Structuring Experimenting

E.L.G. Bertens (0557267)
T. de Ridder (0532086)
H.A. de Vos (0531297)

SE 420511

Masters Team Project

Supervisor: Prof. dr. ir. J.E. Rooda
Advisors: Dr. ir. A.T. Hofkamp
Dr. ir. L.F.P. Etman

Eindhoven University of Technology
Department of Mechanical Engineering
Systems Engineering Group

Eindhoven, 7 May 2007

Contents

Chapter 1

Introduction

1.1 Scope of the assignment

Almost everybody within the Systems Engineering Group (SE Group) is performing simulation experiments. However, little is known about how these experiments are actually done. The knowledge gained is often not recorded and shared with others. Everybody develops his or her own method of running simulations, which takes a lot of time and effort. This situation can be improved by analyzing, structuring and publishing these methods. To come to this improvement some steps have to be taken. To take some of these steps in the right direction, an assignment has been formulated as a Masters Team Project (MTP).

The main objective of this Masters Team Project is the establishment of a structured method for processing output of (multiple) simulation experiments with χ-models. The target user group consists of people within the SE Group that are doing experiments with χ-models on a large scale. The developed method has to be applicable for the described target group. When this is realized, a lot of time and effort can be saved in performing simulation experiments within the SE Group.

The project consists of the development of a tool for performing simulation experiments. In advance of the development, a survey of the current methods for running simulations, processing the output, and visualization of the output of the experiments within the SE Group has been carried out. Moreover, desires and recommendations of the people within the SE Group are included in the survey. Based on the analysis of the survey, an overview of the functionality is created. After that, an investigation into available software packages for performing simulation experiments has been executed. For the conceptual design of the tool, the available software packages are compared to other concepts of structured methods, on which the final concept was chosen and choices were made to design the framework and tools. Subsequently, an implementation of the final design has been performed, which results in the final tool or software package. To make the tool applicable for the target user group, most simple 'best practices' have been established. With this the operation of the framework and the accompanying scripts will be clarified.

1.2 Outline

In this report the steps that have been taken to realize a structured method for performing simulation experiments are described. The final product, a software package for performing simulation experiments with χ-models, has been clarified by means of some implementation examples.

In Chapter 2, the investigation into the current situation of methods for performing simulation experiments within the SE Group is described. Interviews that have been taken with staff members and SE students are summarized, which gives an overview of the current methods of simulating and desires within the SE Group. After that, an analysis on the survey has been performed, which is presented in Chapter 3. From this analysis, restrictions and functionality of a possible tool for performing simulation experiments have been defined. In Chapter 4 three basic concepts for implementation of such a tool are presented, followed by the concept that has been chosen for the final product. The actual implementation of the general part of the final product is described in Chapter 5, including choices that have been made and implementation details. A description of the more case-specific parts of the implementation is given in Chapter 6. In this chapter, two examples are presented which illustrates the functionality and usage of the final product. Finally, conclusions are drawn in Chapter 7 and recommendations are presented. These recommendations include possible extensions of the developed tool, as well as general recommendations for a more efficient method of performing simulation experiments within the SE Group.

Chapter 2

Survey

To develop a structured method of performing multiple simulation experiments, a survey has been made of the current practices of performing simulation experiments within the SE Group. This survey includes information about the different types and the size of the output data obtained from simulation experiments. In addition, the survey includes a list of most used software and tools for processing output. Finally, the survey includes desires and recommendations of the people within the SE Group.

2.1 Interviews

To obtain all the information for this survey, the following people within the SE group have been interviewed:

Prof. dr. ir. J.E. Rooda
: Full professor and group leader of the Systems Engineering Group

Dr. ir. A.A.J. Lefeber
: Research staff member of the SE Group, his work focuses on PDE modeling of a process.

Ir. A.A.A. Kock
: Ph. D. student, his work focuses on the broadening of the applicability of EPT as a tool for performance analysis and system optimisation.

Ir. J.A.W.M. van Eekelen
: Ph. D. student, his work focuses on building continuous approximation models of discrete event manufacturing systems.

E.A.F. van de Rijt
: Master student, he has done many simulation experiments during his internship in the United States. His graduation project concerns simulations of biological processes that take place during the fermentation of alcohol.

J. Hamer
: Master student, his work focuses on developing supervisory controllers with Wonham for a paint factory.

S.A.I. Veraa
: Master student, she develops a simulation model of transportation of containers from the seaside to the yard in a container terminal.

W.A.P. van den Bremer
: Master student, his internship concerns PDE modeling of a process.

During the interviews the following subjects have been discussed:

Running simulations Processing output data is generally correlated to the way simulation experiments are performed. Because of that, it is interesting and of importance to ask about how simulations are done.

Output Regarding this subject the structure, type, and size of the output have been discussed. Also the calculations on the output and how the size of output files is reduced have been discussed.

Software and tools Next, the software and tools for processing output have been discussed. Questions have been asked about the usage of these software or tools, i.e. if the usage of these tools is very time consuming and if these software or tools are orderly, so that other people of the SE Group are capable of understanding or using these software or tools. Furthermore, visualization of the output has been discussed.

Desires and recommendations It is important for this Masters Team Project to have knowledge about the desires and recommendations that people have within the SE Group. With this knowledge the project team can gear the method to the desires of the people within the SE Group.

2.2 Results

The outcome of the interviews have been organized into the discussed subjects. A summary of the interviews is discussed in this chapter. To indicate which information comes from which person, the last name of the person is used in between brackets. In chapter 3 the analysis of the interviews will be given.

Running simulations

The common way of performing simulations is using the SE rack systems that are available for people within the SE Group. On the basis of the interviews three different methods of simulating χ-models on the racks appeared to be in use. One of the methods is simulating on one rack. If a number of simulations has to be done, they are started up sequentially.

[Kock], [v/d Bremer], [van Eekelen]

Another method is using the Python script `batchlib.py`, developed by A.T. Hofkamp, which can start a simulation of a χ-model on several rack systems simultaneously. A remark to this Python script is that it is not fully safe. For example when an error occurs during the simulations some rack systems can fail and output files can end up on many rack systems. [Lefeber]

The last method to be discussed is the SE Cluster, which is also used within the SE Group. This cluster consists of ten rack systems (with 2 cpu's each) that are connected in parallel. In this way several simulation experiments can be done at the same time. The SE Cluster has to be directed with a Python script that is available on the SE Wiki page. [Rij06] The SE Cluster can be used for two different kind of simulation experiments:

- A long simulation can be split up into pieces, which will simulate in parallel on the cluster racks. In this case a prove for ergodicity of the χ-model has to be given.
- Several simulations with the same χ-model with different input parameters running on the SE Cluster in parallel.

Disadvantage of these SE Cluster racks is the lower processing speed. [v/d Rijt]

Output

Most used type of output file is a text file. The output file of multiple simulation experiments can for example consist of 3.000.000 lines [v/d Rijt]. It can also exist of 1000 output files from 1000 simulations [Lefeber]. Generally, the output file contains a number of columns that include the output parameters (e.g. processing times, buffer levels etc.) at several time steps, at several processes or for each lot or job. Examples of common types of output used within the SE Group are shown below. The first item in each row indicates the time, the second item gives the name of a process and the third item contains the output value. In the example below, each row contains the number of lots in progress on the indicated moment in time for the mentioned part of a manufacturing line.

```
0.0     Buffer1     9
2.0     Buffer2     5
2.0     Machine1    1
```

Another form of output can be as follows:

```
0   9.8139127092076    10.31799669595326   10.3156612970864
0   7.1352070332582    7.45812768529704    11.6347680028546
1   2.1118928850355    0.21477439700813    1.0537647205416
```

In this case, the first parameter represents the input parameter used in the χ-model. The following three values represent the output values, for instance process times of three processes in the model.

Another aspect of the output is the way of collecting the output. Two different methods of collecting the output are in use within the SE Group. The first method is writing the output to a text file during or after the simulation. Mostly, this is done by saving all output that is written to the screen, in a text file. It can also be done by using a Python function in the χ-model to save the output in one or more text files.
Another method is updating values during simulation, writing to a text file is not necessary. Generally, this is done when the amount of simulations is so large, that it is not possible to store the output of every simulation on the computer, due to the size of the output. In that case, the output values will be updated after every simulation, which results in one output file. This is done in two different ways:

- *Matlab*
  An m-file is written which reads the output file (text file) after a simulation. With the values in the output file Matlab updates the desired output values. After that, a new simulation starts and it overwrites the old output file. In that case the original output will not be kept. [v/d Bremer]

- *Python scripts*
  A Python script is written which reads the output of the χ-model during or after a simulation. This Python script saves the useful output in a matrix for example. After each next simulation the Python script updates the matrix. In that case, the χ-model does not need to write output to a file, but the output flows to Python directly. This construction saves time and capacity, because no output has to be written to a file and no output has to be kept. [Lefeber]

Software and Tools

There are several methods for processing the large amount of output into the desired information or plots.

- *GREP*
  This is a Linux program, which filters the desired data of the output file after giving a correct commando. A great advantage of this program is its speed. [v/d Rijt]

- *Matlab*
  Matlab imports output files and sorts out the desired data of the output file with an m-file. An advantage is the great calculation skills of Matlab. A disadvantage is that it costs time to import the output data into Matlab. [v/d Bremer]

- *Python scripts*
  Python reads the output files and sorts out the desired data of the output file according to what is written in the Python script. An advantage is that Python reads the data of the output files instead of importing the data, which saves time in comparison with Matlab. A disadvantage is that it can be difficult, writing a Python script that sorts out the desired data of the output file. [Kock], [Veraa], [van Eekelen], [v/d Rijt]

Another aspect of software is the visualization software or tools. Most used visualization software within the SE Group is presented in Table 2.1.

Table 2.1: Visualization software used in the SE Group

Visualization Software	**Description**
Matlab	High-level technical computing language
Excel	Spreadsheet software
Labview	A real time application. Output in type of graphs can be seen during the simulation
Gnuplot	An interactive plotting program [v/d Rijt]
Simplot	This tool (a Python script) converts output data into input data that is suitable for plotting with Gnuplot
Latex	For making graphs after a simulation [van Eekelen]
Noodle View	A prototype for the interactive visualization of multivariate state transition graphs [Hamer]
Diagraphica	A prototype for the interactive visual analysis of multivariate state transition graphs [Hamer]

Desires and recommendations

The SE Group members came up with the following recommendations.

- Limiting file I/O. This means that the output of a simulation does not have to be written to disk but remains in the computer memory. When a large amount of data occurs, limiting file I/O will save a lot of time. [Lefeber]
- A graphical interface, e.g. a block structure, that clarifies the method or structure of performing simulations. [Rooda]
- A manual with tips and tricks for simulations. More information about the χ-language 1.0 for advanced users. [Veraa] [v/d Rijt]
- An elective course which will include all important information about how to perform simulations. Students have just too little knowledge about performing simulations. [Kock]
- Develop a method which calculates after each simulation if the experiment is statistically well grounded. [van Eekelen]
- Develop a method, different from Simplot, which can plot a figure quickly from a large data output. [v/d Rijt]
- Efficiency with respect to the use of the rack systems. Perhaps, this can be done with a Python script that watches when an SE rack becomes free. When a rack system becomes available the Python script starts a simulation on the rack system. [Kock]

Chapter 3

Analysis of the survey

From the survey of the interviews, achieved during the first phase of the project, similarities in the functionality are found in the different methods currently used for performing simulations. The tool that has been developed in the next phase is based on these similarities. The functionality can be divided in three blocks, as shown in Figure 3.1. The first block represents running the χ-model. The input required here, is the model itself as well as the input parameters. From this block, output data is transferred to the 'Processing data' block, where several actions are performed on the data. Finally the processed data is presented in the last block 'Obtaining Information'. The most common way of presenting this data is by visualizing it in diagrams.

3.1 Current functionality

Within the SE Group, a number of scripts have been written and tools exist for performing certain operations, calculations or actions that are useful in the process of doing experiments. However, many scripts and tools are only known by a single researcher and not by the whole group. Collecting these scripts, making them more general usable and adding documentation can already be quite a large improvement of the current situation. Only a very small and uncertain start of such a process has been made on the SE Wiki pages.
The different functions of the existing tools and scripts are divided in the three blocks

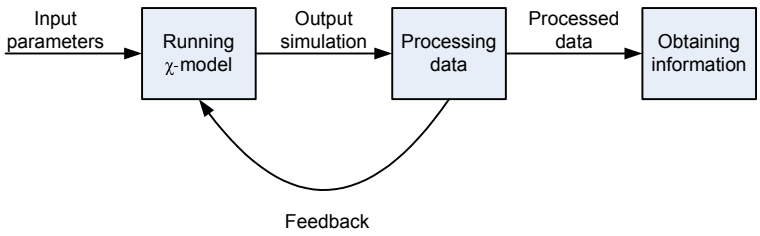

Figure 3.1: Functionality

described above and shown in Figure 3.1. In the next subsections the functions within the blocks are defined, as well as the data flows between the blocks. The data flows are of great significance for the implementation of the tool support, where different scripts will communicate.

Running χ-model

A number of differences have been discovered in the way of performing simulation experiments within the SE Group. Some people have to perform multiple simulations with a χ-model without varying input parameters. Mostly, this is needed for statistical reasons. It costs relatively much time to start up a file for one simulation. When for example 1.000.000 simulations have to be done, it is more efficient to do already 1000 simulations within a χ-model. Subsequently, the χ-model has to be simulated for 1000 times. Other people have to perform multiple simulations with a χ-model with different input parameters for every simulation.
For a high utilization of the SE racks, decisions can be made concerning the location to run the simulation. The most simple way of running a simulation is on an SE rack. The user only needs to start a serie of simulations after logging into one of the SE racks. It is also possible to spread the simulations over multiple SE racks by using a Python script. This increases the utilization of the rack systems. A disadvantage of such a script is that it can cause serious problems when the simulations do not work properly. A more safe option of spreading simulations over multiple cpu's is the usage of the SE Cluster. With a Python script jobs can be spread over approximately 10 processors. This script is available via the SE Wiki page. [Rij06]

Besides the difference in the location of the simulations, three different simulation 'modes' can be distinguished:

1. Test mode; run one simulation to check the model.
2. Run all simulations in advance of the data processing.
3. Run the simulations in "parallel" with the data processing. After each simulation run, the data is analyzed and a next simulation run is started at the same time.

Input parameters In many cases, simulations need to be performed for different values of input parameters. Examples of these parameters are: mean process times and variation, buffer sizes and seeds. Some problems have only one variable input parameter, but others can have more, which may increase the number of possible simulations exponentially.

Output The output of the simulations is roughly divided into two different types. On the one hand it can contain values belonging to several processes, jobs or time steps in the model. In the latter case, a buffer WIP level is often a value to measure for several time steps. The output can then be given as a column for the actual time step, followed by one or more columns with values of the corresponding WIP levels. On the other hand the output of the χ-model can consist of values calculated in the model, such as EPT data or process times. In this case, the output of each simulation is an array of one or more values, specific to the case.
The output is written to files, e.g. text files, that are read later on by the processing part.

Another option is to pipe the output to the processing part, which reduces the file I/O and required storage space.

Processing data

Although the computations in this process can be very case-specific, some calculations are generally executed during experiments within the SE Group. For instance the calculation of mean values and standard deviations is a common operation, performed after a large number of simulations. Sometimes statistical tests are performed with this data. Another common operation is the selection of data from a large output file. Also the conversion of data from one format to another is a type of data processing that is performed regularly.

Processed data Like in the previous process, there is a choice between writing the data to files or pipe it to the next process.

Feedback When statistical tests on the output data of the χ-model are performed in parallel with the running of the simulations, the outcome of these tests are transferred back to the 'Running' block, where is decided whether another simulation is started or not. Feedback data is also required in the case when values of input parameters for the next run are based on simulation output of the current run.

Obtaining information

In most cases, the desired information is a diagram, for example a graph in which WIP levels are plotted against time. Sometimes a set of those diagrams is desired, possibly transformed to a movie. Another form of information can be a distribution type of EPT values with corresponding parameters. This can be achieved by using distribution fitting tools. Some people want to save the output data from the χ-model in a well organized way for further analysis in the future. This can also be seen as information. It has been noticed that most of the software scripts and tools used for visualization are directed by the programming language Python.

3.2 Other desires

- Reduction of the amount of output data (storage space).
  This can be done by filtering data (in the processing part), i.e. only storing a selection of the data instead of storing all output data of the simulations.
- In some simulations it may be useful to store data after each simulation. With this data a quick check of the operation of the model is performed while the experiment is running. This backup data is also useful in the case a simulation experiment gets stuck.
- Reduce the File I/O.
  This increases the speed of the entire process of performing simulations and processing data. A way to realize this is piping data instead of printing data to intermediate files.

Some functions described above are already used in general and are easy to implement. Other functions are used for specific cases. A selection of these functions has to be made before the actual implementation can start. A conceptual design for the developed tool support is specified in the next chapter.

Chapter 4

Conceptual design

The previous chapter describes the desired functionality of the tool support that has to be developed. For the implementation of this functionality, a number of choices has to be made with respect to a (general) part that has been implemented in this project and a part that has to be implemented by the end user for a particular case. The many ways in which these choices can be made, are roughly divided into three different kinds of implementations:

- A complete software package in which the user only has to select a number of options, set some parameters, and provide a χ-model.
- A framework that only provides the necessary communication between the three different parts of the whole process. These three parts are running the χ-model, performing the calculations, and visualization of the results. The user has to provide the scripts or tools that do the actual running of the χ-model, the calculations on the output data, and the creation of plots. The framework only connects the different scripts and tools by coordinating the data flow from one part to another.
- A collection of scripts, tools and software packages that can perform the different actions that are desired in the whole process of doing simulations and processing the output. In this case the user has to take care of the data exchange and coordination between the different tools.

Sections 4.1, 4.2 and 4.3 describe these implementation options in more detail and discuss the advantages and disadvantages of them.

4.1 Complete software package

To discuss the advantages and disadvantages of a complete software package, a search on the internet for available software for performing multiple simulation experiments has been performed. During this internet search, a simulation software survey has been found. [Pub05] In this survey approximately 50 simulation software packages are presented with their properties. On the basis of these properties, some conclusions have been formulated:

- Most software packages cannot function in a Linux environment. It is required that software packages also run in a Linux environment, since simulations of χ-models on the rack systems take place in a Linux environment. In the case the software package can be integrated on the rack systems, the output of simulations does not have to be moved from the rack system to a Windows environment after each simulation.

- In most of the software packages, too many options and functions are integrated. The purpose of this project restricts to simulation methods and calculations that are used within the SE Group. For that reason it is not useful to have a software package with so many possibilities. An example of a useless possibility that is integrated in many software packages, is a model building environment, since most people within the SE Group use the χ-language for building a model.

- It takes quite some effort and time to learn these software packages.

- Most commercial software packages are quite expensive.

Besides the disadvantages concluding from the internet search, there are also some other advantages and disadvantages of a complete software package solution. A big advantage of a complete package is the very quick setup of an experiment. Instead of writing scripts that perform the desired operations, the user only has to select the desired options and set some parameters. It also requires less knowledge of the user about the details of the process of doing experiments and the user does not need to learn a scripting language.

The main disadvantage is the limited flexibility of such a package. The functionality is limited to the options that are provided and the output data of the χ-model has to be exactly in the format that is desired by the predefined functions and methods. Just the smallest difference between the wishes of a user and the provided possibilities causes the package to be useless to this user. When the package is open source, this problem is partly solved, because the user can adapt the things that are not exactly like he or she would like to have it. However, it is not expected that the SE Group members are willing to investigate and extend complex software that is written by others to adapt it to their needs. People of the SE Group will presumably return to use self-written scripts in such a case. Another disadvantage of the large part of code that is not written by the user, is the fact that many researchers really like to know (and control) everything that is happening, instead of just putting something in and getting something out of it.

Besides the above mentioned disadvantages, also the size of the package can be an important drawback. A package that contains all kinds of functions and methods takes a relatively long time to start up and uses a rather large amount of memory and probably takes quite some processor time. This can keep experimenters from using it when processing time and memory capacity is a major issue in their experiment.

A possibility to increase the flexibility of a complete package, is to provide a kind of scripting language in it, in which the user can define her or his own functions and methods. However, this requires the user to learn this particular scripting language. One way of letting the user choose in which scripting language or with which tool the user would like to perform simulation experiments is to only create a framework for the whole process. This option is the subject of the next section.

4.2 Framework only

The framework as mentioned above can be seen as a program that executes the complete process of doing experiments, but has some gaps that have to be filled in by other tools or scripts that handle a particular part of the process. The framework can provide a number of parallel processes in which different parts of the experimenting process are handled. For example one process for running the simulations and another process for performing calculations on the simulation output, which run in parallel. Buffers can be provided by the framework for communication between the different processes that run in parallel. In this way the framework relieves the user from organizing and structuring the whole process; the user only has to deal with the implementation of calculations and other operations or actions that contain characteristic details for his or her experiment.

A framework in which the user can fit his or her own scripts and tools is much more flexible than a complete package in which the user can only select some predefined functions and methods, and set some parameters. In a framework solution, the user can still continue to use (parts of) his or her own scripts and tools. This can lower the threshold for starting to use such a framework.
A disadvantage of a framework with respect to a complete package is the knowledge that is still required for writing scripts and applying other tools for the different parts of the experimenting process.

In contrast to the flexibility in the functions and methods that are used for the different parts of the process, the framework restricts the flexibility in the structure of the whole process. An approach in which this restriction is removed, is providing only a collection of scripts and tools that perform certain actions and calculations. The user only has to tie the scripts and tools together by means of a main script. This method is described in more detail in the next section.

4.3 Collection of scripts and tools

The advantage of providing a collection of scripts and tools for performing simulation experiments is the great flexibility of all these tools and scripts without any restriction. The drawback is the unstructured way of linking the different tools and scripts to support the complete process of doing experiments. This causes extra effort that is needed for understanding the experimenting process of other people and for understanding experimenting that is done earlier and has to be continued.
Besides that, the user still needs quite a lot of knowledge of a scripting language to create a process that automatically performs the different tasks that are required. The conversion of data formats and redirecting the output of one tool towards a next tool can be rather complicated.

As can be seen, all three concepts that are described, have their advantages and disadvantages. The next subsection discusses the design that is chosen for the final product of this project, which uses aspects of all three concepts.

4.4 Final design

The final design is a combination of a framework, and a collection of scripts and tools that can run within this framework. Together they form a kind of complete package that is very easy to extend.

The framework provides three processes that run in parallel. One is for running simulations with a χ-model, one for performing calculations on the simulation output, and one for visualization of the desired information (see Figure 4.1). The different parallel processes exchange data by means of buffers.

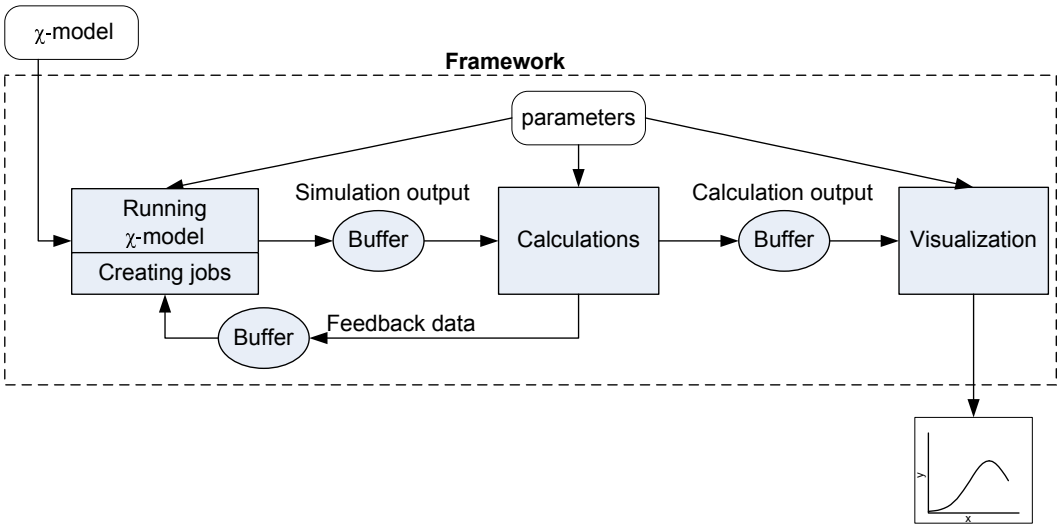

Figure 4.1: The framework

Within each of the three processes the user can implement his or her own scripts and use his or her own favorite tools. The division into three clearly separated processes provides the possibility to reuse parts of different experimenting set ups. For the process in which simulations are run, one can for example choose whether to use a script that runs the simulations on an SE rack or use the SE Cluster. For the calculation process, the user can choose to do the calculations in Matlab or implement them in Python or use another program. For the visualization, the user can use a script that creates a graph in Gnuplot or a spreadsheet program, in which the obtained information can be presented in the form of a table, for example. The user can also create a script for writing the calculation output to a file.

Another advantage of the division into three separate processes is their parallelism. The parallelism of the *run* process and the other two processes provides the possibility to run simulations and process output data at the same time. Simulations can be run either in parallel or sequential, dependent on the actual script that is used for running the simulations. When a simulation is done, the χ-simulation output is written to the buffer, together with the values of the input parameters that are used for running the simulation. As soon as the

calculation process notices that new simulation data is written to the buffer, it reads this data and performs the desired calculations. These calculations can be for example updating the average of a number of similar simulations. After performing the desired calculations, the output data can be removed or compressed and stored if desired. This reduces the amount of memory or disk space that is required in comparison with a non-parallel method in which calculations do not start until all simulations are completed.

The separation of the *visualization* process and the *calculation* process provides the possibility to do the calculations in a programming language or tool that can handle large amounts of data and perform calculations very well, while at the same time running another (graphical) program that makes a graph of the calculated data. Because of the parallelism of the two processes, the graph that is drawn, can be updated each time new calculation data becomes available. In this way the user can check whether the simulations produce the expected data while the experiment is going on. This gives the user the option of aborting the experiment before it is finished, avoiding a lot of otherwise wasted simulation time.

At a central place within the framework, a number of parameters can be set, like for example a parameter N, which is the number of times each simulation needs to be done. These parameters are used as input for the scripts and tools that run in the different processes. The mentioned parameter N might be used as input parameter for the script that starts the simulations of the χ-model as well as for the script or tool that calculates for example the average of these simulations.

The developed framework has been written in Python. A number of reasons for this choice are listed below:

- Python provides all necessary possibilities for creating the parallel processes and buffers.
- It is well-known by most people within the SE group. These people can easily extend and adapt the codes, because it is an interpreted language. Because of that the framework can be adapted for any specific case.
- It runs on the Linux-systems without any extra difficulties. Python 2.3 is already used for a number of purposes on the racks. The framework is written in Python 2.4, but it is also made suitable for Python 2.3.
- It provides various useful possibilities for interaction with other software.

The next chapter discusses the final product, which is the actual implementation of the concept that has been discussed here.

Chapter 5

Final product

Chapter 4 discusses the developed tool in a conceptual way. In this chapter, the actual implementation is described and some main decisions that have been made with respect to the implementation are explained.

5.1 Main structure

The final product is a software package and consists of the Python files listed below. The asterisks denote any kind of name corresponding to the experiment that is executed.

- `frame.py`
- `run_*.py`
- `calc_*.py`
- `vis_*.py`
- `main_*.py`

Appendix A contains the code that is written in `frame.py`. Examples of the other files are included in Appendix B and Appendix C. These files are discussed in more detail in Chapter 6.
Figure 5.1 shows how the different files are related to each other.

In `frame.py` a class called `Frame` has been defined. `Frame` possesses three execution threads, which correspond to the *run*, *calculation* and *visualization* processes. In these execution threads, three Python functions, called `run_func()`, `calc_func()` and `vis_func()`, are executed. So, these functions are executed concurrently. The three functions are defined in separate files called `run_*.py`, `calc_*.py` and `vis_*.py`. These files are based on the template files `run_template.py`, `calc_template.py` and `vis_template.py`. A number of different run, calculation and visualization functions are already defined and are explained in Chapter 6.

Three buffers are available within `Frame` for communication between the three threads. First of all, the *run* process and the *calculation* process are connected through a buffer,

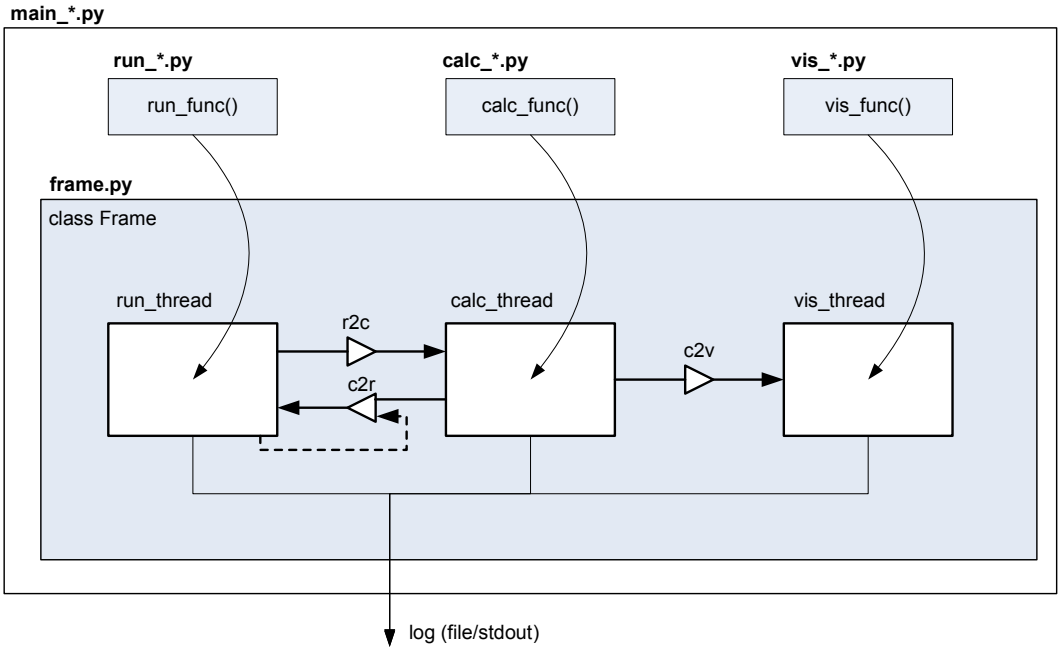

Figure 5.1: The Python files that form the final product.

called `r2c`. This buffer is used for passing the output data of the χ-simulations from the *run* process to the *calculation* process. Second, a buffer called `c2r` is used for collecting 'simulation jobs' before and/or during the experiment. When an experiment is performed in which a fixed number of simulations have to be executed, the buffer `c2r` is initially filled with simulation jobs, created in the *run* process. This is indicated in figure 5.1 with the dotted line. In experiments with a feedback construction, jobs are also created in the *calculation* process and written to buffer `c2r` during the experiment. Finally, a buffer called `c2v` is provided for passing data from the *calculation* process to the *visualization* process.
Data can be written to and read from a buffer by means of its `write()` and `read()` methods. A buffer can be closed by its `close()` method to indicate that no data will be put onto the buffer anymore. From this information, the process that reads from the buffer, can determine when its job is done.

An experiment is started by executing a Python script called `main_*.py`. This script performs the following actions:

- Import the three Python functions from the files in which they are defined.
- Set global parameters for the experiment.
- Create an instance of `Frame` and pass the required functions and parameters to it.
- Start the frame, i.e. start concurrent execution of the three functions.

A template for this script is provided in the file `main_template.py`.

When setting up a simulation experiment, the user first selects the three functions that come closest to his or her wishes. Subsequently, the user can make the desired adaptations to these functions. Finally, the user creates a `main_*.py` script with which he or she can start the simulation experiment.

5.2 Implementation details and decisions

In the previous section, the final implementation of the developed tool is described. In this section a number of implementation details are discussed and choices that have been made with respect to some implementation alternatives are explained.

Wrapping Python functions

The three parallel processes are implemented by means of threads, each of them executing a Python function. In these Python functions the desired actions are implemented in Python code or an external tool is called. As an alternative, the external tools could be executed directly, without being 'wrapped' by a Python function. A number of reasons for the using 'wrapping' Python functions are the following:

- The Python functions are all called in the same way, so no change has to be made in the frame itself when exchanging functions between different users or experimenting set ups.
- The Python functions can exchange data by means of Python objects that are written to and read from the buffers. The format of this data can be easily formatted in a standard way, such that functions of different processes can correctly communicate with each other. However, in most external tools it is not possible to format the output in any desired way.
- From within a Python function, one or more external tools can easily be called, beside doing actions with Python. For example Matlab might be used to perform a calculation, while Python itself performs other data processing tasks, in the same process.

Communication through 'piping'

For the communication with external tools (like the χ-simulator or Gnuplot) so-called 'piping' is used. This means that the output of the tools that is normally written to screen, is redirected to make it accessible in the Python code that called the tool.
An alternative could be writing the output (for instance the χ-output) to a file that is read later on, after the simulation is finished. However, this file I/O may take a lot of time, compared to the processor time that is actually needed for performing the χ-simulations. This drawback is avoided when piping is used for exchanging data.

External functions

The `Frame` class is implemented in such a way that it calls three external functions in its run, calculation and visualization threads. These functions are passed as arguments to the `__init__()` method of `Frame`. The `Frame` class could also have been implemented with three methods that are executed in the run, calculation and visualization threads. Then, the user can implement his or her own methods by defining a subclass of `Frame` and overriding these three methods. This is perhaps a more elegant solution from a programming point of view. However, the external functions, implemented in separate files, are much easier to exchange between different users and different experimenting set ups. That is why external functions are used.

Implementation of the buffers

For the implementation of the buffers, a `Buffer` class is defined. These buffers accept any kind of object as data, except `None`. When a buffer is closed, `None` is written to its internal list. The `eob()` (end of buffer) method returns `True` if the next data to be read from the buffer is this `None` value.

In this chapter, the final product is presented and the implementations details and decisions are clarified. To test if this final product is working correctly, some experiments have been done. Therefore, simple χ-specifications and Python scripts for the *run*, *calculation*, and *visualization* processes have been written. From the experiments, it turned out that the processes are exchanging data correctly through the buffers, which means that the framework is working properly. Moreover, the performance of the final product is tested by comparing the process times in each process. From that, it appeared that starting the χ-simulations in *run* process is the most time-consuming action of the whole experiment. Other actions in Python like data flow, calculations, and visualizations are hardly slowing down the experiment.

To give an impression of the usage of the software package and some of its functionalities, two completely worked-out examples have been created and are clarified in Chapter 6.

Chapter 6

Implementation examples

To explain the functionality of the implemented tool and to show that the framework is working properly, two examples are considered. For both examples, four specific Python files have been created, namely a main file and three files for the *run*, *calculation* and *visualization* processes. These files are included in Appendix B and Appendix C. Also very simple χ-models have been made to represent simulation models that are used in practice. In the first example, simulations are run a fixed number of times with a χ-model that has multiple input and output parameters. With the output of the model, some calculations are done and a 3-dimensional plot is created from the data.
The second example includes a more simple χ-model, without any input parameters and with only one value as output. The functionality of this example contains a decision-making process in the calculation thread. According to a simple statistical test with the mean value of the χ-output, the *calculation* process decides whether a new simulation is needed or not. Communication takes place with the *run* process which either runs a new simulation or terminates. Also in this example, a plot is created to visualize the mean value of the simulation output.

6.1 Example 1: Multiple parameters

A simple χ-specification has been made to model a system with two input parameters and four output parameters. The values of the four output parameters are given with the following formula:

$$output(i) = input_1 + 2 \times input_2 + i + \sigma \tag{6.1}$$

where i is the index of the output value with range from 0 through 3. The variables $input_1$ and $input_2$ are the values of the input parameters used for a simulation and σ is a standard normally distributed value. This model can represent a simple manufacturing line including four buffers. The two input parameters might be two mean process times, as the output values might represent the WIP level for each buffer in the system. The output is now linear dependent on both input parameters and buffer index, and includes stochasticity.

The goal of the experiment is to analyze the output of the simulation model for different values of the input parameters. Therefore, the mean values and standard deviations are calculated from the χ-output. To visualize the results, the mean values are plotted against

both two input parameters. This is done for only one output value (buffer) at a time. In this example, a plot is created for buffer 3. When enough repetitions are done for every simulation, the plot should give a surface with increasing values in the directions of the input parameter axes. The calculated standard deviations are written to a separate text file. The three processes *run*, *calculation* and *visualization* run in parallel.

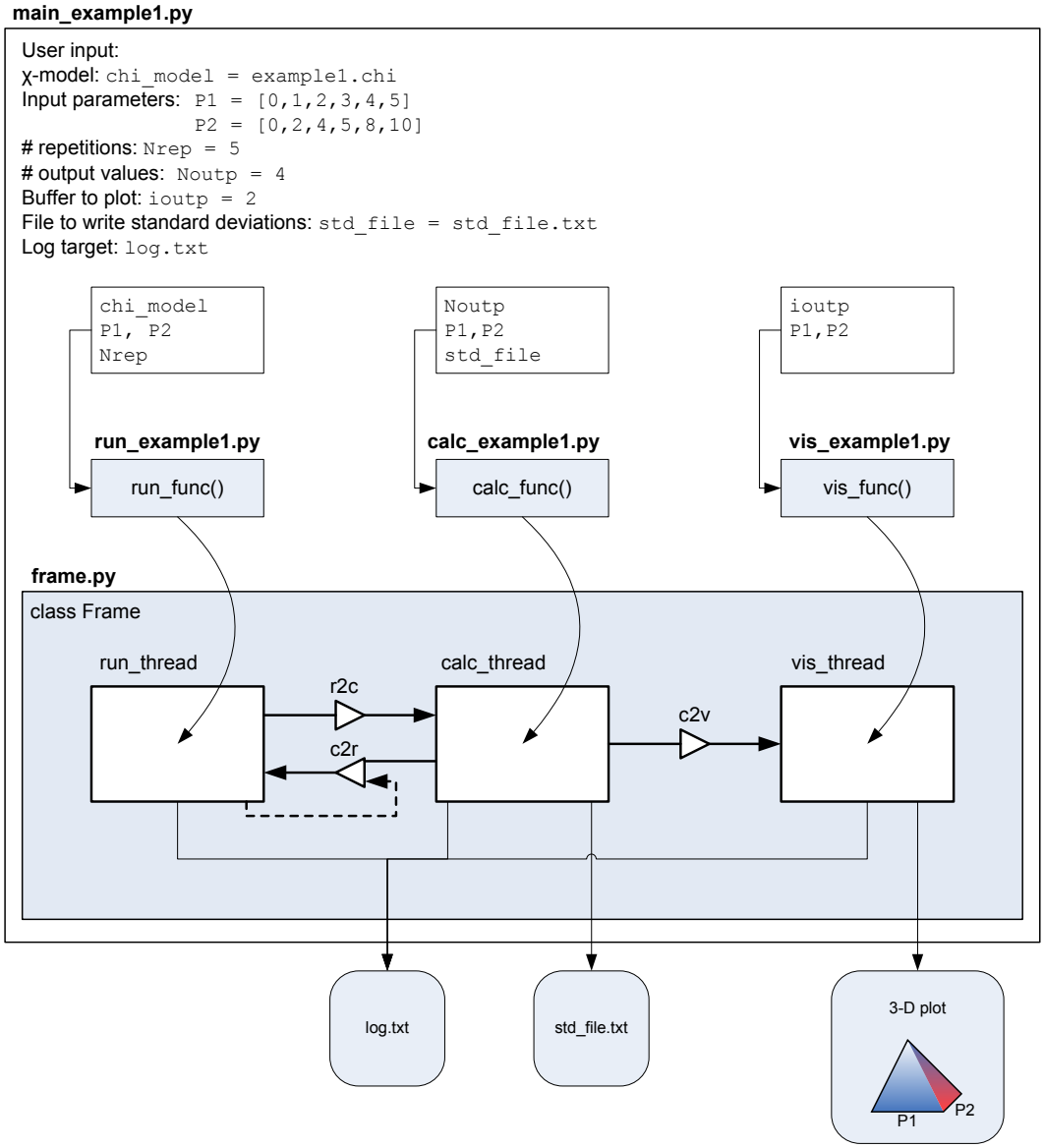

Figure 6.1: Structure of example 1

User input The χ-model '`example1.chi`' is the simulation model for this example and requires two input parameters. Input parameter 1 varies from 0 to 5 with step size of one

and input parameter 2 is specified as follows: `[0, 2, 4, 5, 8, 10]`. Because the model has four output values, $Noutp$ is set to 4. The number of repetitions for each simulation, $Nrep$, is set to 5 in this experiment. This number is not based on statistical tests and might not be large enough for actual experiments. For the visualization of the data, the mean values of buffer 3 are plotted only. Therefore, $ioutp$ equals 2, corresponding to the index of buffer 3 (index starts with 0). Finally the target for log information and standard deviations is chosen; both are written to a text file.

Run process The χ-model runs $Nrep$ times for all combinations of input parameters (6×6). This is done by using the script `run_example1.py`, which first calculates all combinations of input parameters and puts them as 'simulation jobs' in buffer `c2r`. This is realized with the following code:

```
for n in range(Nrep):
    for p1 in P1:
      for p2 in P2:
        c2r.write( (p1,p2) )
```

After that, it handles all simulation jobs sequentially, by reading the input parameters from the buffer `c2r` and running the χ-model on one SE rack with the corresponding input parameters. The script terminates when all simulation jobs are performed.

Simulation output The values of the output of the χ-model are distributed according to a normal distribution and are of type real. This data is transferred to the *Calculations* process via buffer `r2c` together with the combination of input parameters that is used. In the case that the input parameters of one simulation are 3 and 8, the output might look like this: `([3,8], "0.012  0.034  0.045  1.789")`, in which second item is one string, consisting of the four output values of the χ-model.

Calculation process The calculation of the mean values and standard deviations is performed with the script `calc_example1.py`. The function within this script needs some information to define the format of the incoming data. In this case, four WIP levels for 6×6 different combinations of input parameters need to be updated. This results in a 3-dimensional, $6 \times 6 \times 4$ matrix with mean values. Any time a set of data is available from the buffer `r2c`, the mean values and standard deviations are updated. For updating the n-th mean value, formula 6.2 is implemented in the Python script. [Roo06]

$$\overline{w}_n = ((n-1)/n) \times \overline{w}_{n-1} + w_n/n, \text{ for } n = 1,2,3,... \tag{6.2}$$

where w_n contains the output values coming from the *run* process. So, the four values for all output parameters are updated at once. A similar formula is implemented for updating the standard deviations. The script `calc_example1.py` terminates after all data is processed. For more details about the calculation script Appendix B or the SE Wiki pages can be consulted.

Calculation output The matrix including the mean values of output parameters is transferred to the *visualization* process via buffer `c2v` after each update. The matrix with standard deviations is written to a text file after all simulation runs are completed and all data is processed. After each update, information is written to the log file.

Visualization process In the final process, '`vis_example1.py`' is used to create a diagram in which the mean value of one selected buffer is plotted against the two input parameters. This will generate a 3-D plot with the input parameters on the x and y axis and the mean output value on the z axis. The values of both input parameters and the index of the desired buffer is given as a parameter of the process *visualization*. For making the plots, Gnuplot is used. Anytime a set of data is available from the buffer `c2v`, the mean values are plotted. This results in a diagram that is updated each time a new simulation is finished. As said in Chapter 4 it can be an advantage, that the user can quickly check whether the simulations are performed properly at any time.

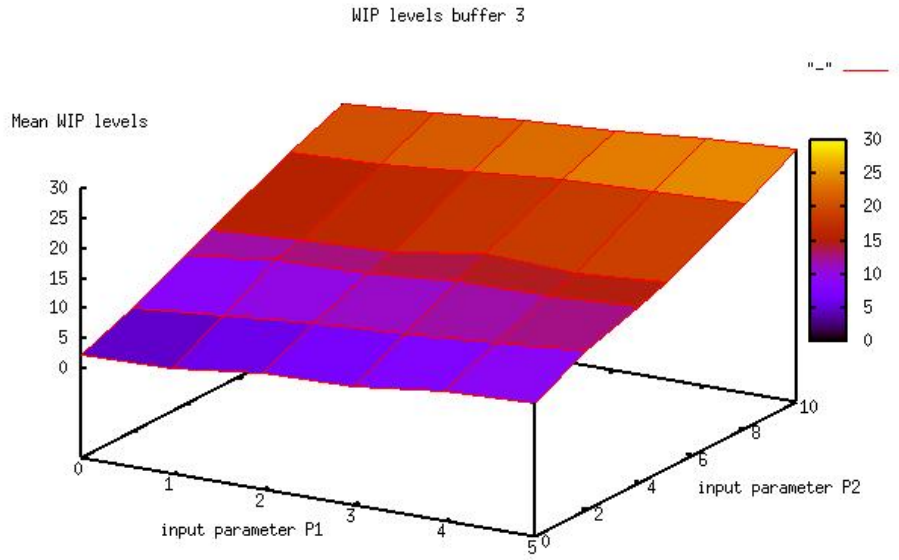

Figure 6.2: 3D plot of the mean values of example 1

Validation Figure 6.2 gives the plot that is created in the *visualization* process. It shows a rough surface, because of only 5 repetitions are done for each simulation. The

minimum value in the plot is 2, where the input parameters are both 0 and the maximum is approximately 27, where input parameter 1 is 5 and input parameter 2 is 10. These values match with the outcome of the formula given in this section.

6.2 Example 2: Feedback

In the previous example, the buffer `c2r` is only used for storing jobs initially, but not for creating extra simulation jobs during the experiment. To show that this buffer can be used for running a number of extra simulations, based on a statistical test, the following case is considered. Again, a very simple χ-model is made, which now represents a system without input parameters and only one output parameter. This value is again distributed according to a normal distribution and is of type real. The goal of the experiment is to run this simulation a number of times until the mean value of the output has a variation lower than a specified value. At that moment, the experiment terminates. This is realized by checking the variation of the last updated mean values after each simulation. After that, the decision is made whether to run a new simulation or not. The mean value is plotted into a figure each time it is updated. The simulation runs and calculations are performed sequentially, in contrast with the processes in example 1.
In this example, all log information is written to the screen.

User input The χ-model '`example2.chi`' has no input parameters. Besides the name of the χ-model, the only input the user gives is the number of mean values to be evaluated $Ncheck$ and a number for determining the maximum allowed variation $Xcheck$ of the last $Ncheck$ mean values. The value of $Ncheck$ is set to 20 and the value of $Xcheck$ is set to 0.1. Finally the log target is defined as `stdout`, so the log information from all processes is written to the screen.

Run process The Python script '`run_example2.py`' uses both buffers `r2c` and `c2r`. Initially, `c2r` is filled with 20 jobs. After that the process runs the χ-model until the buffer `c2r` is closed. When the buffer is not closed, the the value 0 is read from `c2r` and a new simulation is started. The outcome of the simulation is collected and log information is written to the screen.

Simulation output The outcome of the simulation contains a string with only one float. This is written to the buffer `r2c`.

Calculations process The calculations and decisions are performed with the script '`calc_example2.py`'. First, it calculates the mean value of the last $Ncheck$ values, which are stored as *means*. The following script takes the decision to start a new simulation or terminate the experiment.

```
if n >= Ncheck:
  if max(means)-min(means) < Xcheck:
    c2r.close()
  else:
    c2r.write(0)
```

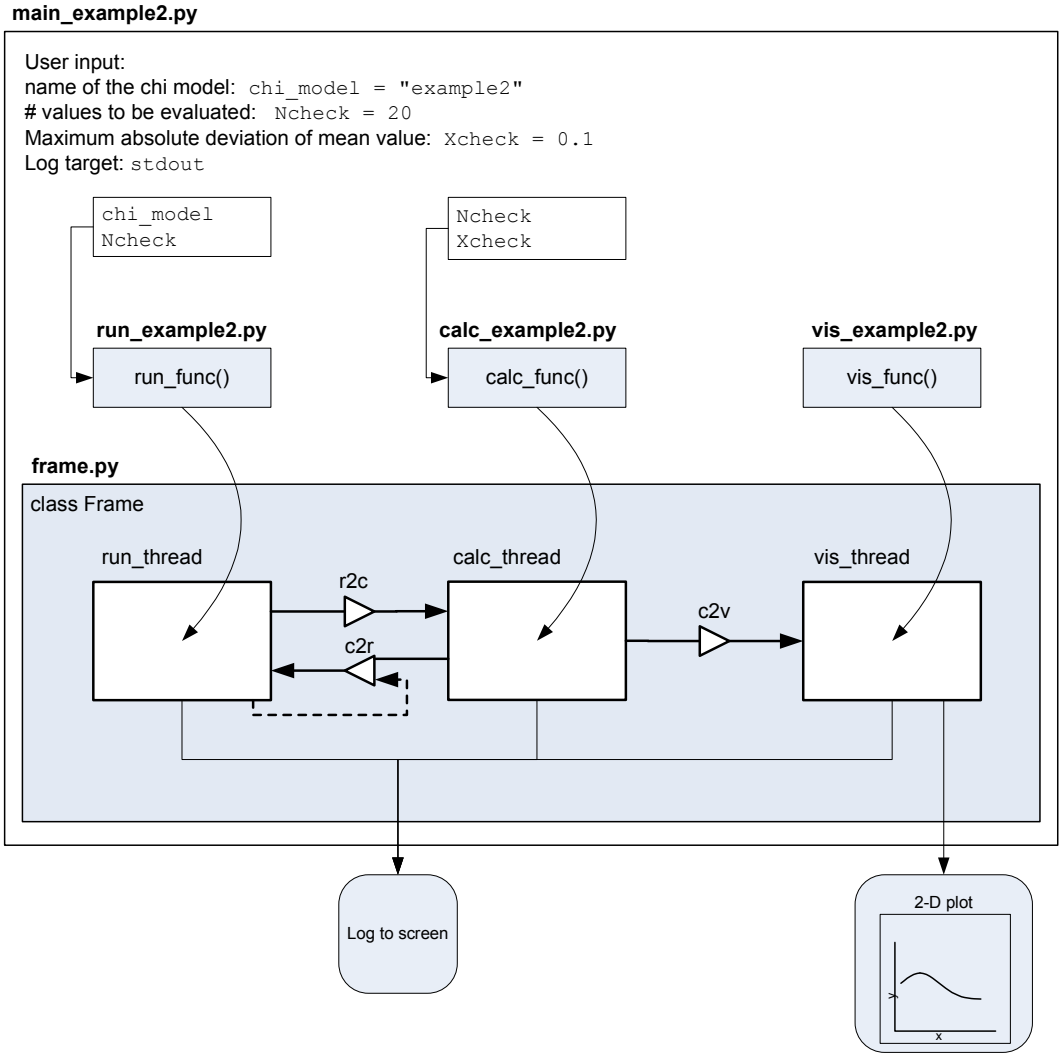

Figure 6.3: Structure of example 2

The difference between minimum and maximum value within the last updated mean values is determined and compared with the parameter $Xcheck$. When $Xcheck$ is larger then the difference, enough simulations are done and buffer `c2r` is closed. Otherwise, 0 is written in `c2r`, which means a next run can be started.
The test executed in this process is not meant for use in large simulation experiments. Many accurate statistical tests are available from the statistics literature.

Calculation output Every time a mean value is calculated, this is written to the buffer `c2v` for visualization. Besides that, information is written on the screen.

Visualization process Like in example 1, Gnuplot is used to create a diagram, this time with the script `vis_example2.py`. The only parameter that is plotted here is the mean value coming from the *calculation* process. Every time a new value is available in buffer `c2v` a plot is updated. On the x axis, the number of simulations is shown and on y axis the mean value of the χ-output is given.

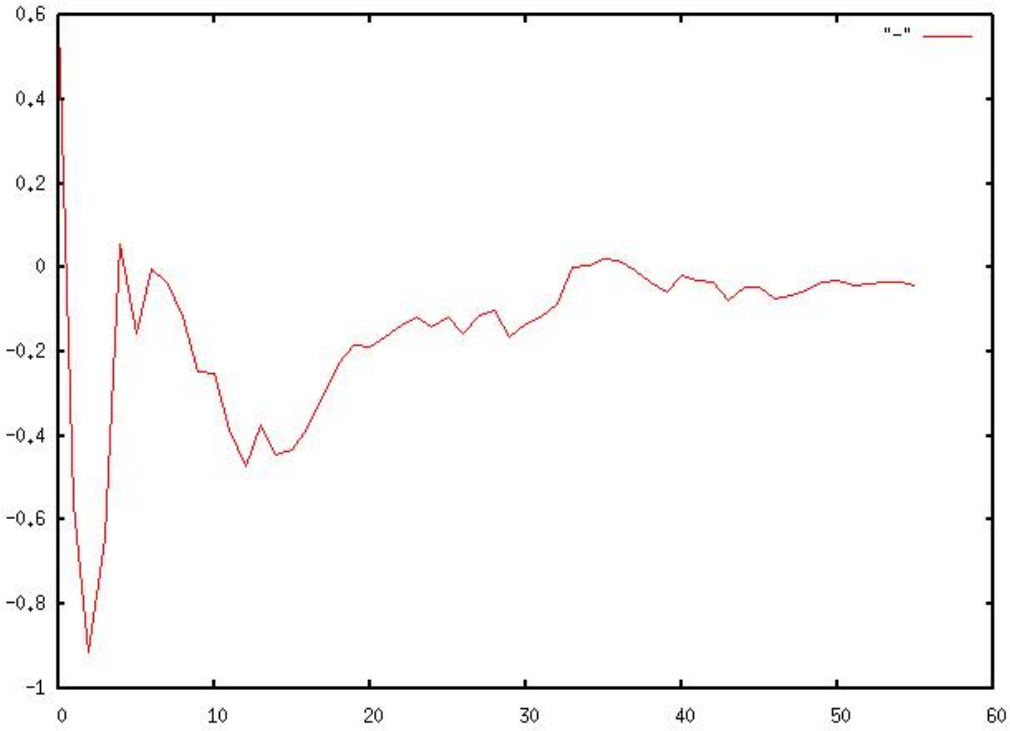

Figure 6.4: Plot of the mean value of example 2

Validation A plot of this experiment is given in Figure 6.4. It clearly indicates that the mean value is approaching zero when more simulations are done. This experiment is finished after 56 simulations. Because the problem is stochastic and no seed is used, another execution of the experiment will give different results, but comparable to the plot given here.

Chapter 7

Conclusions and Recommendations

In this chapter the development of the final product and the preceding investigations are discussed and conclusions related to the final product are drawn. After that, some extensions to the final product and recommendations for future development are presented.

7.1 Conclusions

In order to develop a structured method for performing (multiple) simulation experiments an overview has been made in Chapter 2 about the current methods for running simulations with χ-models, processing the output and visualization of the output and the desires within the SE Group. This has been performed by interviewing several students and staff members of the SE Group, who have experiences with multiple simulation experiments. On the basis of the survey that has been made, the functionality of the method to be designed has been created and presented in Chapter 3. Subsequently, an investigation on the internet into available software packages for performing (multiple) simulation experiments has been executed. This investigation, Section 4.1, has been compared with other concepts of structured methods, on which a final design has been chosen, see Section 4.4.

The final product, which is presented in Chapter 5, consists of a framework which is designed in Python and several Python scripts for running experiments, performing calculations and visualization of the output coming from simulation experiments. The framework consists of three threads, corresponding to a *run* process, a *calculation* process and a *visualization* process, and three buffers for data exchange between the processes. Within these processes functions that are specified in separate Python scripts can be imported. A powerful property of the framework is that all objects are accepted by the buffers, which leads to an unrestricted way of data exchange between processes.

To test the framework and to give the user an impression of the functions that have to be developed for the processes, two examples have been established which are presented in Chapter 6. In one example simulations are performed with a χ-model with two varying input parameters and a number of output values. In the other example a feedback construction

has been implemented for determining when the number of simulations is enough based on a statistical test. This construction uses the 'feedback' buffer from the calculation process to the running process for the creation of new simulations if the statistical test is not satisfied yet. Both examples proved that the framework works in a correct way.

All together, the final product is a software package which consists of a framework and some template scripts for the *run* process, the *calculation* process, the *visualization* process, and a main script that directs the framework and includes specification of parameters and the χ-model. Furthermore, a number of specific implementations of the process scripts are added, forming the beginning of a collection that still has to be extended considerably in the future. It is of importance to notice that this final product is just a start to a more structured method for processing output of (multiple) simulation experiments.

7.2 Recommendations

A software package has been developed for performing multiple simulation experiments with χ-models. Because this software package is just a start to a more structured method, there are still quite a number of things that can be mentioned for further improvement. Some extensions are listed below, noticing that most of these extensions are related to the construction of the functions `run_func()`, `calc_func()`, and `vis_func()` and are not deficiencies of the framework.

- For this software package a feedback construction has been developed in example 2. Unfortunately, this example is only suitable for simulation experiments with χ-models with no input parameters and only one output parameter. Therefore, it is useful to extend this feedback construction for χ-models with one or more input parameters and multiple output parameters.
- Another extension to the feedback construction is to program the construction in such a way that the *run* process and the *calculation* process work in parallel, in order to make the decisions simultaneous with the start of new jobs . The current construction restricts the *run* process and the *calculation* process to operate in succession (i.e. the *run* process has to wait for a decision from the *calculation* process before another simulation can be started), because otherwise simulation output of different input parameters can mix up which can lead into wrong calculations.
- A connection from the calculation process to Matlab would be a useful extension. With that, other calculations (which are easier with Matlab) can be performed. Moreover, Matlab has an extensive visualization environment for plotting data, so a connection from the visualization process to Matlab would also be a useful extension. When Matlab is used for both calculations and visualization, some difficulties can occur, because data for calculation and visualization has to be separated.
- The run functions that are provided are only suitable for simulations on the SE-racks. An extension would be that the software package is also applicable for the SE-Cluster. When performing simulations on the SE-Cluster, many simulations with different input parameters can be done simultaneously. Therefore it is of importance that the results of simulations with different input parameters are not going to mix up. This is yet implemented in the software package by including the input parameters

with every simulation output. The only adaption that has to be made is that the run process can run simulations on the SE-Cluster.

- In the visualization process Gnuplot is started up. Because the data flow to Gnuplot is via a pipe in a python script no mouse actions are possible on the figure that Gnuplot generates. Possibly, this can be solved by accessing Gnuplot in another way, avoiding the construction with piping that is used in the current implementation.
- One can understand that it may be useful that a figure is plotted after all simulations have been executed (instead of plotting during the experiment). This is not yet included in one of the examples, but this feature is not difficult to add to the visualization process.
- Plot more figures simultaneously, so the user can compare the results while simulations are performed.
- The robustness of the class Buffer can be improved. For instance when a process sends data via the buffer while it is closed, an exception should be raised to inform the user.
- It is useful to limit the buffers on the amount of data it can store. When large data sets are processed in the current tool, the capacity of the internal memory of the computer might be exceeded, with serious consequences for the experiment.
- A similar framework is developed for sequential approximate optimization experiments. [Jac04] Though the functionality of this framework is specific to the research area of optimization, it is useful to compare both frameworks with each other for further improvements.

These are all extensions the project team came up with. The user is encouraged to add his or her own scripts to extend the functionality of this software package.

Furthermore, there are some general recommendations that are important to mention.

- For the usage of this software package a SE-Wiki page has been established. On this page the user can download the software, the user manual and can find some demonstrations of the tool. For future development it could be useful that improvements and modifications to the tool will be collected and clarified on this page.
- Another recommendation is to test the software package on more different cases from within the SE Group.

Bibliography

[Jac04] J.H. Jacobs, L.P.F. Etman, F. van Keulen, and J.E. Rooda. Framework for sequential approximate optimazation. Technical report, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven University of Technology, Structural Optimization and Computational Mechanics Group, Faculty of Design, Engineering and Production, Delft, 2004.

[Pub05] Lionheart Publishing. *Simulation Software Survey.* http://www.lionhrtpub.com/orms/surveys/Simulation/Simulation.html (downloaded on 13-03-2007), 2005.

[Rij06] E.A.F. van de Rijt. *SE Cluster.* http://se.wtb.tue.nl/sewiki/wiki/secluster, 2006.

[Roo06] J.E. Rooda and J. Vervoort. Analysis of manufacturing systems using χ 1.0. Technical report, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven University of Technology, 2006.

Appendix A

Python script frame.py

In this appendix the Python script `frame.py` is presented. Notice that, the documentation of functions and parameters is not included in this script. The complete Python scripts with documentation can be found on the SE Wiki page.

frame.py

```
"""
This module defines a framework for performing S{chi}-simulation experiments
and has to be directed by the Python script main_*.py
"""

from threading import *

class Frame:
  """
  Framework for running S{chi} simulations and processing their output data.

  The frame consists of three threads (C{run_thread}, C{calc_thread} and
  C{vis_thread}) and three buffers (C{r2c}, C{c2r} and C{c2v}). In each thread
  a function is executed that is defined somewhere else by the user.
  The C{run_thread} is meant for running S{chi} simulations en collecting their
  output. The C{calc_thread} is meant for performing the desired calculations
  to the simulation output. The C{vis_thread} is meant for visualizing the
  obtained data. Data is exchanged between the threads by means of the buffers.
  """

  def __init__(self,log,run_func,run_pars,calc_func,calc_pars,
               vis_func,vis_pars):
    """
    Initialize a Frame instance by creating its threads and buffers.
    """
    self.r2c = Buffer()
    self.c2r = Buffer()
```

```
    self.c2v = Buffer()
    self.run_thread = Thread(target=run_func,args=(log,self.r2c,self.c2r,
                                                   run_pars))
    self.calc_thread = Thread(target=calc_func,args=(log,self.r2c,self.c2r,
                                                     self.c2v,calc_pars))
    self.vis_thread = Thread(target=vis_func,args=(log,self.c2v,vis_pars))

  def start(self):
    """
    Start the three threads of the frame, resulting in a concurrent execution
    of their corresponding functions.
    """
    self.run_thread.start()
    self.calc_thread.start()
    self.vis_thread.start()

  def wait(self):
    """
    Block until all three threads have finished the execution of their
    corresponding functions.
    """
    self.run_thread.join()
    self.calc_thread.join()
    self.vis_thread.join()



class Buffer:
  """
  FIFO buffer that takes arbitrary objects except <None>.
  """
  def __init__(self):
    self.__list = []
    self.__avail = Event()

  def write(self,obj):
    """
    Write data to buffer.

    @param obj: Data that has to be put into the buffer.
    @type obj: Object
    """
    self.__list.append(obj)
    self.__avail.set()

  def read(self):
    """
    Read data from buffer.
    """
    self.__avail.wait()
```

```
  obj = self.__list.pop(0)
  if len(self.__list) == 0:
    self.__avail.clear()
  return obj

def close(self):
  """
  Close the buffer. When the buffer is closed, the C{eob()} method returns
  C{True} as soon as the buffer is empty.
  """
  self.write(None)

def eob(self):
  """
  @return: C{True} iff the C{close()} method has been called and the buffer
  has become empty.
  @rtype : bool
  """
  self.__avail.wait()
  return self.__list[0] == None

def __repr__(self):
  return str(self.__list)
```

Appendix B

Python scripts Example 1

In this appendix the Python scripts `main_example1.py`, `run_example1.py`, `calc_example1.py`, and `vis_example1.py` are presented. Notice that, the documentation of functions and parameters is not included in these scripts. The complete Python scripts with documentation can be found on the SE Wiki page.

main_example1.py

```
# Functionality:
# Example 1 (multiple parameters) uses the following scripts:
# 'example1.chi'     - Simulation model which has two input parameters and
#                      gives one output value.
# 'run_example1.py'  - Runs Nrep simulations for all combinations of input
#                      parameters.
# 'calc_example1.py' - Calculation of the mean values and standard deviations
#                      of the chi output.
# 'vis_example1.py'  - Create a 3-D plot of the mean values with Gnuplot.


# ===== Define functions for each process ====================================
# The interfaces of the functions need to be as follow:
# - run_func(log,r2c,c2r,pars)
# - calc_func(log,r2c,c2r,c2v,pars)
# - vis_func(log,c2v,pars)

from run_example1 import run_func
from calc_example1 import calc_func
from vis_example1 import vis_func

# ===== Define 'global' parameters ===========================================
# The parameters defined here are used in the three threads: run, calc and vis

chi_model = "example1"          # Name of the chi model
P1 = range(6)                   # Input parameter 1 values
```

```
P2 = [0,2,4,5,8,10]             # Input parameter 2 values
Nrep = 5                        # Number of repetitions for each simulation
Noutp = 4                       # Number of output parameters
ioutp = 2                       # Buffer (output value of chi-model) to plot

std_file = "std_output.txt"     # File to write standard deviations

# Determine which parameters are provided in each thread:
# run_pars = [list_with_params]
# calc_pars = [list_with_params]
# vis_pars = [list_with_params]

run_pars = [chi_model,Nrep,[P1,P2]]
calc_pars = [Noutp,[P1,P2],std_file]
vis_pars = [ioutp,[P1,P2]]

# ===== Choose log target ================================================
# The log information can be written to text files or printed to the screen
# This target is used in all processes

import sys
##log = sys.stdout        # log information written to the screen
log = open("log.txt",'w') # log information written to text file

# ===== Build frame and start running ====================================
from frame import Frame
frame = Frame(log,run_func,run_pars,calc_func,calc_pars,vis_func,vis_pars)
frame.start()
frame.wait()
print "Experiment finished."

log.close()               # Closing log when it is written to a text file
                          # !!! When log information is written to the screen,
                          # please comment log.close()
```

run_example1.py

```
try:
  from subprocess import *     # Python 2.4
except ImportError:
  from subprocess23 import *   # Python 2.3

def run_func(log,r2c,c2r,pars):
  """
  Run process for running simulations with a Chi model for all combinations of
  input parameters. The process runs all simulations a fixed number of times.
  All required variables for running the the simulations is specified from the
  main file. This process is suitable for any number of different input
  parameters that is specified in the Chi model. This is possible by manually
  inserting statements for importing the values of the input parameters and
  manually insert loops to find all combinations of input parameters.
  The actual execution of the simulations is done automaticly.
  """

  log.write("RUN\tStarted.\n")

# ===== Import parameters ================================================
# chi_model - Name of chi model without extension
# Nrep      - Number of repetitions of each simulation
# P1        - Values of input parameter 1
# P2        - Values of input parameter 2

  chi_model = pars[0]
  Nrep = pars[1]
  P1 = pars[2][0]
  P2 = pars[2][1]

# ===== Initialize buffer c2r ============================================
# Find all combinations of input parameters
# modify for more than 2 input parameters by inserting more for-loops !!!

  ## combinations = [(i,) for i in P1 ]          # for 1 input parameter
  combinations = [(i,j) for i in P1 for j in P2] # for 2 input parameters
  for n in range(Nrep):
    for p1 in P1:
      for p2 in P2:
        c2r.write( (p1,p2) )
  c2r.close() # In this example there is no feedback from calculation process,
              # so no simulations will be added anymore -> buffer c2r closed

# ===== Start running simulations ========================================
  while not c2r.eob():

    # ----- Read tuple of input parameters -----------
    chi_input = c2r.read()
```

```
    # ----- Run a chi-simulation ---------------------
    chi_proc = Popen(['startmodel',chi_model]+[str(x) for x in chi_input]\
                     ,stdout=PIPE)
    chi_output = chi_proc.communicate()[0]

    # ----- Write results --------------------
    log.write("RUN\t%s\t%s"%(chi_input,chi_output))
    r2c.write((chi_input,chi_output))

  r2c.close()

  log.write("RUN\tFinished.\n")
```

calc_example1.py

```
try:
  from numpy import zeros,divide      # Python 2.4
except ImportError:
  from numarray import zeros,divide   # Python 2.3

def calc_func(log,r2c,c2r,c2v,pars):
  """
  Calculation process for determining mean values and standard deviations.

  This process calculates the mean values and standard deviations of the output
  values of the S{chi} model for a number of repetitions.
  """
  log.write("CALC\tStarted.\n")

# ===== Import parameters ================================================
# Noutp   -  Number of output values from the chi model
# p       -  Values of all input parameters in lists
# lenp    -  Tuple of lengths of input parameter ranges, which is used for
#            mapping the input parameters
# mapping -  From p, create mapping of input parameters to ranges
#            Example: p = [ [2, 4, 7] , [10, 20, 30] ] ->
#                     mapping = [ {2: 0, 4: 1, 7: 2} , {10: 0, 20: 1, 30: 2} ]

  Noutp = pars[0]
  p = pars[1]
  lenp = [len(p_i) for p_i in p]
  mapping = [dict(zip(p_i,range(len(p_i)))) for p_i in p]

  std_file = pars[2]
  f_std = open(std_file,'w')

# ===== Initialization ===================================================
# Initialize stds, means and count, which will update the while loop below
# stds  - Matrix for updating standard deviations
# means - Matrix for updating mean values
# count - Array for counting number of updates

  stds = zeros( tuple(lenp+[Noutp]) )
  means = zeros( tuple(lenp+[Noutp]) )
  count = zeros( tuple(lenp) )
  C = 0.0

# ===== Start calculations ===============================================
# Each time new values appear in the buffer r2c, stds and means are updated

  while not r2c.eob():

    # ----- Read from buffer ---------------
```

```
  # chi_input  - List of inputparameters
  # chi_output -  String of output values from chi model
  # i          -  From chi_input, create tuple of input parameters,
  #               according to mapping
  #               Example: p = [ [2, 4, 7] , [10, 20, 30] ] then
  #                        chi_input = [7, 20] ->  i = (2, 1)

  data = r2c.read()
  chi_input = data[0]
  chi_output = data[1]
  i = tuple([x[0][x[1]] for x in zip(mapping,chi_input)])
  output = [float(x) for x in chi_output.split()]

  count[i] = count[i] + 1.0
  C = count[i]

  # ----- Perform updates ----------------
  # Standard deviations are updated only when number of updates > 1
  # Mean values are updated

  if C <= 1:
      stds[i] = 0
  else:
      stds[i] = ((C-2)/(C-1))*stds[i] + ((output - means[i])**2)/C
  old = means[i]
  means[i] = ((C-1)/C)*means[i] + output/C

  # ----- Write results ------------------
  # Write info to log, updated means and updated standard deviations
  # to buffer c2v

  log.write("CALC:\tMean and Standard deviation updated for a simulation\
            with input %s\n"%str(chi_input))
  c2v.write(means)


c2v.close()

f_std.write(str(stds))
log.write("CALC\tstds written to file")
f_std.close()

log.write("CALC\tFinished.\n")
```

vis_example1.py

```
try:
  from subprocess import *     # Python 2.4
except ImportError:
  from subprocess23 import *   # Python 2.3

def vis_func(log,c2v,pars):
  """
  Visualization process for visualising the results with Gnuplot for example 1.

  Visualization process for making a 3D plot on the basis of 2 input parameters
  of the S{chi} model and 1 output parameter of S{chi} model.
  """

  log.write("VIS\tStarted.\n")

  # =========== Import parameters ===============================================
  ioutp = pars[0]
  P1 = pars[1][0]
  P2 = pars[1][1]

  # =========== Open Gnuplot ====================================================
  gnu_proc = Popen(['gnuplot'],stdin=PIPE)

  while not c2v.eob():

    means = c2v.read()

    #------ Convert outcome of the calculation process into plotting data ---
    data = ""
    for i1 in range(len(P1)):
      for i2 in range(len(P2)):
        meanoutcome = means[i1][i2][ioutp]
        data += "%f %f %f\n" % (P1[i1],P2[i2],meanoutcome)
      data += "\n"

    #------ Command for plotting data ---------------------------------------
    cmd = 'set pm3d\n' + \
          'set title "WIP levels buffer 3"\n' + \
          'set xlabel "input parameter P1" font "Helvetica,24"\n' + \
          'set ylabel "input parameter P2"\n' + \
          'set zlabel "Mean WIP levels"\n' + \
          'splot "-" with lines\n' + \
          data + \
          'end\n'

    gnu_proc.stdin.write(cmd)

    log.write("VIS:\tdata for drawing/updating graph sent to Gnuplot\n")
```

```
raw_input("First save your figure and then press ENTER...")
log.write("VIS\tFinished.\n")
```

Appendix C

Python scripts Example 2

In this appendix the Python scripts `main_example2.py`, `run_example2.py`, `calc_example2.py`, and `vis_example2.py` are presented. Notice that, the documentation of functions and parameters is not included in these scripts. The complete Python scripts with documentation can be found on the SE Wiki page.

main_example2.py

```
# Functionality:
# Example 2 (feedback) uses the following scripts:
# 'example2.chi'     - Simulation model which has no inputparameters and gives
#                      1 output value.
# 'run_example2.py'  - Runs simulations till the calculation process stops
# 'calc_example2.py' - Calculation of the mean value and check if this is
#                      approaching a constant value.
# 'vis_example2.py'  - Plotting mean value after every new calculation.


# ===== Import functions for each process ===================================
# The interfaces of the functions needs to be as follows:
# - run_func(log,r2c,c2r,pars):
# - calc_func(log,r2c,c2r,c2v,pars):
# - vis_func(log,c2v,pars):

from run_example2 import run_func
from calc_example2 import calc_func
from vis_example2 import vis_func

# ===== Define 'global' parameters ==========================================
# The parameters defined here are used in the three threads: run, calc and vis

chi_model = "example2"  # Name of the chi model
Ncheck = 20             # Number of values to be evaluated
Xcheck = 0.1            # Maximum absolute deviation of mean value
```

```
# Determine which parameters are provided in each thread:
# run_pars = [list_with_params]
# calc_pars = [list_with_params]
# vis_pars = [list_with_params]

run_pars = [chi_model,Ncheck]
calc_pars = [Ncheck,Xcheck]
vis_pars = []

# ===== Choose log target ============================================
# The log information can be written to text files or printed to the screen
# In all processes this target chosen here is used.

import sys
log = sys.stdout               # log information written to the screen
##log = open("log.txt",'w')    # log information written to text file

# ===== Build frame and start running ================================
from frame import Frame
frame = Frame(log,run_func,run_pars,calc_func,calc_pars,vis_func,vis_pars)
frame.start()
frame.wait()
print "Experiment finished."

##log.close()              # Closing log when it is written to a text file
                           # !!! When log information is written to the screen,
                           # please comment log.close()
```

run_example2.py

```
try:
  from subprocess import *     # Python 2.4
except ImportError:
  from subprocess23 import *   # Python 2.3

def run_func(log,r2c,c2r,pars):
  """
  Run process for running a S{chi} model without inputparameters. The number of
  simulations depends on a signal the process receives from the calculation
  process via the buffer c2r. The run process first performs a number of
  I{Ncheck} simulation runs. After that, it performs simulation runs as long
  as data appears on the buffer c2r.
  """

  log.write("RUN\tStarted.\n")

# ===== Import parameters ================================================
# chi_model - Name of chi model without extension
# Ncheck    - Number of mean values to be evaluated

  chi_model = pars[0]
  Ncheck = pars[1]

# ===== Initialize buffer c2r ============================================
  for n in range(Ncheck):
    c2r.write(0)

# ===== Start running simulations ========================================
  while not c2r.eob():
    c2r.read()
    # ----- Start new simulation -----------
    chi_proc = Popen(['startmodel',chi_model],stdout=PIPE)
    chi_output = chi_proc.communicate()[0]
    # ----- Write results ------------------
    log.write("RUN\tsample = %s\n"%chi_output.strip())
    r2c.write(chi_output)
  r2c.close()

  log.write("RUN\tFinished.\n")
```

calc_example2.py

```
def calc_func(log,r2c,c2r,c2v,pars):
  """
  Calculation process for determining if the experiment is statistically
  well grounded after each simulation.

  From the output of the S{chi} model a mean value is determined after each
  simulation. Subsequently a statistical test is performed that checks whether
  the maximum and the minimum value of the last I{Ncheck} simulation runs are
  within an interval of I{Xcheck}. When the statistical test is not satisfied,
  the calculation process sends a zero to the buffer c2r, which denotes that
  another simulation run has to be performed.
  """

  log.write("CALC\tStarted.\n")

# ===== Import parameters ================================================
# Ncheck - Number of values to be evaluated
# Xcheck - Maximum absolute deviation of mean value

  Ncheck = pars[0]
  Xcheck = pars[1]

# ===== Initialization ===================================================
# Initialize mean, means and n, which are updated in the while loop below
# mean  - Calculated mean value
# n     - Variable for counting number of updates
# means - List of last 'Ncheck' number of calculated means

  mean = 0
  n = 0
  means = [0]*Ncheck

# ===== Start calculations ===============================================
# Each time new values appear in the buffer r2c, mean and means are updated
# and the decision is made wheter to run a new simulation or not
  while not r2c.eob():

    # ----- Read from buffer ---------------
    data = float(r2c.read())

    # ----- Perform updates ----------------
    mean = (n*mean+data)/(n+1)
    means[n%Ncheck] = mean
    n = n + 1

    # ----- Write to buffer ----------------
    # Write info to log, updated mean to buffer c2v
    log.write("CALC\tmean = %f\n"%mean)
```

```
  c2v.write(mean)

  # ----- Decide if new simulation is started --------
  if n >= Ncheck:
    if max(means)-min(means) < Xcheck:
      c2r.close()
    else:
      c2r.write(0)
c2v.close()

log.write("CALC\tFinished.\n")
```

vis_example2.py

```
try:
  from subprocess import *     # Python 2.4
except ImportError:
  from subprocess23 import *   # Python 2.3

def vis_func(log,c2v,pars):
  """
  Visualization process for visualizing the results with Gnuplot of example 2.

  Visualization process for making a plot on the basis of 0 input parameters
  in the S{chi} model and 1 output parameter of the S{chi} model.
  """

  log.write("VIS\tStarted.\n")

  gnu_proc = Popen(['gnuplot'],stdin=PIPE,stderr=log)

  data = []
  while not c2v.eob():

    # ----- Read from buffer ---------------
    data.append(c2v.read())

    # ----- Create plot --------------------
    cmd = 'set xlabel "Number of runs"\n' + \
          'set ylabel "Mean value"\n' + \
          'plot "-" with lines\n' + \
          '\n'.join([str(x) for x in data]) + '\n' + \
          'end\n'
    gnu_proc.stdin.write(cmd)

  raw_input("First save picture and then press Enter...")

  log.write("VIS\tFinished.\n")
```