system should support, and collect those changes that may affect the software architecture.
2. **Description of candidate architecture.** The candidate architecture or architectures should be described in a syntactic architectural notation in such a way that it is understood by all the parties involved in the analysis. Practically, the development of scenarios and description of candidate architectures will be done in several iterations, since new scenarios and adaptations will arise.
3. **Classification and prioritization of the scenarios.** The scenarios are inspected and prioritized, and divided in direct and indirect scenarios. Direct scenarios do not require changes of the software architecture, while indirect scenarios require modification to be satisfied.
4. **Individual evaluation of the indirect scenarios.** The direct scenarios are mapped on the architecture to verify functionality. For each indirect scenario, the impact on the architecture and change costs are estimated and described.
5. **Assessing scenario interactions.** The results of the previous step are used to investigate whether a single component is affected by more than one indirect scenario, called scenario interaction. This reveals if the architecture supports an appropriate separation of concerns.
6. **Overall evaluation.** The scenarios and interactions should be ranked by importance. The result should reflect the relative importance of the qualities that the scenarios manifest.

SAAM can best be used after the functionality has been allocated to the software architecture modules.

3.2 ATAM – Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method (ATAM) is a method developed for the evaluation of software architectures in relation to quality attribute goals. The ATAM is the successor of SAAM. It got its name because it reveals how well software architecture satisfies particular quality goals and also provides insight into how those qualities interact or trade off with each other. Like SAAM, this method is scenario based.

The ATAM consists of nine steps, which are separated in four groups. The following table is directly from the creator of ATAM [6]:

Presentation

1. **Present the ATAM.** The evaluation leader describes the evaluation method to the assembled participants, tries to set their expectations, and answers questions they may have.
2. **Present business drivers.** A project spokesperson (ideally the project manager or system customer) describes what business goals are motivating the development effort and hence what will be the primary architectural drivers (for example, high availability or time to market or high security).
3. **Present architecture.** The architect will describe the architecture, focusing on how it addresses to the business drivers.

Investigation and Analysis

4. **Identify architectural approaches.** Architectural approaches are identified by the architect, but are not analyzed.
5. **Generate quality attribute utility tree.** The quality factors that comprise system "utility" (performance, availability, security, modifiability, usability, etc.) are elicited, specified down to the level of scenarios, annotated with stimuli and responses, and prioritized.
6. **Analyze architectural approaches.** Based upon the high-priority factors identified in Step 5, the architectural approaches that address those factors are elicited and analyzed (for example, an architectural approach aimed at meeting performance goals will be subjected to a performance analysis). During this step architectural risks, sensitivity points, and tradeoff points are identified.

Testing

7. **Brainstorm and prioritize scenarios.** A larger set of scenarios is elicited from the entire group of stakeholders. This set of scenarios is prioritized via a voting process involving the entire stakeholder group.
8. **Analyze architectural approaches.** This step reiterates the activities of Step 6, but using the highly ranked scenarios from Step 7. Those scenarios are considered to be test cases to confirm the analysis performed thus far. This analysis may uncover additional architectural approaches, risks, sensitivity points, and tradeoff points, which are then documented.

Reporting

9. **Present results.** Based upon the information collected in the ATAM (approaches, scenarios, attribute-specific questions, the utility tree, risks, non-risks, sensitivity points, tradeoffs) the ATAM team presents the findings to the assembled stakeholders.

ATAM is a framework for analyzing the system qualities concurrently; it does not prescribe specific methods for analyzing the individual qualities itself. Other methods like SAAM or ALMA may be used within ATAM to analyze qualities like modifiability and functionality.
ATAM can be a very comprehensive method that covers all system qualities. It can be applied when architecture designs is completed enough to allow scenario walk-troughs and the major architecture design approaches have been chosen. It is the most effective on a project with a large development team.

3.3 ARID – Active Reviews for Intermediate Designs

Active Reviews for Intermediate Designs (ARID) are a combination of the scenario-based methods as mentioned above and ADR, Active Design Reviews. The method was developed because of the need for a method that gives insight into the architecture design in an early stage of development.

ADR asks the reviewer to make use of the architectural design by assigning them review tasks that are carefully structured to avoid asking yes/no questions. Such questions can undermine a review if a reviewer gives a carelessly considered answer. An active design review demands the reviewer to utilize the design in a series of exercises that test actual understanding. ARID can be described as an ATAM/ADR hybrid and it consists of nine steps, distributed over two main phases, the rehearsal and the review phase.

Phase 1: Rehearsal
In this phase, the lead designer and the review facilitator have a meeting to prepare the exercise.

1. **Identify the reviewers.** The reviewers of the project are identified, typically the software engineers of the project.
2. **Prepare the design briefing.** The lead designer prepares a briefing to present the design in sufficient detail so that the audience could use the design.
3. **Prepare the seed scenarios.** The designer and facilitator prepare about a dozen scenarios to illustrate the concept of a scenario to the reviewers.
4. **Prepare the materials.** Make copies for presentation, scenarios, etc.

Phase 2: Review
The stakeholders are assembled and the main activities can begin.

5. **Present ARID.** The ARID method is explained to the participants
6. **Present the design.** The designer explains the design in a presentation, to investigate if the design is suitable for the problem.
7. **Brainstorm and prioritize scenarios.** Stakeholders suggest scenarios for using the design to solve problems they expect to face, like in the ATAM.
8. **Apply the scenarios.** The reviewers will be asked to start writing the (pseudo) code to solve the problem as described in the scenario. They will make extensive use of the examples that were handed out during the designer's presentation
9. **Summarize.** The participants are polled for their opinions regarding the efficacy of the review exercise, and they are thanked for their participation.

ARID can be described as an ATAM/ADR hybrid and is particularly useful in an early stage of development, during the architecture design. Like ATAM, ARID is most effective with a large development team.

3.4 ALMA – Architecture-Level Modifiability Analysis

The modifiability of a program can be defined as *the ease with which a program can be modified to changes in environment, requirement or functional specification* [7]. This important system quality can be analyzed with various evaluation methods, like mentioned in the previous sections. However, these evaluation methods can have some fundamentally different results, because of the different goals pursued by each method. Therefore, the Architecture-Level Modifiability Analysis (ALMA) [7] was developed, a generalized, adaptable method for software architecture analysis of the modifiability.
ALMA consists of five steps:

1. **Goal selection.** The aim of the software architecture analysis of modifiability has to be determined. It can be one of the following goals:
    - **Maintenance cost prediction.** Estimate the cost that is required to modify the system
    - **Risk assessment.** Reveal for what types of changes the current software architecture is inflexible.
    - **Software architecture selection.** Compare two or more software architecture candidates to find the best one.
2. **Software architecture Description.** For the software architecture description, the following information is essential:
    - The decomposition into components
    - The relationships among the components
    - The (lower-level) design and implementation of the architecture
    - The relationships to the system's environment

3. **Scenario elicitation.** Finding and selecting the set of relevant scenarios. The selection technique depends on which goal was chosen at step 1:
   - **Maintenance cost prediction**: Bottom-op approach; interview the stakeholders for the changes they anticipate.
   - **Risk assessment**. Top-down approach; focus the interview on changes that are expected to be complex
   - **Software architecture selection.** Top-down approach; only those scenarios that expose differences between the candidate architectures are of interest.
4. **Scenario evaluati on.** Evaluate the effect of the scenarios on the architecture and express the results in a way suitable for the goal of the analysis. This is divided in the following steps:
   - List the affected components
   - Locate the operations that have to be modified
   - Determine ripple effects
5. **Interpretation.** Depending on the selected goal, a conclusion has to be drawn from the results of the previous step.

ALMA can be used when the data mentioned in step 2 is available.

4. The terms of reference for the new tool

4.1 Current versions of MIMICS that should be integrated

Currently, there exist several versions of MIMICS, which should all be integrated into the new simulation tool. The four most important versions are:

- *Simple food uptake version*
  This version only considers two bacteria that consume one food element. The differential equations for these three substances are

$$F(x_{specie\_a}) = \mu_{max,a} \cdot S(x_{specie\_a}) \cdot \frac{S(x_{food})}{S(x_{food}) + K_a(x_{food})} \tag{2.1}$$

$$F(x_{specie\_b}) = \mu_{max,b} \cdot S(x_{specie\_b}) \cdot \frac{S(x_{food})}{S(x_{food}) + K_b(x_{food})} \tag{2.2}$$

$$F(x_{food}) = -V_{max,a} \cdot S(x_{specie\_a}) \cdot \frac{S(x_{food})}{S(x_{food}) + K_a(x_{food})} - V_{max,b} \cdot S(x_{specie\_b}) \cdot \frac{S(x_{food})}{S(x_{food}) + K_b(x_{food})} \tag{2.3}$$

F	Time derivative of specified element	
S	concentration of the specified element	$({}^{mol}/_{l})$
$K_{a,b}$	food uptake saturation constant	$({}^{mol}/_{l})$
$\mu_{max,a,b}$	maximum growth rate of the bacteria	(s^{-1})
$V_{max,a,b}$	maximum food uptake rate	(s^{-1})

- *Oxygen version*
  Same as the previous version, but now oxygen is included. Therefore, the differential equations (1.1) and (1.2) in chapter 2 are used to define the growth of the bacteria, the food uptake is defined with (1.3), and the oxygen usage is defined with (1.4). An extra option is to add diarrhoea related variables and functions to the program.

- *Toxin version:*
  The toxin kill-rate and usage are included in the project, by implementing differential equations (1.5), (1.6) and (1.7) in chapter 2. In this version, oxygen is not included, so the food uptake is defined like in the *Simple food uptake version.*

- *Intestinal Wall Version 31-5-2001:*
  This version almost includes the previous ones, but now the intestinal wall is added to the simulation. Bacteria can be divided in strictly luminal and wall attaching bacteria. The bacteria which are capable of attaching themselves to the intestinal wall can be in two situations, which influence the rate of growth: attached to the wall or not. When a bacterium is attached, it does not move i.e. it is not affected by the transport routines. In the model, a wall population is situated next to the last element in radial direction. More detailed description of this process and how it influences the differential equations can be found in *In Silico modeling of the human intestinal microflora* [4].

The intestinal wall version works with FORTRAN 90 modules instead of one big program, and can use multiple differential equation solvers for computing the metabolism.

On the basis of the above, the most important variation points of the current versions are:

- Different numbers of substances and computation precision.
- The presence of the toxin and/or oxygen elements in the simulation, which affects initialisation and metabolism routines.
- Multiple differential equation solvers. (Rungk4 and D02EAE routines)
- Use of FORTRAN 90 modules and multiple files
- The presence of the intestinal wall, which affects:
    1. the transport functions, wall elements do not move
    2. the metabolism reactions, wall elements have slight different metabolism reactions
    3. the boundary conditions

Additionally to the differential equations already included in the current versions, several other equations can be used to model the bacterial interactions in the gut. Various models and its equations are proposed in MIMICS Technical Report II: *Ordinary Equations for Modelling Bacterial Interactions in the Gut* [3].

4.2 Comparison of the possible evaluation methods for the new tool

For the new MIMICS project, the most important system qualities are:

- **Performance** The program should exhibit an acceptable performance. Currently, MIMICS satisfies to this condition and some of its speed it may even be sacrificed to improve the other two qualities.
- **Ease of use** There exists no graphical user interface for MIMICS yet, so it is not very user friendly and this should be improved in the next version
- **Modifiability** The most important system quality is the modifiability of the program, since the simulation environment must be able to adapt easily to different simulation tasks.

One of the evaluation methods has to be chosen to make the first design decisions. ATAM is a comprehensive method, but the new project is in a too early stage to use this method. ARID would be a good choice at this moment, but the development team of this project is too small to have a good result. ALMA could be an option, as the goal of the evaluation can be given, but it does not refer to all the system qualities that are important for the MIMICS project. Therefore, it is decided to evaluate this project with the SAAM. This method has proven to be especially valuable for analyzing modifiability, as well for the other system qualities. The SAAM results may be used within an evaluation process with ATAM in a later stadium.

4.3 Evaluation of the new MIMICS tool with SAAM

Step 1: Development of scenarios

Scenarios should illustrate the kinds of activities the system supports. In this case, a number of desired changes and additions will be listed that should be or already have been made.

Table 4-1: Possible scenarios

Scenario Number	Scenario Description
1	Add an graphical user interface to the program
2	Real-time interaction during experiment
3	Include an immune system
4	Simulation of peristaltic movements
5	Addition of a intestinal wall
6	Easy insertion of new types of metabolism
7	Addition of facultative bacteria
8	Addition of a mucus layer
9	Use different kinds of food
10	Possibility to change the metabolism and transport equations easily
11	Generating multiple progra m languages for system different architectures
12	Access to the experiments through internet

Step 2: Description of candidate architecture(s).

The candidate architectures should now be described. However, SAAM does not provide a clear method to make this description. Steps 1 and 2 have been developed in an iteration process.
The current tool appeared to be very suitable for data-parallel execution. This is because the metabolism, which is the most computation intensive process of the simulation, can be computed for every grid point concurrently. Research has proven that the best performance is obtained on memory-shared systems like the Cray SV1e supercomputer. These systems work best with FORTRAN, which is not a language that gives the possibility to use objects or add graphical user interfaces easily. In order to achieve the goal of a flexible system that covers the variation points and some of the listed scenarios, but still keeps a high performance, there are two candidate architectures:

Candidate 1
The simplest solution seems to adapt the latest version of the MIMICS tool in such a way that it is possible to import differential equations for various situations. Therefore, a new FORTRAN package should be created, from where the user can import the differential subroutines and their initialisation routines by simple statements. The user interface could then be improved by making it possible to select the simulation model from the command prompt, or by adaptation of the initialisation files.

Candidate 2
A more drastic solution is to develop a code generator to completely or partly generate the FORTRAN simulation code. With such a code generator, it will be easier to generate or import new subroutines for different situations. A graphical user interface can be added easily to such architecture. This software architecture candidate can be divided into a client side, which is the computer that will generate the code, and the server side, which will be for instance the Cray supercomputer. The client and server can be connected through a LAN or internet. Figure 4-1 provides a global overview of this architecture:

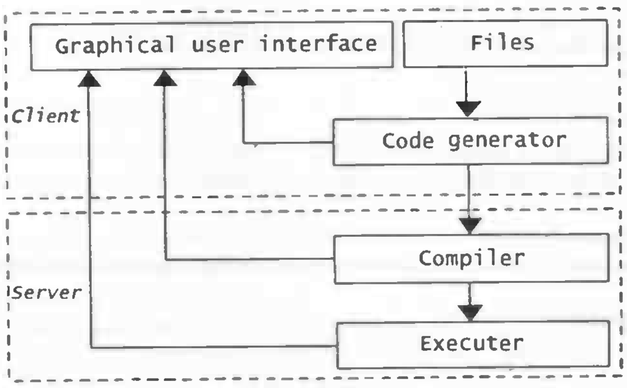


Figure 4-1: Architectural design

Client-side:	
Graphical user interface	The user gives its input though this module.
Files	This can be imported subroutines, intialisation files and tables
Code generator	This module generates the simulation code, by the input given by the files and the GUI
Server-side:	
Compiler	The (FORTRAN) compiler which creates the excutabable file.
Executer	This module executes the program and sends its output back to the user interface

Step 3: Classification and prioritization of the scenarios

The related scenarios are grouped together, ordered by priority and classified as direct or indirect scenario for each architecture candidate. Direct scenarios can be executed without any modifications to the architecture whilst indirect do require modifications. Since none of the architectures have already been created, all modifications are indirect. The next table shows the scenario groups.

Table 4-2: Scenarios ordered by priority and the effect on the candidate architectures

Priority	Scenario Number	Scenario Description	Effect on candidate #1	Effect on candidate #2
1	3,4,5,8 6,7,9,10	Possibility to include new environment circumstances, and new bacteria and food elements with their differential equations.	For every model, new subroutines, initialization-files and routines have to be included into the package.	The code generator has to be implemented in such a way that it is easy to import or generate new subroutines.
2	11	Generating the simulations code in different languages, for different system architectures and computations.	Not possible.	The code generator has to be an interface, which can be implemented for different languages
3	1,12	Change the user interface to a GUI, preferable Web-based or Java	Not possible.	A new GUI should be developed which controls the code generator in all its

				aspects.
4	12	Access trough the internet to do experiments	Not possible.	Add a server part to the program, through which a Java-applet or an active script language can execute experiments.
5	2	Real-time interaction during experiment	Threads have to be included in the simulation code, to make it p ossible to stop and continue the simulation and modify the current state.	The code generator has to include threads in the generated code, so the user can interfere during the experiment.

Step 4: Individual evaluation of the indirect scenarios

At this step, the effect of the indirect scenarios on the architectures has to be evaluated. These effects are also listed in table 4-2.

Step 5: Assessment of the scenario interactions

All the scenarios have to be mapped on the architectural representations, so the areas of scenario interaction can be assessed. This reveals if the architecture supports an appropriate separation of concerns.
Only the first and last scenario groups can be implemented with architecture candidate 1, and they affect different parts of the program.
Scenarios 1, 2 and 5 affect the code generator component of architecture candidate 2. This indicates that when the code generator is implemented, it should be split in multiple components on the basis of functionality. However, the project is in a too early stage of development to assess this more precisely.

Step 6: The overall evaluation

At this step, the right architecture candidate should be chosen, and the relative importance of the qualities should be assessed.

- **Performance** Both simulations will eventually be made in a FORTRAN program, so the performance of the candidates should not be significantly different.
- **Ease of use** The ease of use of candidate 2 will be much better, because of its graphical user interface.
- **Modifiability** The modifiability of candidate 2 will be better. On the basis of this architecture it is easier to import routines, or to change the structure of the program without rewriting the complete code.
- **Change costs** The implementation of a candidate 1 will probably be less time consuming as candidate 2.

Since the aim of this project is to create a simulation environment which can easily be adapted to new simulation tasks in the future, the decisions has been made to choose for candidate 2.

Conclusion

A totally new design for the simulation tool is needed. Therefore, the decision has been made to generate the simulation code, to get an optimal modifiability and flexibility of the tool. The code generator itself, should have a high modifiability as well, so new features can easily be implemented at a later stage.

5. Development of the new MIMICS tool: MIMICS-J

5.1 Variation points and decisions

The following design decisions have been made at the beginning of the development:

The tool will not do the simulation itself but generates the code to be compiled and executed
To make the modifiability of the simulation as high as possible and keep almost the same performance, the simulation code will be generated.
It should also be possible to generate the code in a different program language, like for instance C++ with MPI extension. With code generation, it is possible to add other programming languages in the future.

The code generator will be written in the programming language Java
The code generator obviously has to be written in an object oriented and flexible programming language, to get a high modifiability for this tool as well. The choice has been made to use the programming language Java, because this language is easy to use, frequently updated, can be executed on every operating system, and can easily be used to implement internet based applications. Therefore, the new tool is called MIMICS-J.

Initialisation files will be read during execution time, not during the compilation
In the current version of MIMICS, all initial values of the environment, transport and metabolism variables of the experiment are read by the program at the start of the simulation. Another option would be to insert the values of these variables hard-coded in the generated simulation code. The advantage of the latter method would be that all read errors can already be detected during the generation of the code. A disadvantage would be that, every time the user wants to run a simulation with a slightly different initial situation, the complete program has to be compiled again. In practice this situation occurs so often, that the choice has been made to read the initialisation files at run-time.

Metabolism interactions can also be approached per reaction
In the current implementation, the metabolism interactions are approached per substance. In the next section it will be demonstrated that if the interactions are approached per reaction, they can be described in a generalized form that gives the user the possibility to add new interactions and bacteria more easily. Of course, the import of substance interaction subroutines will still be possible.

The generated FORTRAN code will be divided in modules
To keep the generated code well organized, it will be divided by functionality in separate modules and files, like for instance a transport module, an interaction module, an environment module, etc.

5.2 Metabolism interaction with reactions

5.2.1 General description

The interactions between the simulation substances can be described by one or more differential equations. In the current implementation, the equations are drawn up for oxygen, toxin, food and every bacterium separately. The disadvantage of this approach is that these equations are difficult to implement in a user interface, because they have different structures and parameters. A generalized form of interaction would make the process easier to understand and to modify. Therefore, a new approach of defining the metabolism is introduced and this is the description of the metabolism not per substance but per reaction. In the next section it will be demonstrated that by this method a standard equation form can be used to define all interactions.

5.2.2 Reactions definition

Bio-chemical processes can be defined by the concentration of substances before the reaction, the concentration of substances after the reaction and the effect of this reaction on the environment. This approach can be described with this equation:

$$\vec{F} = \sum_j (\vec{b}_j - \vec{a}_j) r_j(\vec{S}, t) \tag{5.1}$$

t	time
$\vec{F}$	derivatives withrespect to t
$\vec{S}$	the concentration of the substances at this grid point
$\vec{a}$	stochiometric constants which define the substance-concentrations that exist before the reaction
$\vec{b}$	stochiometric constants which define the substance-concentrations that exist after the reaction
r	the rate-equations

The reactions can use multiple rate equations to define the interactions. Currently, all the necessary interactions can be modeled with two rate equations:

Mass Action Kinetics
The kinetic relationship between the substances is modeled with this simple equation:

$$r_{Mass}(\vec{S}) = \prod_{i=1}^{Subs} S_i^{n_i} \tag{5.2}$$

Monod Growth Rate
The growth of the bacteria is modeled according to the Monod Growth Model [5].

$$r_{Monod}(\vec{S}) = \prod_{i=1}^{Subs} \frac{S_i^{m_i}}{K_i + S_i^{\bar{m}_i}} \tag{5.3}$$

The total equation for a reaction will be this:

$$\vec{F} = \sum_j (\vec{b}_j - \vec{a}_j) \prod_{i=1}^{Subs} S_i^{n_{i,j}} \cdot \prod_{i=1}^{Subs} \frac{S_i^{m_{i,j}}}{K_{i,j} + S_i^{m_{i,j}}} \tag{5.4}$$

All the metabolism of the MIMICS project can be defined with (multiple) definitions of the form (5.4).

5.2.3 Converting substance interactions to reaction interactions

Case 1: Simple food uptake version

The simple food uptake (2.1, 2.2 and 2.3) can be converted to formula (5.4) easily.
This system can be defined by two reactions, the food consumption of bacterium *a* and of bacterium *b*. This system consists of three substances; two bacteria at index 1 and 2, and one food element at index 3, so the vectors have a size of three elements. For bacterium *a*, the reaction is defined as follows:

The Mass Kinetics power array $\vec{n}$ selects on which substances this reaction depends, in this case the first bacterium:

$$\vec{n} = \begin{pmatrix} 1.0 \\ 0 \\ 0 \end{pmatrix} \tag{5.5}$$

The Monod Growth Rate power array $\vec{m}$ should be defined in such a way that only the food element influences the growth:

$$\vec{m} = \begin{pmatrix} 0 \\ 0 \\ 1.0 \end{pmatrix} \tag{5.6}$$

For Monod Growth Rate array K, only the concentration of the food substrate has to be filled:

$$\vec{K} = \begin{pmatrix} 0 \\ 0 \\ K_{food} \end{pmatrix} \tag{5.7}$$

If these three vectors of the rate equations are inserted in equation 5.4, the result will be as follows:

$$\vec{F} = (\vec{b} - \vec{a}) \cdot S(x_{specie\_a}) \cdot \frac{S(x_{food})}{S(x_{food}) + K(x_{food})} \tag{5.8}$$

In vectors *b* and *a* the stochiometric constants can be defined to determine which substances exist before and after the reaction.

The food concentration will decrease at a rate of $V_{max} \cdot r(\vec{S}, t)$:

$$\vec{a} = \begin{pmatrix} 0 \\ 0 \\ V_{max} \end{pmatrix} \tag{5.9}$$

The concentration of bacterium a will increase at a rate of $\mu_{max} \cdot r(\vec{S}, t)$:

$$\vec{b} = \begin{pmatrix} \mu_{\max} \\ 0 \\ 0 \end{pmatrix} \qquad (5.10)$$

If vectors 5.9 and 5.10 are inserted in equation 5.8, the differential equation vector F is updated for all the substances which are involved in the reaction of the food consumption of bacterium a. If a second reaction is defined in the same way for bacterium b, then the simple food uptake is defined in reactions. The other versions can be converted in a similar way, as shown in the next chapters.

Case 2: Toxin version

The toxin version can also be converted to formula (5.4). The food uptake is the same as in case 1, only a fourth element is added to the system, the toxin. Assumed is that bacteria a produces toxin (1.5) to kill bacteria b (1.6). The differential equation for toxin is defined by (1.7).
These are the reactions needed for this model, and the definitions in the arrays of formula (5.4).

The toxin production:

$$F(X_{specie\_a}) = F(X_{specie\_A}) - Y_{T,k} \cdot S(X_{specie\_a}) \qquad (5.11)$$

$$F(X_{toxin}) = F(X_{toxin}) + Y_{T,k} \cdot S(X_{specie\_a}) \qquad (5.12)$$

$$\vec{a} = \begin{pmatrix} Y_{T,k} \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ Y_{T,k} \end{pmatrix} \quad \vec{n} = \begin{pmatrix} 1.0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{m} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{K} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad (5.13)$$

The toxin destruction:

$$F(X_{specie\_b}) = F(X_{specie\_b}) - \kappa_{T,k} \cdot S(X_{toxin}) \cdot S(X_{specie\_b}) \qquad (5.14)$$

$$F(X_{toxin}) = F(X_{toxin}) - \beta_{T,k} \cdot S(X_{toxin}) \cdot S(X_{specie\_b}) \qquad (5.15)$$

$$\vec{a} = \begin{pmatrix} 0 \\ \kappa_{T,k} \\ 0 \\ \beta_{T,k} \end{pmatrix} \quad \vec{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{n} = \begin{pmatrix} 0 \\ 1.0 \\ 0 \\ 1.0 \end{pmatrix} \quad \vec{m} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{K} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad (5.16)$$

The bacterium growth (in a similar way for bacterium *b*):

$$F(X_{specie\_a}) = F(X_{specie\_a}) + \mu_{\max} \cdot \frac{S(X_{food})}{S(X_{food}) + K(X_{food})} S(X_{specie\_a}) \tag{5.17}$$

$$F(X_{food}) = F(X_{food}) - V_{\max} \cdot \frac{S(X_{food})}{S(X_{food}) + K(X_{food})} S(X_{specie\_a}) \tag{5.18}$$

$$\vec{a} = \begin{pmatrix} 0 \\ 0 \\ V_{\max} \\ 0 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} \mu_{\max} \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{n} = \begin{pmatrix} 1.0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{m} = \begin{pmatrix} 0 \\ 0 \\ 1.0 \\ 0 \end{pmatrix} \quad \vec{K} = \begin{pmatrix} 0 \\ 0 \\ K_{food} \\ 0 \end{pmatrix} \tag{5.19}$$

S	concentration of the specified element (mol/l)
K	half saturation of the specified element (mol/l)
$\mu_{\max}$	maximum growth rate of the bacteria
$V_{\max}$	maximum food uptake rate
$Y_{T,k}$	toxin produce rate
$\beta_{T,l}$	rate for the loss of toxin due to its toxic effect on a bacterium
$\kappa_{T,l}$	toxin kill rate

Case 3: Oxygen version

The metabolisms of the aerobe and anaerobe bacteria are defined with formula (1.1), (1.3) and (1.4). The fourth index of the arrays is the oxygen. It is assumed that there are two bacteria: bacterium *a* is a strict aerobe and bacterium *b* is a strict anaerobe (table 2-1). Below the reactions needed for this model, and the definitions in the arrays of formula (5.4).

The growth of bacterium *a*:

$$F(x_{specie\_a}) = \mu_{o_2} \cdot S(x_{specie\_a}) \cdot \frac{S(x_{food})}{S(x_{food}) + K(x_{food})} \cdot \frac{S(x_{O_2})}{S(x_{O_2}) + K_R(x_{O_2})} \tag{5.20}$$

$$F(x_{food}) = -V_{o_2} \cdot S(x_{specie\_a}) \cdot \frac{S(x_{food})}{S(x_{food}) + K(x_{food})} \cdot \frac{S(x_{O_2})}{S(x_{O_2}) + K_R(x_{O_2})} \tag{5.21}$$

$$F(x_{o_2}) = -\beta_R \cdot S(x_{specie\_a}) \cdot \frac{S(x_{food})}{S(x_{food}) + K(x_{food})} \cdot \frac{S(x_{O_2})}{S(x_{O_2}) + K_R(x_{O_2})} \tag{5.22}$$

$$\vec{a} = \begin{pmatrix} 0 \\ 0 \\ V_{o_2} \\ \beta_R \end{pmatrix} \quad \vec{b} = \begin{pmatrix} \mu_{o_2} \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{n} = \begin{pmatrix} 1.0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{m} = \begin{pmatrix} 0 \\ 0 \\ 1.0 \\ 1.0 \end{pmatrix} \quad \vec{K} = \begin{pmatrix} 0 \\ 0 \\ K_{food} \\ K_{R,O_2} \end{pmatrix} \tag{5.23}$$

The growth of bacterium *b*:

$$F(x_{specie\_b}) = \mu_{an} \cdot S(x_{specie\_b}) \cdot \frac{S(x_{food})}{S(x_{food}) + K(x_{food})} \tag{5.24}$$

$$F(x_{food}) = F(x_{food}) - V_{an} \cdot S(x_{specie\_b}) \cdot \frac{S(x_{food})}{S(x_{food}) + K(x_{food})} \tag{5.25}$$

$$\bar{a} = \begin{pmatrix} 0 \\ 0 \\ V_{an} \\ 0 \end{pmatrix} \quad \bar{b} = \begin{pmatrix} 0 \\ \mu_{an} \\ 0 \\ 0 \end{pmatrix} \quad \bar{n} = \begin{pmatrix} 0 \\ 1.0 \\ 0 \\ 0 \end{pmatrix} \quad \overline{m} = \begin{pmatrix} 0 \\ 0 \\ 1.0 \\ 0 \end{pmatrix} \quad \overline{K} = \begin{pmatrix} 0 \\ 0 \\ K_{food} \\ 0 \end{pmatrix} \tag{5.26}$$

The inhibition of metabolism of bacterium *b* by oxygen:

$$F(x_{specie\_b}) = F(x_{specie\_b}) - \mu_{O_2} \cdot S(x_{specie\_b}) \cdot \frac{S(x_{food})}{S(x_{food}) + K(x_{food})} \cdot \frac{S(x_{O_2})}{S(x_{O_2}) + K_R(x_{O_2})} \tag{5.27}$$

$$\bar{a} = \begin{pmatrix} \mu_{O_2} \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \bar{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \bar{n} = \begin{pmatrix} 0 \\ 1.0 \\ 0 \\ 0 \end{pmatrix} \quad \overline{m} = \begin{pmatrix} 0 \\ 0 \\ 1.0 \\ 1.0 \end{pmatrix} \quad \overline{K} = \begin{pmatrix} 0 \\ 0 \\ K_{food} \\ K_{R,O_2} \end{pmatrix} \tag{5.28}$$

The oxygen destruction:

$$F(x_{specie\_b}) = F(x_{specie\_b}) - \kappa_{O_2} \cdot S(x_{specie\_b}) \cdot \frac{S(x_{O_2})}{S(x_{O_2}) + K_T(x_{O_2})} \tag{5.29}$$

$$F(x_{food}) = F(x_{food}) + Y_\kappa \cdot \kappa_{O_2} \cdot S(x_{specie\_b}) \cdot \frac{S(x_{O_2})}{S(x_{O_2}) + K_T(x_{O_2})} \tag{5.30}$$

$$F(x_{O_2}) = F(x_{O_2}) - \beta_T \cdot S(x_{specie\_b}) \cdot \frac{S(x_{O_2})}{S(x_{O_2}) + K_T(x_{O_2})} \tag{5.31}$$

$$\bar{a} = \begin{pmatrix} 0 \\ \kappa_{O_2} \\ 0 \\ \beta_{O_2} \end{pmatrix} \quad \bar{b} = \begin{pmatrix} 0 \\ 0 \\ Y_\kappa \cdot \kappa_{O_2} \\ 0 \end{pmatrix} \quad \bar{n} = \begin{pmatrix} 0 \\ 1.0 \\ 0 \\ 0 \end{pmatrix} \quad \overline{m} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1.0 \end{pmatrix} \quad \overline{K} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ K_{T,O_2} \end{pmatrix} \tag{5.32}$$

S	concentration of the specified element $\left({}^{mol}\!/\!{}_{l}\right)$
K	half saturation of the specified element $\left({}^{mol}\!/\!{}_{l}\right)$
μ_{an}	maximum growth rate of anaerobic metabolism
V_{an}	maximum specific uptake rate of the anaerobes
V_{o_2}	maximum specific uptake rate of the aerobes
β_R	maximum oxygen uptake rates due to aerobic metabolism
β_T	maximum oxygen uptake rates due to toxic effect on anaerobes or microaerophiles
K_{food}	half saturation rate food concentration$\left({}^{mol}\!/\!{}_{l}\right)$
K_{R,O_2}	half saturation respiration rate oxygen concentration$\left({}^{mol}\!/\!{}_{l}\right)$
K_{T,O_2}	half saturation kill rate oxygen concentration$\left({}^{mol}\!/\!{}_{l}\right)$
κ_{O_2}	maximum oxygen kill rate
Y_κ	yield of substrate per unit of bacteria killed by oxygen

5.3 Constraints of MIMICS-J and possible other applications

The MIMICS-J main tool will be useable not only for the MIMICS project but also for other comparable situations. Therefore, it has to be defined which situations can be simulated by the new tool, and which fall beyond its scope.

The tool covers only situations, in which the organisms are present in such a large quantity that it is impossible to simulate it with individual entities.

If it was tried to get a betterinsight in the interaction of, for instance, a flock of sheep, then each individual sheep could be simulated by a single process. But in situations like the microbial ecosystem in the human intestines, there are too many entities (10^{14}), so this problem has to be approached with concentrations. Hence, the tool will not be useful for small population experiments.

The tool covers only situations in which the interaction between the organisms and its environment can be described as bio chemical reactions that can be defined as a list of differential equations. These equations define how the organisms affect each other and in what way.

So this excludes any kind of process experiment that can not be described in such a way.

The simulation can be divided in interaction processes and transport processes.

The tool will make a clear distinction between these processes, so if the situation cannot be divided in that way, then it falls beyond the scope of the possible application of the new tool.

Taking into account these constraints of the simulation tool, the conclusion must be that the concept of the tool might also be applicable for simulation of other processes or situations than only the human intestine. Subject to further examination one could think of:

- The ecosystem in a river: concentrations of bacteria and species
- Climate change models: greenhouse effects; concentrations of carbon-dioxide etc.
- Fermentation processes in biochemical industry

6. Implementation of MIMICS-J

6.1 The client-server model

The MIMICS-J tool is divided in a client and a server part (fig 6-1).

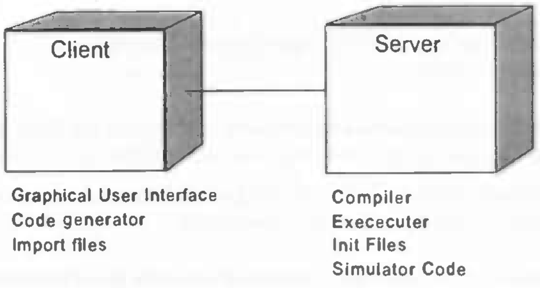


Figure 6-1: Client-server model of MIMICS-J

The creation and execution of a simulation will be done in five steps:

1. On the client, the user can give its input by the graphical user interface (see also fig 4-1). He/she can define which routines should be imported for which situations and components. Also the interactions can be defined, either by imported substance differential equations, or by manual defined reactions.
2. The code will be generated. It will be divided over different files and modules by functionality. Also the initialization files of the components are created.
3. The simulation code and the initialization files are sent to the server. In the current imp lementation of MIMICS-J, the files are sent to the Cray SV1e supercomputer by FTP.
4. The simulation code has to be compiled. Currently, this is done manually by a telnet-connection to the Cray.
5. The simulation code is executed to do the actual simulation.

6.2 Overall program structure

The overall view of the MIMIC-J tool looks like this:

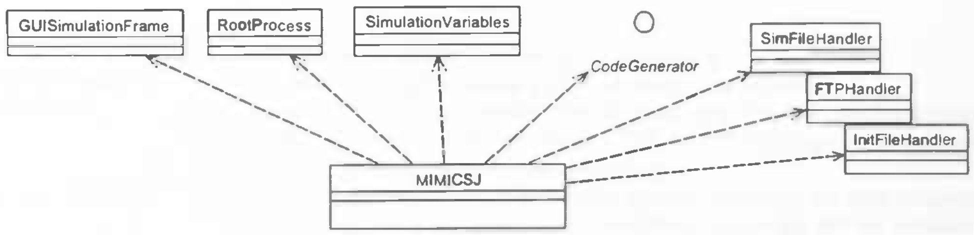


Figure 6-2: Overall structure of MIMIC-J tool

The main-file of the tool is the connection between all these components.

GUISimulationFrame	This is the graphical user interface of the tool; it will be explained in chapter 7.
RootProcess	The root process holds the simulation processes of the simulation, like the transport processes, the interaction processes, the initialization processes.
SimulationVariables	This class holds the variables of the simulation. It is separated in groups like for instance transport variables, environment variables, etc.

CodeGenerator	The code generator is an interface, which implementations will generate the actual simulation code files.
SimFileHandler	This component handles the opening and saving of project files
FTPHandler	This component handles the FTP connection to the server, the Cray.
InitFileHandler	This component generates and creates the initialization files.

6.3 The simulation processes and subroutines

In the MIMICS-J project, the simulations are divided in processes. A process is a component that can run independently from the other components, and can be distinguished because of its functionality. Practically, this will be a subroutine, or a group of subroutines to compute a particular part of the simulation. The default process-class is *SimProcess* and it has the following attributes:

name	the name of this process
description	the description of this process
localVars	the local variables needed for this process (section 6.4)
subroutines	the subroutines of this process
code	the code to be generated of this process (section 6.5)
loop	this defines if the process needs a special sub-loop

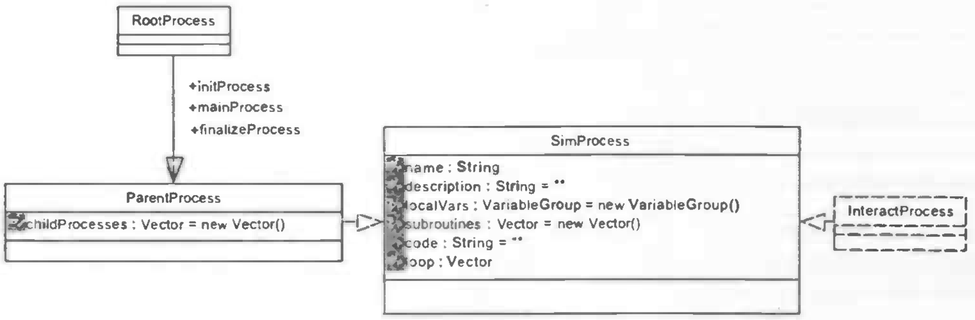

RootProcess
+initProcess
+mainProcess
+finalizeProcess
ParentProcess
childProcesses : Vector = new Vector()
SimProcess
name : String
description : String = ""
localVars : VariableGroup = new VariableGroup()
subroutines : Vector = new Vector()
code : String = ""
loop : Vector
InteractProcess

Figure 6-3: Processes

There are a couple of inherited classes of the *SimProcess* class:

ParentProcess	A parent process is a process that holds and manages sub processes.
RootProcess	The root process is a specialization of the parent process, and holds all the processes of the entire simulation. It holds three child processes, which are of the type *ParentProcess* as well: the initialization process, the main process and the finalize process. The simulator will first initialize with the child processes of *initProcess*, then iterate trough all the child processes of *mainProcess*, and finalize the simulator with *finalizeProcess*
InteractProcess	At the current stage the only real specialization of the default process is *InteractProcess*. This process computes the interaction between the substances of the simulation. Its subroutines, variables and initialization are not imported or manually created (see also section 6.6), but they can be generated automatically, like for instance the differential equation routine *DifEq* (fig 6-4).

If a project is started from scratch, a root process is created with an empty *InitProcess* and *FinalizeProcess*, and a *MainProcess* which holds *InteractProcess* as its only child.

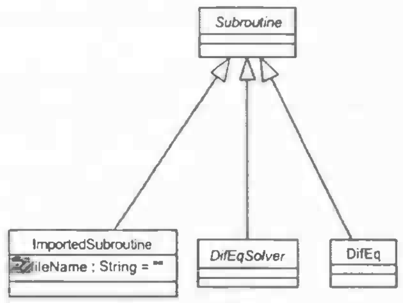


Figure 6-4: Subroutines

6.4 The simulation variables

The simulation variables are structured as showed in fig 6-5.

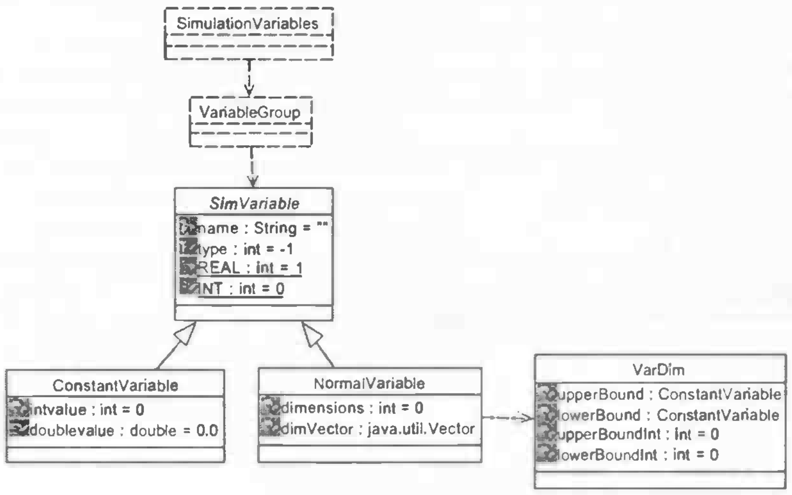


Figure 6-5: Variables

SimulationVariables	This class holds all the variables of the simulation
VariableGroup	This class holds and manages groups of simulation variables.
SimVariable	This abstract class represents a simulation variable. *SimVariable* has a *type* attribute that can be assigned with two values; *REAL* or *INT*. This represents the type of values the variable can hold (reals or integers).
ConstantVariable	This class implements the *SimVariable* class. It represents the kind of variable that is set at the beginning of the simulation and does not change, like for instance the number of substances. This variable is always 0 dimensional; it can hold only one value, which can be an integer or a real.
NormalVariable	This class also implements *SimVariable,* and represents the kind of variable which value is change during the simulation, so it has no value. This variable can consist of multiple dimensions; therefore a vector is filled with objects of the type *VarDim*

VarDim	This class represents a dimension of a standard variable. A variable dimension has an upper and a lower limit. Both limits are specified with a constant variable and an extra integer. This extra integer is necessary to correct the size of an array for boundary conditions, like for instance the main grid. A standard variable *temp* with one dimension will be like declared like this: `temp[lowerBound + lowerBoundInt, upperBound + upperBoundInt]`

With this structure practically all necessary variables can be defined. If a project is started from scratch, the default variable-groups and variables of table 6-1 are automatically created.

Table 6-1: Default variables and groups

Group	Variable	Type	Kind	Description
GlobalVars				The global variables
	Substances	int	constant	This constant holds the number of species of this simulation
	Time	real	standard	This variable holds the real current time of the simulation
	TimeInit	real	constant	Initialization value of the Time
	TimeStep	real	constant	This variable holds the time-steps that are taken at every iteration.
	RadialMax	int	constant	Radial subdivision maximal value
	RadialMin	int	constant	Radial subdivision minimal value
	AxialMax	int	constant	Axial subdivision maximal value
	AxialMin	int	constant	Axial subdivision minimal value
	Iter	int	standard	Variable for iteration through the main process
	MaxIterator	int	constant	Iterations maximal value
	MinIterator	int	constant	Iterations minimal value
EnvirVars				The environment variables
	MainGrid	real	standard	This multidimensional variable represents the whole simulation grid, in radial, axial and substantial direction
MetabVars				This group has to exist, so the interact process can add its automatically generated variables to it.

This table can become larger, when more processes are explicitly implemented. Of course when a project like MIMICS is created with the MIMICS-J tool, a lot of extra variables are needed. For instance, the environment structure and transport variables are not declared yet.

6.5 The code generator

The code generator of fig 6-1 is an interface. At this moment, only a FORTRAN 90 implementation of this interface has been created, but this way it is possible to generate simulation code in other program languages in the future (fig 6-6).

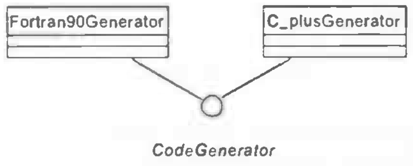


Figure 6-6: The code generator

The code generator has several functions that can be grouped according table 6-2:

Table 6-2: Code generator function groups

Function type	Examples
Statement functions	creation of iterations, program headers, subroutine headers
Variable functions	converting the MIMICS-J variables to the new language, creating variable modules and files, make variable declarations, parsing arguments or dimensions
File functions	create and manage the simulation code files
Math functions	add, multiply, subtract and power operations
Logic functions	bigger, smaller or equals statements
Subroutine functions	create subroutine headers, call subroutines, or import subroutines

The FORTRAN implementation of this interface will also have language specific functions like parsing the string lengths and generating comments. When the user demands to generate the simulation code, the code generator takes the following steps:

1. The variable-groups are parsed. Every variable group gets its own module and file. The code generator starts with *GlobalVars* (table 6-1), since important array sizes are defined in this, which may be used in other groups.
2. The root-process is passed to the generator. First *mainProcess* is parsed (fig 6-3). All its child-processes are processed one by one in the right order. In some cases extra initialization subroutines has to be added to *initProcess*. Every process gets its own module and file to keep the generated code well-ordered.
3. Finally *initProcess* and *finalizeProcess* are passed to the code generator.

6.6 The interact process

The *InteractProcess* class is a specialization of *SimProcess*. This class simulates the interaction between the substances. The process consists of two parts (6-6):

1. The Differential equation
2. The Differential equation solver

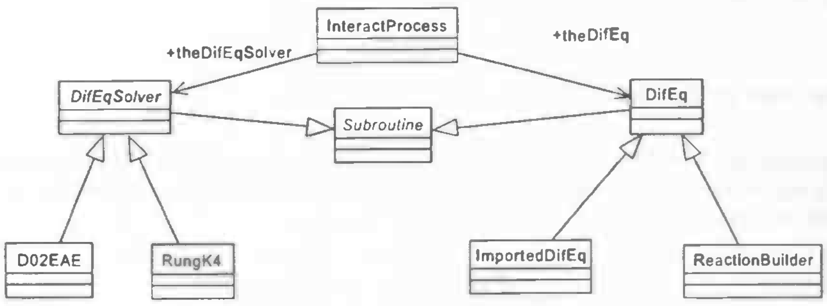


Figure 6-6: Interact process

The differential equation solver is a routine that will be imported in most situations. At the current stage, mainly the RungK4 solver [1] is used, but using other solvers, such as the D02EAE routine from the FORTRAN NAG library, will also be possible.

All interactions between the substances are defined in one subroutine (*DifEq)* that holds all the differential equations. This subroutine needs three arguments (table 6-3).

Table 6-3: Differential equation arguments

argument	Description	Type
t	time	input
$\bar{S}$	the concentration of the substances at this grid point	input
$\bar{F}$	derivatives with respect to t	output

This subroutine is passed to the differential equation solver, to compute the new concentrations at the current grid point for the next time-step.

There are two forms of differential equations of the metabolism in this project:

1. Interaction per substance: Every substance is represented by a (complex) differential equation. In this form of interaction the differential equations routine, and its initialization routines and files, will be imported.
2. Interaction per reaction: Every reaction is represented by a differential equation. The differential equations subroutine, the necessary variables and the initialization routines and files will be generated by the code-generator through user input by the *ReactionBuilder* class.

6.7 The reaction builder

The *ReactionBuilder* class generates the differential equation subroutine (table 6-3) by user input. Every reaction can be described with equation (5.1). The class consist of two parts, the *Reaction* class and the *RateEquation* class.

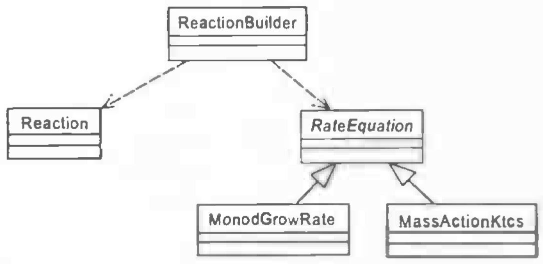


Figure 6-7: Reaction builder

At the current stage, the project implements two rate equations (fig. 6-7); the Mass Action Kinetics (5.2) and the Monod Growth Rate (5.3). For every reaction that is created by the user, the reaction-builder creates input arrays, which sizes are as large as the amount of substances in the simulation.

Table 6-4: Reaction input arrays

Array	Description
b_minus_a	conc. after reaction minus conc. before (5.1)
MassActionKtcs_n	power *n* array of (5.2)
MonodGrowRate_n	power *n* array of (5.3)
MonodGrowRate_k	*k* array of (5.3)

These arrays are stored in the *MetabVars* variable group of table 6-1. Note that only one array has to be used to store *(b-a)* of equation 5.1. The *Reaction* class holds all the values of the input-arrays which are entered by the users.

When the user demands to generate the simulation code, the ReactionBuilder class takes three steps:

1. The *DifEq* subroutine is filled with the operations according to formula 5.4, by calling the *makeEquation* functions of the rate-equations. These functions generate the appropriate code by using the *Math* and *Logic* functions of the code generator (table 6-2).
2. The *Reaction* classes are passed to the *IniFileHandler* (fig. 6-2) which generates a text file with all the values of the reaction arrays
3. A subroutine is generated to initialize all the arrays that were added to the *MetabVars* variable-group. This subroutine is added to the first child-process of *initProcess* (fig 6-3).

7. Building MIMICS in MIMICS-J

7.1 General description

The MIMICS-J tool has been implemented. The next step is to verify the functionality of the tool by implementing a MIMICS simulation. Therefore, it will be demonstrated how the food-uptake version (chapter 5.2.3, Case 1) can be generated with the MIMICS-J tool. An extensive description of all the transport and environment variables and processes of MIMICS can be found in the MIMICS Technical Report [2]. The other MIMICS versions can be simulated in a similar way, by adapting the reactions, and transport functions of the simulation.

7.2 Variables

The MIMICS food-uptake version needs extra variables to define its environment. The following table shows all the variables that should be created in the variable panel (See also Appendix A, fig. A-1 – A-4), including the default ones.

Table 7-1: Variables and groups

Group	Variable	Type	Kind	Val	Description
GlobalVars					The global variables
	Substances	int	constant	4	The number of species of this simulation
	Time	real	standard		The current time of the simulation
	TimeInit	real	constant	0.0	Initialization value of the Time
	TimeStep	real	constant	720.0	The time-steps that are taken at every iteration.
	RadialMax	int	constant	10	Radial subdivision maximal value
	RadialMin	int	constant	1	Radial subdivision minimal value
	AxialMax	int	constant	100	Axial subdivision maximal value
	AxialMin	int	constant	1	Axial subdivision minimal value
	Iter	int	standard		Variable for iteration through the main process
	MaxIterator	int	constant	1000	Iterations maximal value
	MinIterator	int	constant	1	Iterations minimal value
	FineStep	int	constant	10	Finer step to ensure that only the diffusion between direct neighbors has to be considered
	FineStepStart	int	constant	1	Start of the fine step loop
	Length	real	constant	6.0	Total length of the intestine
EnvirVars					The environment variables
	MainGrid	real	standard		This multidimensional variable represents the whole simulation grid, in radial, axial and substantial direction
	Volume	real	standard		Volume of the elements, in axial direction
	Radius	real	standard		Local radius of the intestine, in axial direction
	Aaxial	real	standard		Axial contact area
	Arad	real	standard		Radial contact area
MetabVars					Metabolism variables
	FoodIndex	int	constant	3	The index of the food element in this simulation
	NrOfSpecies	int	constant	2	The number of species in the simulation

TransVars				Variables for the transport processes
	FlConc	real	standard	Concentration of bacteria and food
	Imax	real	standard	Maximum flow rate
	Imin	real	standard	Minimum flow rate
	Wjp	real	standard	Weights pre-computation
	Wim	real	standard	Weights pre-computation
	Wip	real	standard	Weights pre-computation
	Wjm	real	standard	Weights pre-computation
	Delta	real	standard	Weights pre-computation

7.3 Processes & Imported subroutines

Most of the subroutines used in the MIMICS simulation in MIMICS-J, are modified parts of the original FORTRAN code. The safest way to import a subroutine is to rewrite it in such a way, that all not local variables used in it are passed by the arguments. The user can also refer hard-coded to the defined variables, but this can cause unexpected results and errors if a variable is ever changed or deleted. The following processes and subroutines should be created in the process (fig A-5) and subroutine panel (fig A-7 – A-11).

Table 7-2: Processes & subroutines

Parent-process	Process	Subroutine	Description
Initialize	Initialize	EnvirIni	Initializes the environment, except the grid
		TransIni	Initializes the transport variables
		GridIni	Initializes the main-grid of the simulation
	DivideWgt	DifWgt	This subroutine pre-computes some of the transport variables, to improve the performance
Mainprocess	InteractProcess	-	The interact process
	LaminarFlow	LFLOW	Computes the laminar flow through the intestine
	Diffusion	Bounds	Computes the boundary conditions
		Diffus	Computes the diffusion between the elements.
	StoreAndReport	StoreData	Store the new computed data
		ReportProg	Report the progress of the simulation on the command line
Finalize	FinalizeSim	FinalReport	Generates a final report of the simulation

For the diffusion process, the loop (fig. A-6) has to be enabled from *FineStepStart* through *FineStep*, to ensure that only the diffusion between direct neighbors has to be considered.

7.4 Interaction specification

Firstly, the *Per Reaction*-button (fig. A-12) has to be selected. In the reaction specification (fig A-13), the two reactions (see Chapter 5.2.3, Case 1) should be created, by filling in the following tables:

Table 7-3: **Reactions**

	Specie 1			Specie 2		
	Subs 1	Subs 2	Subs 3	Subs 1	Subs 2	Subs 3
Conc before (a)	0.0	0.0	2.0E-4	0.0	0.0	2.0E-4
Conc after (b)	1.0E-4	0.0	0.0	0.0	1.0E-4	0.0
MassActionKtcs_n	1.0	0.0	0.0	0.0	1.0	0.0
MonodGrowRate_n	0.0	0.0	1.0E-4	0.0	0.0	1.0E-4
MonodGrowRate_ks	0.0	0.0	0.02	0.0	0.0	0.02

As differential equation solver, the RungK4 solver should be selected (fig A-12).

8. Analysis of MIMICS-J

8.1 Results and performance

If the original MIMICS tool is rebuilt with MIMICS-J, then the results of the simulation are the same. Furthermore, if the food uptake version is created with reactions, then the results are still the same as the original per substance version, which proves the functionality of the use of reactions. If the toxin-version is created with the new tool, then a slight precision loss occurs, in the order of 10^{-7}. Furthermore, if the number of iterations is set to a too large value (more than 2000), then the current differential equation solver seems to have more problems to solve the equation. Further analysis is needed to explain this problem.

To test the performance, the original MIMICS simple food-uptake and toxin-version were created with MIMICS-J. The differential equation was approached per substance, which means that the original differential equation routine could be imported. So e xactly the same program was created, with the same subroutines and computations, with the difference that the subroutines and variables were divided over different modules and files.
It appeared that this caused a significant performance loss. Further analysis revealed that this was because of two reasons:

1. The use of modules and extra subroutines caused a significant overhead.
2. The original MIMICS tool uses special FORTRAN compiler-optimize functions in the code in order to increase the performance. This is not implemented (yet) in the code generator, so processes that generate their code, like the interact-process, do not have this optimize settings. This makes it difficult for the compiler to determine which parts can be computed parallel.

The MIMICS-J program was further tested by creating the old MIMICS tool with the standard parameters as described in the previous chapter, run on 8 CPU's of the Cray. The results of four variations are shown in table 8-1. The first column is the performance with 100 iterations in seconds; the second column with 1000 iterations and the third column shows the performance after manually insertion of compiler-optimize statements.

Table 8-1: Performance

# of iterations	**100**	**1000**	**1000 opt**
Toxic version imported	7,1	60,2	40,6
Toxic version with reactions	14,7	250,6	200,7
Food uptake imported	6,4	74,14	42,3
Food uptake with reactions	10,2	194,4	90,8

It appeared that approaching the interactions per reaction is more computational intensive than approaching it per substance. This is due to the fact that more program statements are needed to describe the reactions. The above table is not fully representative for the performance. Variation in number of used CPU's on the Cray may also influence the results. Especially when the compiler-optimize statements are used, the performance can be improved by using more processors.

8.2 Suggestions and adaptations for the future

The program is still far from complete. The following scenarios are possible further adaptations of the MIMICS-J tool, ordered by priority.

1. Increase the performance by compiler optimize settings

The performance of simulations that were generated by the MIMIC-J tool can be improved by generating compiler-optimize-statements in the code. This way, the computations can be better divided over the processors of the Cray.

2. Implementation of the transport processes

At the current stage, only the interaction process of MIMICS-J generates its own code by user input. The other the subroutines are imported, and the variables manually created. This should change in the future. The transport processes, for example, can be approached as a matrix vector system, with in the general case Navier-Stokes equations [10]. This can be implemented in such a way that the user inputs the initial values and the transport functions. The simulation-code, initialization- files and subroutines, and the transport variables can be automatically generated as in the interactions per reaction.

3. Generation the initialization of all constant variables

The constant variables of the program are now hard coded, to give the opportunity to modify them easily in the user-interface. In the future, the constant values should be stored in an initialization file. The initialization-file and its initialization-subroutines should be generated automatically. In this way, the change of an array size would only result into regenerating the initialization file, instead of a new code generation and compilation.

4. Generation of all the initialization files

Currently, the initialization-file of for instance the simulation grid has to be created with a text -editor or by a separate initialization tool. In the future, initialization tools should be integrated in MIMICS-J, with various default settings for typical begin-situations.

5. Generation of a make-file for compilation

Currently, the simulation-code has to be compiled by making a telnet connection to the Cray, and entering the right compilation command. It would be easier if the compilation-command was stored in an automatically generated make-file, which is sent to the Cray by FTP as well.

6. Implementing an SSH or telnet connection

At this moment, the telnet or SSH connection has to be made with an external program. In the future, the user should have the possibility to just push the *make*-button, and the code will compiled at the Cray or another server automatically. The output of this compilation can be sent to the output panel. The actual execution of the simulation can be done in a similar way.

7. Gathering and presenting the output of the simulation

The output and the result of the simulation are now shown at the command prompt, and store in output files. In the future, the progress of the simulation should be sent back to the client and graphically represented, and the result can be presented in graphics and charts. This way, the user can understand the results of the simulation much easier and faster.

8. Drastic improvement of the graphical user interface

The current interface is far from ideal, and should be improved in the future. If all the processes generate their own code and initialization files, maybe a block-structure can be used, to add new processes and create connections between them.

9. Implementation of other program languages and cross language implementations

Some computer systems work better with other program languages. For instance a C++-code generator should be implemented, with MPI- extensions, to work on clusters. In some cases, it may even be useful to generate cross program-language simulations to get the optimal performance, or to use FORTRAN 90 equation solvers in a C++ simulation.

10. Real-time interaction during the simulation

If a large simulation is executed, then the user may want to test what happens if a high quantity of bacteria invades the intestine immediately. This requires that the code is generated with threats, so the simulation listens to user input while it is running.

8.3 Modifiability

In order to be able to implement the adaptations as mentioned above easily, the modifiability of the MIMICS-J tool should be good as well. The new software architecture is structured in such a way, that it is possible to implement the modifications without changing the whole design.

To implement scenario 1, the FORTRAN code-generator should have extra functions to manually insert optimizing code in modules.

The current interaction-process proves that it is possible to make a specialization of the simulation-process, which creates its own initialization files and routines, so scenarios 2, 3 and 4 can also be implemented.

Scenario 5 requires an extra class for handling the make-files, just like the existing classes for handling the project- and initialization-files.

To make a SSH- or telnet-connection as in scenario 6, a similar class like the existing FTP-handling class should be created.

The output of scenario 7 can be gathered through the SSH-connection. A new graphical user interface can implement a graphic- and chart panel, and the block-structured input as mentioned in scenario 8.

Other languages can be implemented by making a new realization of the code-generator interface. If the different program language modules are connected in the right way, scenario 9 can be implemented.

Scenario 10 requires an adaptation of the code-generator interface. Depending on the possibilities of the program language, threads should be inserted in the simulator code.

9. Conclusion

Currently, there are not many simulation programs which work with such large quantities of species and substances as the tools needed for the MIMICS project. As mentioned in the previous chapter, the MIMICS-J tool could still be further improved. Nevertheless with the development of this new integrated tool a new small step has been taken towards creating a realistic simulation environment for the human intestine and its biochemical processes. Therefore, the development of MIMICS-J could make a contribution to the research into medicines for diseases such cholera and other forms of diarrhoea. Furthermore, when finally more realistic simulation environments are created, scientist will not have to resort to laboratory experiments on animals.

The application of the system architecture analysis tools was effective to assess the sensitivities of the new system architecture. However, these analyses would have been more effective at a later stage of development of the MIMICS-J tool and especially when this development would have been carried out by a large team.

When further developed, the MIMICS-J tool can be useful as a teaching tool, which can be used by researchers that are less specialized in programming, for experiments with different models and situations.

Finally, the principles of the model and approach used for MIMICS-J may be useful for application in other scientific fields within the constraints as described in section 5.3.

10. References

[1] M.H.F. Wilkinson. Model intestinal microflora in computer simulation: a simulation and modelling package for host-microflora interactions. *IEEE Trans. Biomed. Eng.* (2002) 49:1077-1085.

[2] M.H.F. Wilkinson. MIMICS Technical Report: *MIMICS Cellular Automaton Progam Design and Performance Testing*.

[3] M.H.F. Wilkinson. MIMICS Technical Report II: *Ordinary Equations for Modelling Bacterial Interactions in the Gut*.

[4] D. J. Kamerman and M.H.F. Wilkinson. In Silico modelling of the human intestinal microflora. In *Proc. Int. Conf. Computational Science 2002,* volume 2329 of *Lecture Notes in Computational Science*, pages 117-126, Amsterdam, The Nethedands, April 21-24 2002

[5] A.L. Koch, "The Monod model and its alternatives" in *Mathematical Modelling in Microbial Ecology*, A.L. Koch, J.A. Robinson and G.A. Miliken, Eds., pp 62-93/ Chapman & Hall, New York, 1998.

[6] P. Clements, R. Kazman and M. Klein. Evaluating Software Architectures. Addison Wesley 2002

[7] N. Lassing, P.O. Bengtsson, J.C. van Vliet and J. Bosch, Experiences with ALMA: Architecture-Level Modifiability Analysis, *Journal of Systems and Software*, 61 (1), pp. 47-57, 2002

[8] Sun Java 2 Platform, Standard Edition, http://java.sun.com/j2se/

[9] Java FTP client, http://jvftp.sourceforge.net/

[10] Navier-Stokes equations, http://www.eng.vt.edu/fluids/msc/ns/nsintro.htm

Appendix A: User manual of MIMICS-J

A.1 General description

The MIMICS-J tool has been built with the software development kit of Java 2 Platform Standard Edition (J2SE); version 1.4.1 [8]. The main class of the program, *MIMICSJ.class* can run with a JAVA virtual machine. To make a FTP-connection to the Cray, an extra FTP-package [9] needs to be in included the $CLASSPATH environment variable. The user interface has been developed to demonstrate the flexibility of the program, and should be improved at a later stage of the development.

The program uses four subdirectories of the directory where it is located, to read and save its files:

./PROJECTS	The project-files are read and saved in this directory. There is a file called *default* which should not be altered. The program reads all default settings from this file wh en it is started.
./IMPORT	The subroutines to be imported, like the differential equation solver or the transport functions, are stored in this directory.
./INITFILES	The initialization files has to be located here. Also the reaction-builder stores its re actions initialization-files here.
./FORTRAN	This is the directory where the generated simulation code is stored.

A.2 The menu-bar

The menu bar is organized as in the following table:

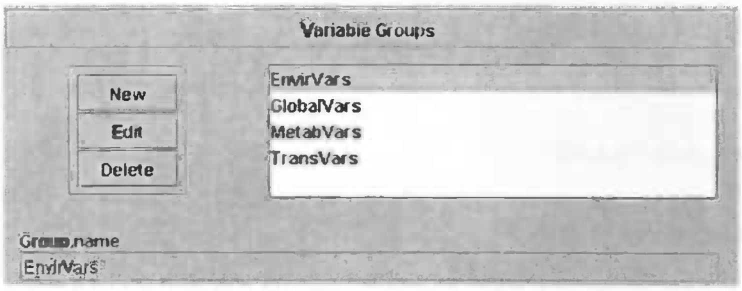
Table A.1: The menu-bar functions

Menu	Sub-menu	Description
File	New s imulation	Start a new simulation form scratch (loads the *default* file)
	Open...	Open a stored simulation, file -selector appears to choose a file
	Save	Save the current simulation
	Save as	Save the simulation with another name, file -selector appears
	Exit	Exit the simulation
Edit		No functionality is allocated to this sub-menu yet
Actions	Generate → Code & Init	Generate the FORTRAN simulation code and the initialization files
	Generate → Only Init	Generate only the new initialization-files (this case the reactions)
	Ftp → Code & Init	Sends the simulation code and the initialization files to the Cray through FTP
	Ftp → Only Init	Sends only the initialization files to the Cray
	Make/Run	No functionality is allocated to these items yet. For building and running the simulation directly on the Cray in the future.
View	Output	Switch to the Output panel. In this panel the user can read all the actions and errors made so far.
	Variables	Switch to the Variables panel
	Processes	Switch to the Processes panel

Some of the items are to be implemented in a later stage

A.3 The variable-panel

In this panel, the user can add, modify or delete variables and variable-groups. The variable-group panel (fig A-1) is located at the left. With the *New*-button, a new group can be created. With the *Edit*-button, the name and description of an existing group can be modified. When finished, the *Finish-* or *Cancel*-button at the bottom of the frame should be pushed (not shown in picture). With the *Delete*-button, the user can remove the entire variable-group with all its variables.

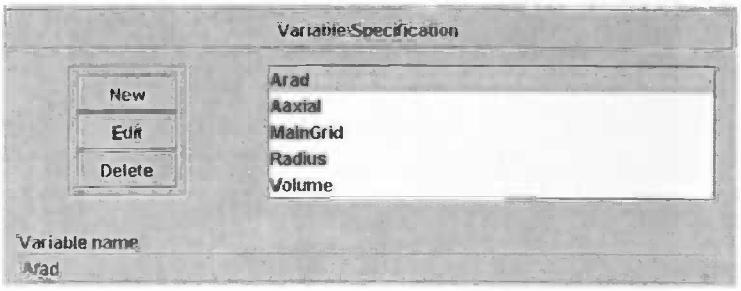


EnvirVars
GlobalVars
MetabVars
TransVars

Group name
EnvirVars

Figure A-1: Variable-groups

The variables-panel (fig A-2) is located at the right. When a variable-group is selected, all the variables of this group are loaded in the variable-list. The variables can be edited in the same way as the variable-groups.

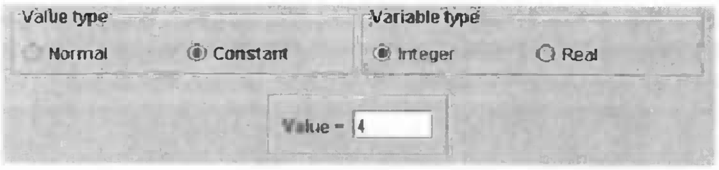


Arad
Aaxial
MainGrid
Radius
Volume

Variable name
Arad

Figure A-2: Variables

When a variable is created or edited, the value type can be selected by the radio buttons. When the *Constant*-button is selected (fig A-3), the constant-panel appears. The user can fill in the integer or real value for this variable, depending on which variable-type is selected.



Value type: Normal [✓ Constant]
Variable type: [✓ Integer] Real
Value = 4

Figure A-3: Constant variable

When the *Normal-button* is selected, the normal-panel appears (fig A-4). The user first has to select how many dimensions this variable should have, and then select the upper- and lower-bounds for every dimension. Note that only constant-variables of the integer-type and located in the *GlobalVars* variable-group, are present in the bounds selection-box.

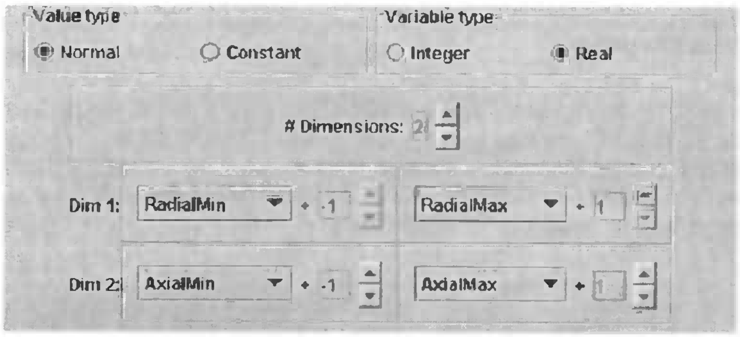


Figure A-4: Normal variable

A.4 The processes-panel

A.4.1 Process specification

In this panel, the user can create, modify or delete processes. The process specification (fig. A-5) is located on the top-left of the panel. On the top, the user can select with radio-buttons, if he wants to edit the process in the initialization, the main-loop or in the finalization of the simulation. Similar to manage variables, the user can create, modify or delete a process. With the *move up* and *move-down* buttons, the order of the processes can be changed.

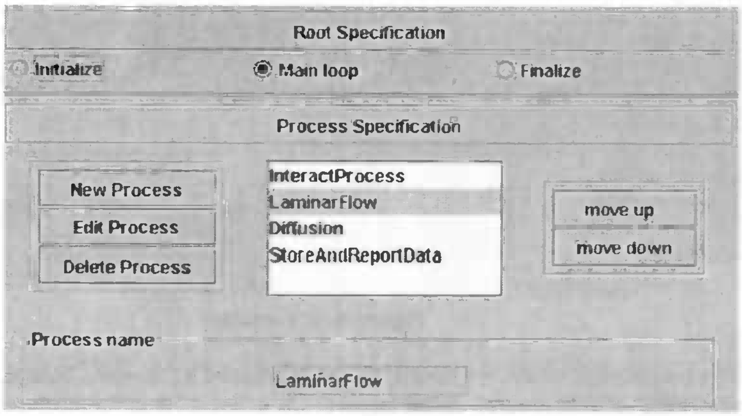


Figure A-5: Processes specification

The loop selection (fig. A-6) is located at the bottom-left of the panel. If the *Loop through this process*-button is selected, the process will iterate through the process as selected in the *start* and *end* variable selection-box.

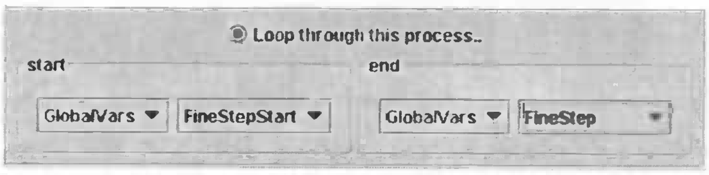


Figure A-6: Loop selection

A.4.2 Subroutine specification

When a normal process is selected in the processes specification, the subroutine specification (fig. A-7) appears on the top-right of the panel. In this panel the user can import new subroutines for this process. With the browse (...)-button, the subroutine-file can be selected. The *subroutine name* field should be the same as the name of the subroutine to be imported. With the *Edit*-button the user can make changes to the subroutine directly. The user has now to select how much parameters this subroutines uses by the arrows under the *#Params*-label.

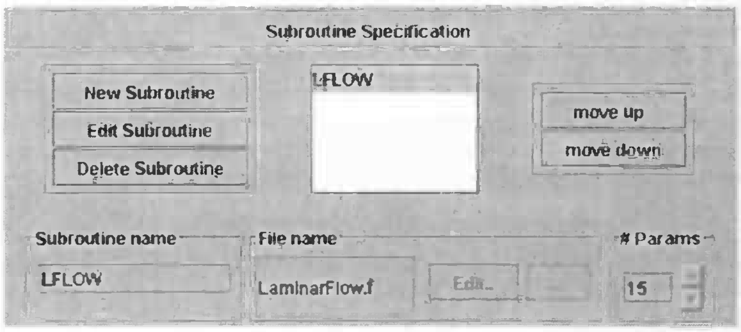


Figure A-7: Subroutine specification

Now the parameters have to be selected. There are four kinds of parameters. The first one is the substance-range parameter (fig. A -8). With this parameter, the user can select which substances are affected by this subroutine. This can be useful in case of transport functions which should not be applied to all substances. Only the first parameter of the subroutine can be a substance range.

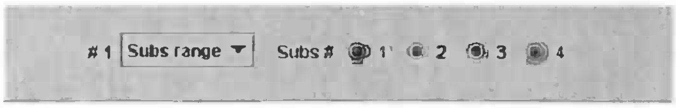


Figure A-8: Substance range parameter

When the subroutine needs a variable as its parameter, is can be selected in the variable parameter panel (fig. A-9).

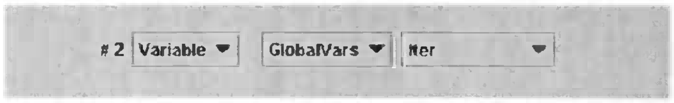


Figure A-9: Variable parameter

An initialization-subroutine may need a file as its parameter. Therefore, a file parameter can be selected (fig. A-10). With the browse (...)-button, the user can choose an initialization-file, and with the *Edit*-button the user can make changes to it.

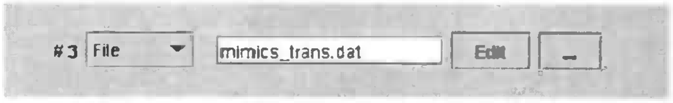


Figure A-10: file parameter

In some cases, the subroutine needs a string as its input, like for instance a subroutine name or output-file name. Therefore, the string-parameter can be selected (fig A-11).

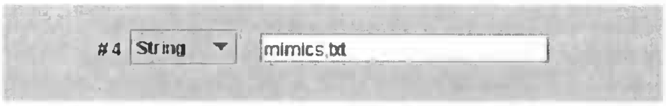


Figure A-11: String parameter

A.4.3 Interaction specification

In the process-specification panel (fig. A-5), also the *InteractProcess* can be selected. In that case, the interaction specification (fig. A-12) appears on the right of the panel. On the top, the user can select which differential equation solver the simulation has to use. At this moment, the user can choose between the RungK4-solver [1] and a dummy-solver, which disables the interact process.

Now the type of differential equation can be selected. If the *Per substance*-button is selected then the substance panel appears (fig A-12). In this case, the differential equations subroutine must be imported. It can be selected and edited with the browse (...) - and *Edit*-buttons.

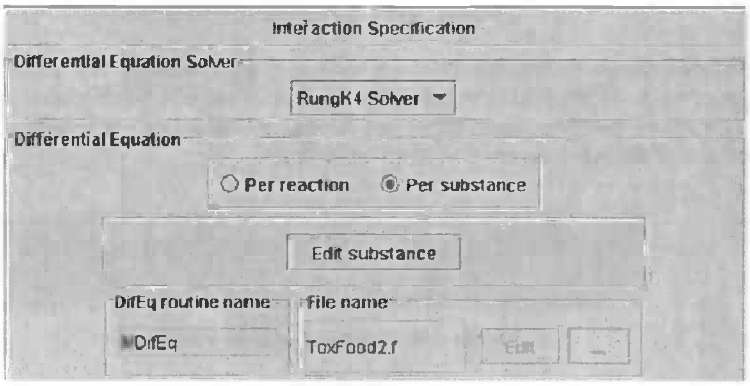
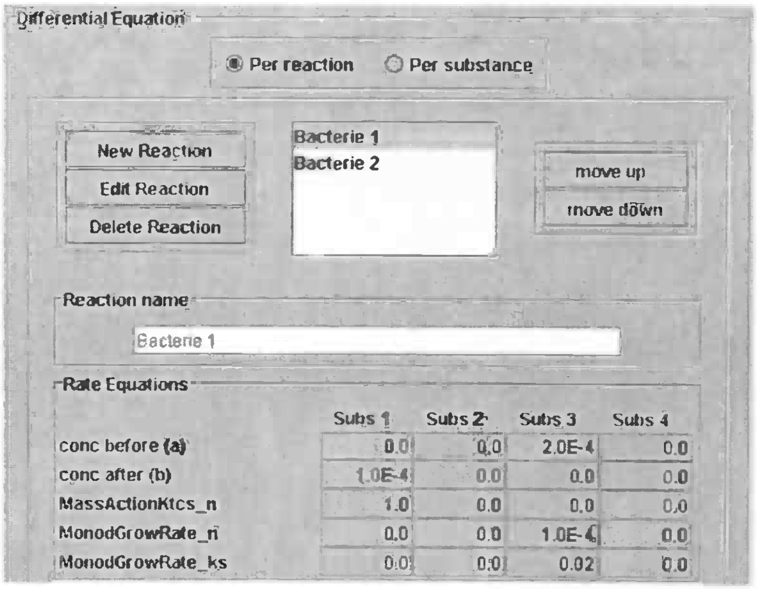


Figure A-12: Interaction Specification: Substances

If the *Per reaction*-button is selected, the reaction panel appears (fig. A-13). In this panel the user can create, edit and delete reactions. The reactions can be defined by filling in the appropriate values of the input arrays of the rate equations for every substance (see also chapter 5.2.2).



[✓ Per reaction] Per substance

Bacterie 1
Bacterie 2

Reaction name: Bacterie 1

	Subs 1	Subs 2	Subs 3	Subs 4
conc before (a)	0.0	0.0	2.0E-4	0.0
conc after (b)	1.0E-4	0.0	0.0	0.0
MassActionKtcs_n	1.0	0.0	0.0	0.0
MonodGrowRate_n	0.0	0.0	1.0E-4	0.0
MonodGrowRate_ks	0.0	0.0	0.02	0.0

Figure A-13: Interaction Specification: Reactions