Development of a graphical user interface (GUI) for a microkinetic engine

Karel Henrard

Promotoren: prof. dr. ir. Bart Dhoedt, prof. dr. ir. Joris Thybaut
Begeleiders: Vinod Kumar, Bruno Volckaert

Masterproef ingediend tot het behalen van de academische graad van
Master in de toegepaste informatica

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. Daniël De Zutter

Vakgroep Chemische Proceskunde en Technische Chemie
Voorzitter: prof. dr. ir. Guy Marin

Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2010-2011
Development of a graphical user interface (GUI) for a microkinetic engine

Karel Henrard

Promotoren: prof. dr. ir. Bart Dhoedt, prof. dr. ir. Joris Thybaut
Begeleiders: Vinod Kumar, Bruno Volckaert

Masterproef ingediend tot het behalen van de academische graad van
Master in de toegepaste informatica

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. Daniël De Zutter

Vakgroep Chemische Proceskunde en Technische Chemie
Voorzitter: prof. dr. ir. Guy Marin

Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2010-2011
PREFACE

This thesis serves as proof that I, as a student, possess the necessary competences to succeed as a Master of applied information technology. The subject of this thesis is not necessarily a direct application of matter featured during our courses. In fact, developing a GUI using C# isn't something that was explicitly discussed. Perhaps one of the more important qualities of a graduating IT student is the ability to get acquainted with other technologies independently.

Another challenge was the fact that the domain in which the developed application will be used is very different from my own field of interest, which meant I couldn’t just follow my own ideas. It necessitated frequent communication with the prospective user to learn the desired functionality, an experience which I feel contributed towards the value of this thesis.

I’d like to express my gratitude to my mentor Vinod Kumar in particular, who assisted me in understanding the application’s domain. I’d also like to thank my promotors Bart Dhoedt and Joris Thybaut and my other mentor Bruno Volckaert for their involvement.

"De auteur(s) geeft(geven) de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef."

"The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In the case of any other use, the limitations of the copyright have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation."

Date 

Signature
**SUMMARY**

In this thesis we discuss the development of an application that serves as an extension to an existing microkinetic engine, which is a software application written in Fortran. The microkinetic engine or µKE uses custom input files filled with parameters and data to run properly. These input files were previously constructed manually, which requires a lot of effort from the user, and increases the chance of invalid files.

The developed application assists in generating these files automatically based on user input, reducing the effort and ensuring the validity of the files. The application has to be able to read and write these files correctly, as detailed in the thesis. It also provides the user with an interface that makes the process of specifying all the necessary data easier and more streamlined. To further reduce the effort on the user’s part, it integrates the existing µKE, making it accessible from within the application. All these topics are discussed in this thesis.

The final section features an example of how the application was tested. We conclude that the developed application meets the requirements.
# TABLE OF CONTENTS

PREFACE ......................................................................................................................................... I

SUMMARY .................................................................................................................................... II

TABLE OF CONTENTS ......................................................................................................... III

1 A MICROKINETIC ENGINE (µKE) .......................................................................................... 1

   1.1 Microkinetics ...................................................................................................................... 1

   1.2 A software approach ........................................................................................................... 1

2 INPUT FILES ........................................................................................................................... 2

   2.1 Overview ................................................................................................................................ 2

   2.2 Experimental data file ............................................................................................................ 2

   2.3 Input data file .......................................................................................................................... 3

3 GUI ............................................................................................................................................. 4

   3.1 Overview .................................................................................................................................. 4

   3.2 Functions .................................................................................................................................. 4

   3.3 Microsoft Visual Studio and C# ............................................................................................... 4

4 INPUT CLASSES ....................................................................................................................... 5

   4.1 Overview .................................................................................................................................. 5

   4.2 ExpData class ........................................................................................................................... 5

   4.3 InputData class ......................................................................................................................... 6

      4.3.1 File sections ....................................................................................................................... 6

      4.3.2 Header lines ......................................................................................................................... 6

      4.3.3 Data fields ............................................................................................................................ 6

      4.3.4 Section classes ...................................................................................................................... 7

      4.3.4.1 InputSize ............................................................................................................................ 7

      4.3.4.2 Job ....................................................................................................................................... 8

      4.3.4.3 ExpDataFile ......................................................................................................................... 9

      4.3.4.4 PlotOptions ......................................................................................................................... 9

      4.3.4.5 DAEParameters ................................................................................................................. 9

      4.3.4.6 PowellParameters .............................................................................................................. 10

      4.3.4.7 ReactorType ....................................................................................................................... 10

      4.3.4.8 CompoundInfo .................................................................................................................. 10
4.3.4.9 Catalytic ................................................................. 11
4.3.4.10 Intermediates......................................................... 11
4.3.4.11 Isothermal ............................................................ 11
4.3.4.12 ReactionNetwork .................................................. 12
4.3.4.13 InitialParameterValues ......................................... 13
4.3.4.14 ParameterEstimation ............................................. 13
4.3.4.15 RosenbrockParameters ......................................... 13
4.3.4.16 LevenbergParameters .......................................... 14
4.3.4.17 ParameterLimits .................................................. 14
4.3.4.18 WeightInfo .......................................................... 14
4.3.5 InputData class ......................................................... 14

5  GRAPHICAL USER INTERFACE (GUI) ........................................ 17
5.1 Overview ........................................................................ 17
5.2 Loading files ................................................................... 17
5.3 Input Data tab .............................................................. 19
5.4 Reaction Network tab ................................................... 20
  5.4.1 Tab area .................................................................. 20
  5.4.2 Edit reaction network form ....................................... 21
5.5 Parameters tab ............................................................ 24
5.6 Settings ......................................................................... 25
5.7 Handling the DataGridView ............................................ 26
  5.7.1 Overview ................................................................ 26
  5.7.2 Textbox cell .......................................................... 26
  5.7.3 Checkbox cell ........................................................ 28
  5.7.4 Dropdown list cell ................................................. 29

6  TESTING THE GUI ............................................................ 30

7  CONCLUSION ................................................................. 36

BIBLIOGRAPHY ................................................................. 37
1 A MICROKINETIC ENGINE ($\mu$KE)

1.1 Microkinetics

Chemical reactions take place at a certain rate or speed. Some reactions may take no more than a few milliseconds while others may take months to finish. The study of these reaction rates and how other factors influence them is the domain of chemical kinetics or reaction kinetics, like which reactants are used and how the use of a catalyst may increase the reaction rate. Chemical kinetics translate reactions into mathematical models to solve rate equations and find rate constants. The study of chemical kinetics in the microdomain is called microkinetics.

1.2 A software approach

One issue with mathematical models is that they can quickly become very complex. Analyzing kinetic models of a network of successive reactions means solving a set of differential algebraic equations and optimizing kinetic parameters. This would be a complicated and time consuming process without the aid of specialized computer software. For this purpose, the MicroKinetic Engine ($\mu$KE) was developed at the Laboratory of Chemical Technology at Ghent University.

The $\mu$KE is a software tool written in Fortran that automatically translates a reaction network into a mathematical model. It allows the user to specify a set of input parameters, such as the number of chemical compounds used, what type of reactor is used in the experiments, whether there's a catalyst involved and of course the reaction network itself, as well supply it with experimental data. The application uses the appropriate mathematical routines to solve differential algebraic equations and handle the experimental data. After these calculations it generates a set of output files and reports containing the results of the calculations. It’s a completely automated process that doesn’t require any interaction from the user once it’s started.
2 INPUT FILES

2.1 Overview

The µKE accepts two input files as arguments. These are, by their default names, the input data file and the experimental data file. The experimental data file contains data collected from prior experiments. In this case, we’re only interested in reading information from the file, not in how the file is generated. This is different for the input data file however, which is generated partly from information gathered from the experimental data file and partly from information specified by the user.

2.2 Experimental data file

The experimental data file is, by default, a .dat file. Figure 2.1 shows an example of an experimental data file, viewed in Excel.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat wt</td>
<td>TEMPERATURE</td>
<td>PRESSURE</td>
<td>C3H8</td>
<td>O2</td>
<td>H2O</td>
<td>CO2</td>
<td>INERT</td>
<td>C3H8</td>
<td>O2</td>
<td>H2O</td>
<td>CO2</td>
<td>INERT</td>
</tr>
<tr>
<td>2.00E-03</td>
<td>595.15</td>
<td>101325.9.5E+00</td>
<td>5.5E+01</td>
<td>0.0</td>
<td>1.00E+00</td>
<td>8.00E+00</td>
<td>5.50E+01</td>
<td>4.38E+01</td>
<td>3.28E+00</td>
<td>1.01E+03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.00E-03</td>
<td>595.15</td>
<td>101325.9.5E+00</td>
<td>5.5E+01</td>
<td>0.0</td>
<td>1.00E+00</td>
<td>8.00E+00</td>
<td>5.50E+01</td>
<td>4.38E+01</td>
<td>3.28E+00</td>
<td>1.01E+03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.00E-03</td>
<td>595.15</td>
<td>101325.9.5E+00</td>
<td>5.5E+01</td>
<td>0.0</td>
<td>1.00E+00</td>
<td>8.00E+00</td>
<td>5.50E+01</td>
<td>4.38E+01</td>
<td>3.28E+00</td>
<td>1.01E+03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.00E-03</td>
<td>595.15</td>
<td>101325.9.5E+00</td>
<td>5.5E+01</td>
<td>0.0</td>
<td>1.00E+00</td>
<td>8.00E+00</td>
<td>5.50E+01</td>
<td>4.38E+01</td>
<td>3.28E+00</td>
<td>1.01E+03</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1 Example of an experimental data file

Every row of data in this file corresponds to a single experiment. We can see that after the first three columns, the next five column headers are repeated once more. These five column headers represent chemical compounds used in the experiments. The data we’re interested in is the number of experiments in this file, which can be extracted by simply counting the rows, and the compounds used, which can be read from the column headers. This information will be used during the generation of the input file.

The actual data fields of the experimental data file are only relevant for the calculations inside the µKE itself. They will not be used in our GUI.
2.3 Input data file

The input data file essentially dictates how the µKE should behave. It contains all the input information required for the µKE to make calculations (except the experimental data discussed in the previous paragraph). By specifying this file as the sole argument when executing the µKE, all the calculations happen automatically without the user having to interfere or interact with the program. Figure 2.2 shows the input data file as a simple .txt text file.

![Input data file](image)

It's essential that this file adheres to certain rules to work well with the µKE, for example the format in which certain numerical data appears in the file. Without the use of a graphical user interface (GUI), this file has to be constructed manually by the user. This can be a tedious process and furthermore, it increases the risk of errors in the file. A simple typing error can endanger the validity of the file.

This is why the development of a GUI is important. The GUI still requires the user to enter the input information, but it relieves him of the responsibility of constructing a valid input data file by generating it automatically.

We will discuss the different sections of the input data file in great detail in a later paragraph.
3 GUI

3.1 Overview

The graphical user interface or GUI of the µKE is an application that allows a user to interact with the µKE in a user-friendly manner. Instead of manually constructing the input data file and calling the µKE from a command-line interface, the GUI presents a more streamlined application that feels familiar to any Windows user. It is developed in C# using Microsoft Visual Studio 2008.

3.2 Functions

The main function of the GUI is to provide an easy-to-use interface for the user to generate the input data file. To achieve this, the application has to be able to extract the necessary information from an existing experimental data file − an input data file can't exist without an experimental data file. Once this information is read, the application will present the user with the appropriate forms he will need to fill out. It can then write all this information to a correctly formatted text file − the actual input data file − ready to be used by the µKE.

Aside from constructing the input data file, the application also supports the opposite operation, i.e. loading an existing input data file. This enables the user to save an unfinished input file and recall it at a later time, or recall a finished case to make adjustments.

To further reduce the effort on the user's part, the application also allows the user to call the µKE Fortran code and µKE plotting code with the click of a button, using the currently opened input data file as an argument.

3.3 Microsoft Visual Studio and C#

The choice of programming platform and language was a straightforward one. C# is an object-oriented language that's aimed at developers creating Windows applications using the Microsoft .NET Framework. It's similar in syntax to languages like Java and C++ and inherits many of the best functions from C++ and Visual Basic, which makes it a popular choice with many developers. This popularity has in turn led to it being a well documented programming language, with many communities, code samples and publications available on the internet.

C# development and Visual Studio go hand in hand. Visual Studio is an excellent tool for creating graphical interfaces through the use of Windows Forms. It allows developers to quickly and efficiently create forms and windows using the design mode. Creating controls and placing them on a form is a matter of a few mouse clicks. Nearly every imaginable control is present in an extensive toolbox, from editable text boxes and image lists to save file dialogs and programmable data grids.
4 \hspace{1em} \textbf{INPUT CLASSES}

4.1 \hspace{1em} \textbf{Overview}

The GUI application can be roughly divided into two parts. On one hand there’s the interface itself, which the user interacts with, and on the other there’s the underlying code which handles the input files. This chapter is about the second part. The code handling the input files consists of two classes, one for the experimental data file and one for the input data file. An object of either class effectively represents an instance of the corresponding input file, containing the relevant information of that file.

4.2 \hspace{1em} \textbf{ExpData class}

The ExpData class handles the experimental data file. When an instance of this class is constructed, the path of the experimental data file is passed as an argument. It contains a public method Read which reads the necessary information from the file using a stream reader.

\begin{verbatim}
public void Read()
{
    using (StreamReader r = new StreamReader(file))
    {
        ...
    }
}
\end{verbatim}

As discussed in paragraph 2.2, the information we wish to read from this file includes the number of rows, which represents the number of experiments, and the number of compounds in the column headers and their names. Using the stream reader we can easily read a row from the experimental data file.

\begin{verbatim}
string line = r.ReadLine();
\end{verbatim}

Because the file consists of different columns, each data field inside the read string is separated by a tab character ‘\t’. This character can then be used to count the number of columns in the file and thus, knowing that the first three columns can be skipped, the number of compounds can be found. This number will be referred to as the number of output variables in the input data file. The compound names can be found by reading the first row, which contains the column headers.

Counting the number of rows can be done by incrementing a counter for every line read, until the end of the stream is reached.

\begin{verbatim}
int count = 0;
while (!r.EndOfStream)
{
    line = r.ReadLine();
    count++;
}
\end{verbatim}

At the of the while loop, the count variable will contain the number of experiments plus one (the header row).
4.3 InputData class

4.3.1 File sections

The input data file consists of several sections of data and information, typically separated by one or more descriptive lines of text (header lines). For each section, a member is created inside the InputData class. The type of each member corresponds to the specific class constructed for each section of the file (we will discuss these classes in a later paragraph). As every section requires at least a Read method to read data from the file and a Write method to save the data, they all inherit from the same base class InputMember. This class provides the methods mentioned above to read and write in the correct formats, but allows them to be overridden by the subclasses if necessary.

One issue was determining where each section starts and ends in the input data file. To do this, an understanding of how the µKE’s own code interprets the file is required first. One could assume the header lines are fixed lines of text that could be used to search the file, however this isn’t the case. The µKE runs through the file line by line and simply ignores each header line. This means that the header lines can contain any arbitrary text, as long as the number of lines preceding each section is known by the code. It also means the order in which the sections appear in the input data file is important and can’t be changed.

4.3.2 Header lines

This paragraph describes the coding solution used in our GUI application to handle the header lines. The first step is to make sure that each section class contains a header variable. The code sample shows the header for the first section of the file.

```csharp
private const string HEADER = "Number_of_Experiments\tNumber_of_Input_variables\tNumber_of_Output_variables\tNumber_of_Parameters";
```

This text will be written to the input data file when using the Write method. When reading from the file, we’re only interested in the number of lines in each header. This is found by counting the number of newline characters '\n' in the header variable – zero in this case – and incrementing by one.

4.3.3 Data fields

After the header, one or more lines of data can be found, containing several data fields. Each section has a certain number of data fields, for example the first section has four. Therefore each section class requires a variable to indicate this number.

```csharp
private const int NUMBER_OF_FIELDS = 4;
```

Note that this variable isn’t always a constant, as in some cases the number of data fields depends on information found in other sections.

The data fields are separated by a space ' ', a comma ',', a tab character '	' or a newline character '\n'. Using these characters and counting the number of data fields while reading from the file, we can find all the necessary data of each section. The fields are stored as an array of strings (see code sample below) in the base class and parsed into the correct data type in each section class.
private string[] data;

This array can be accessed by the subclasses through the public property in the code sample below. After the Read method in the base class is called, the subclass can use this property to get the data fields. Conversely, the subclass should set this property with the correct values before calling the base class’ Write method.

public string[] Data
{
    get { return data; }
    set { data = value; }
}

Once the last data field of a section is found, the next non-empty line of the file is the first header line of the next section. This solves the issue of determining where each section starts and ends.

### 4.3.4 Section classes

#### 4.3.4.1 InputSize

The InputSize class corresponds to the first section of the input data file. Figure 4.1 shows the file section.

![Figure 4.1 InputSize section](image)

The InputSize class contains a single header line and four integer data fields. The first three fields depend on the experimental data file as discussed in paragraph 4.2. The first field is the number of experiments or rows in the experimental data file (actually the user can choose to edit this number, as long as it’s not greater than the number of experiments in the experimental data file). The third field is the number of output variables found in paragraph 4.2, while the second field – the number of input variables – is the number of output variables plus three (the first three columns in the experimental data file that were skipped). We can see that the total number of columns in the experimental data file equals the number of input variables plus the number of output variables.

The fourth and final field contains the number of parameters that have to be calculated in the microkinetic model. The following code samples illustrate the points discussed above. Many of the sections classes in the following paragraphs have similar code sample which will therefore be omitted.

InputSize class members:

```csharp
private const int NUMBER_OF_FIELDS = 4; // Number of data fields
private int nExp = 0; // Number of experiments
private int nInVars = 0; // Number of input variables
private int nOutVars = 0; // Number of output variables
private int nPars = 0; // Number of parameters
private const string HEADER = "Number_of_Experiments\tNumber_of_Input_variables\tNumber_of_Output_variables\tNumber_of_Parameters"; // Header line
```
**InputSize Read method:**

```csharp
public override void Read(StreamReader r)
{
    base.Read(r, HEADER);
    // After all the data is read to a single array, store it in the correct variables
    nExp = int.Parse(Data[0]);
    nInVars = int.Parse(Data[1]);
    nOutVars = int.Parse(Data[2]);
    nPars = int.Parse(Data[3]);
}
```

**InputSize Write method**

```csharp
public override void Write(StreamWriter w)
{
    Data[0] = nExp.ToString();
    Data[1] = nInVars.ToString();
    base.Write(w, HEADER);
}
```

**4.3.4.2 Job**

![Figure 4.2 Job section](image_url)

The Job class contains two Boolean data fields (they’re either true or false). In the input data file, the data fields are written as binary integers, meaning a “1” should be translated to “true” and a “0” to “false”.

The first data field indicates the mode in which the µKE should be run. A “0” corresponds to running it in prediction or simulation mode, while a “1” indicates estimation mode. This mode determines whether or not certain routines should be used during calculations. The second field indicates whether the reaction network is entered manually or generated automatically. The current version of the µKE only supports manually entered networks however, which means the value of this field is not important at this time.
4.3.4.3 ExpDataFile

Experimental data File name (!!!!!!! WITHOUT ANY SPACE)
Copy of Propane_oxidation_data_mm.dat

Figure 4.3 ExpDataFile section

The ExpDataFile class contains the file name of the experimental data file. Note that because the µKE requires the input data file and experimental data file to be in the same folder, only the file name is required, not the path of the file.

4.3.4.4 PlotOptions

<table>
<thead>
<tr>
<th>PLOT PARITY (1=Yes/0-No)</th>
<th>PLOT RESIDUALS (1=Yes/0-No)</th>
<th>residual can only be plotted with Parities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Indicator if Parity is to be plotted (1) or not (0)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.4 PlotOptions section

As evident from Figure 4.4, the PlotOptions class encompasses a double section, with two headers and two data parts. This means that when reading or writing, the base class’ Read or Write method is called twice. All of the data in this double section relates to the plotting application of the µKE. This application uses the output results generated by the µKE to plot figures and graphs.

The first part contains two binary data fields that indicates whether or not the parity and residual graphs have to be plotted. Once this is established, the µKE plotting application still needs to know the compounds for which these graphs have to be plotted. This is covered in the second part. Remember that the compounds were found in the experimental data file (paragraph 2.2), so the number of data fields in the second part is the same as the number of output variables of the InputSize class. Each data field is again a binary integer, indicating whether or not the corresponding chemical compound has to be plotted.

4.3.4.5 DAEParameters

<table>
<thead>
<tr>
<th>DAE_solver_control_parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.0</td>
</tr>
<tr>
<td>-1.0</td>
</tr>
<tr>
<td>-1.0</td>
</tr>
<tr>
<td>-1.0</td>
</tr>
<tr>
<td>-1.0</td>
</tr>
</tbody>
</table>

Figure 4.5 DAEParameters section

DAE stands for differential algebraic equation. The subroutine used by the µKE to solve DAEs requires five parameters to be specified by the user. The data fields in this section represent these control parameters. As shown in Figure 4.5, they are double precision values.
4.3.4.6 PowellParameters

\[
\begin{array}{ccc}
-1.0 & -1.0 & -1.0 \\
\end{array}
\]

*Figure 4.6 PowellParameters section*

This section closely resembles the previous section. The data fields in this section are control parameters for another equation solving subroutine, which uses Powell’s hybrid algorithm.

4.3.4.7 ReactorType

\[
\begin{array}{c}
1 \\
0 \\
\end{array}
\]

*Figure 4.7 ReactorType section*

The ReactorType class contains information on which type of chemical reactor was used during the experiments. Two types are possible, plug flow reactor (PFR) or continuous stirred-tank reactor (CSTR). For both types there’s a binary data field, indicating whether or not the corresponding type of reactor was used.

4.3.4.8 CompoundInfo

\[
\begin{array}{cccc}
\text{Number of Compounds (Including intermediates + 1 for catalytic reactions for cat surface)} & 8 \\
\text{Number of Reactions} & 5 \\
\text{Name of Compounds} & \text{C}\text{H}\text{H} & \text{O}_2 & \text{H}_2\text{O} & \text{CO}_2 & \text{INERT} \\
\end{array}
\]

*Figure 4.8 CompoundInfo section*

Like the PlotOptions in paragraph 4.3.4.4, CompoundInfo is a section with two headers and data parts. This first part contains two integer data fields. The first one is the total number of compounds in the reaction network. This includes the compounds that were found in the experimental data file, intermediate compounds and a catalytic surface if there are catalytic reactions. The second integer is the number of reactions in the reaction network. In the second part we see the names of the compounds we found in the column headers of the experimental data file. They are stored as strings in the CompoundInfo class.
### 4.3.4.9 Catalytic

The Catalytic class contains just one field, which is either TRUE or FALSE in the input data file. If it's TRUE, we have a catalytic reaction which means a catalytic surface should be added to the compounds.

### 4.3.4.10 Intermediates

The intermediate compounds are chemical compounds that appear in the reaction network but aren't final products. The number of data fields (#int) in this class can be calculated from the number of output variables (#outvar) of the InputSize class, the total number of compounds (#comp) of the CompoundInfo class and the Boolean value in the Catalytic class as follows:

If catalytic: \( \text{#int} = \text{#comp} - \text{#outvar} - 1 \)

If non-catalytic: \( \text{#int} = \text{#comp} - \text{#outvar} \)

The values of the data fields are the names of the intermediate compounds and are stored as strings in the Intermediates class.

### 4.3.4.11 Isothermal

Like the Catalytic class, the Isothermal class contains a single data field that’s either TRUE or FALSE, indicating an isothermal or non-isothermal reaction network respectively. This value has an effect on the reaction network.
4.3.4.12 ReactionNetwork

The ReactionNetwork class covers the most complex section of the input data file, which contains the network of chemical reactions. The number of lines in the data part of this section can be found by looking at the number of reactions in the CompoundInfo class (see paragraph 4.3.4.8). In the example shown in Figure 4.12, there are five reactions.

Another class was constructed to handle the individual lines or reactions, suitably called the Reaction class. The ReactionNetwork class contains a member that's an array of the Reaction type. In our example, the array will have a length of five.

```java
private Reaction[] reactions;
```

We'll now take a closer look at the individual reactions.

![Figure 4.12 ReactionNetwork section](image)

The actual chemical reaction can be found after the first two numbers. There's always at least one reactant, which consists of a double precision number that represents the stoichiometric coefficient and a string with the reactant's name. When reading from the file, the code then checks the next data fields for reactants until it encounters the reaction arrow "->". By subtracting the number of reactants that were found before the arrow from the total number of compounds in the reaction, the number of products that will appear after the arrow can be found. Like reactants, they consist of a stoichiometric coefficient and a name.

The next part of the line after the products contains either one or two integers, separated by a comma in the figure. These integers are called the betas, and they refer to the parameters that apply to this reaction. Recall that the fourth field of the InputSize section held the number of parameters (paragraph 4.3.4.1). This means the betas should always be less than or equal to that number for a non-catalytic reaction network or less than or equal to one less than that number for a catalytic reaction network (an extra parameter is introduced for a catalytic network).

The final part contains the reaction orders. A reaction order consists of a double precision number which is the order itself, and a single character which is either 'N' or
'Y'. This character indicates whether or not this reaction order has to be estimated as a parameter during the µKE calculations, with 'N' indicating 'no' and 'Y' indicating 'yes'. These extra parameters to be estimated don't count towards the number of parameters in the InputSize class. Note that the number of reaction orders in a reaction should be the same as the number of reactants.

4.3.4.13 InitialParameterValues

<table>
<thead>
<tr>
<th>Initial parameter values</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.479603D-4</td>
<td>1.18540D-5</td>
<td>9.79901D-08</td>
<td>5.02320D-4</td>
<td>5.361608D-5</td>
<td>1.0</td>
<td>1.309635D8</td>
</tr>
<tr>
<td>3.186606D2</td>
<td>3.71109D9</td>
<td>1.445119D5</td>
<td>1.24806</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 4.14 InitialParameterValues section*

This section contains the initial values for the parameters that are used during calculations in the µKE. The number of values is equal to the number of parameters in the InputSize class. The order of the values is of particular importance, as it's these values the betas in the reactions refer to (so a beta value of 2 points to the second value of this section). The values are written in scientific notation. Note that the scientific notation used here features a 'D' instead of 'E' (D-scientific notation), which Fortran supports but C# doesn't. An additional method was therefore needed to make sure C# handles this notation correctly.

4.3.4.14 ParameterEstimation

```
Indicator if parameter is to be estimated(1) or not(0)
1 0 0 1 1 1 1 1 1 1 0
```

*Figure 4.15 ParameterEstimation section*

The user may not want every parameter to be estimated during a µKE run. The binary data fields in this section indicate which parameters have to be estimated by setting the corresponding field to '1' and which don't by setting them to '0'. The number of data fields is again equal to the number of parameters in the InputSize class, while the order of the fields is consistent with values of the InitialParameterValues section.

4.3.4.15 RosenbrockParameters

```
Rosenbrock control parameters(Maxeval,efrac)
50.0 -1.0
```

*Figure 4.16 RosenbrockParameters section*

Rosenbrock is a first optimization routine used during the parameter estimation in the µKE. The two data fields hold control parameters used in this routine.
4.3.4.16 LevenbergParameters

![Figure 4.17 LevenbergParameters section](image)

The Levenberg-Marquardt method is a more sophisticated optimization routine that continues to estimate the optimal values for the parameters. Like Rosenbrock, it requires control parameters. The 17 control parameters are stored in this section.

4.3.4.17 ParameterLimits

![Figure 4.18 ParameterLimits section](image)

The ParameterLimits section contains both the lower and the upper limits of the parameter values. The number of fields in both parts is equal to the number of parameters in the InputSize class plus the number of reaction orders that have to be estimated. The same scientific notation as used in the InitialParameterValues data fields is used here.

4.3.4.18 WeightInfo

![Figure 4.19 WeightInfo section](image)

The final section contains information on the weights used during μKE calculations. The top part holds a single Boolean value similar to the Catalytic and Isothermal classes, indicating whether or not the calculations will be weighted. The bottom part contains the weights itself, again in scientific notation. The number of weights is the same as the number of output variables in the InputSize class.

4.3.5 InputData class

Now we’ve discussed all the individual section classes, we can take a look at how it all comes together in the overarching InputData class. All the section classes appear as members of this class.

```java
private string file = ""; // The file path of the input data
private InputSize inputSizeObj = new InputSize();
private Job jobObj = new Job();
private ExpDataFile expDataFileObj = new ExpDataFile();
```
The Read method of this class calls the Read method of each section class in the correct order.

```csharp
public void Read()
{
    using (StreamReader r = new StreamReader(file))
    {
        inputSizeObj.Read(r);
        jobObj.Read(r);
        expDataFileObj.Read(r);
        plotOptionsObj.Read(r, inputSizeObj.NOutVars);
        daeParametersObj.Read(r);
        powellParametersObj.Read(r);
        reactorTypeObj.Read(r);
        compoundInfoObj.Read(r, inputSizeObj.NOutVars);
        catalyticObj.Read(r);
        int catIntermediate = catalyticObj.IsCatalytic ? 1 : 0; // Number of catalytic intermediates will be 1 if catalytic, 0 if non-catalytic
        intermediatesObj.Read(r, compoundInfoObj.NCompounds - inputSizeObj.NOutVars - catIntermediate);
        isothermalObj.Read(r);
        networkObj.Read(r, compoundInfoObj.NReactions);
        initValuesObj.Read(r, inputSizeObj.NPars);
        paramEstObj.Read(r, inputSizeObj.NPars);
        rosenbrockParametersObj.Read(r);
        levenbergParametersObj.Read(r);
        parameterLimitsObj.Read(r, inputSizeObj.NPars + networkObj.NOrderEstimations);
        weightInfoObj.Read(r, inputSizeObj.NOutVars);
    }
}
```

The Write method is similar.

```csharp
public void Write()
{
    using (StreamWriter w = new StreamWriter(file))
    {
        inputSizeObj.Write(w);
        jobObj.Write(w);
        expDataFileObj.Write(w);
        plotOptionsObj.Write(w, inputSizeObj.NOutVars);
    }
```
The modular nature of this class provides flexibility and allows it to easily keep up with any future additions or changes to the input data file. If a new section is added in the future, a new appropriate section class can be developed and added as a member of the InputData class. Its Read and Write methods should then be called in the right position of the InputData’s Read and Write methods. As long as the correct order of the sections is preserved, these changes won’t have any impact on the validity of the input data file.
5 GRAPHICAL USER INTERFACE (GUI)

5.1 Overview

The GUI is the interface part of our application. It aims to speed up the process of constructing a valid input data file by presenting a clean and user-friendly set of forms to the user. All the data specified by the user of the GUI is collected by the section classes discussed in the previous chapter. Constructing the input data file is done easily by using the InputData’s Write method. The figure below shows the start screen of the application.

![GUI start screen](image)

Figure 5.1 GUI start screen

5.2 Loading files

The GUI present the user with the option to either load an experimental data file or start from an already existing input data file. Regardless of which option is chosen, it loads up the main tab.

Loading an experimental data file creates an instance of the ExpData class discussed in paragraph 4.2. The information in that class is used to automatically initialize the relevant controls on the interface. All the other controls are set to their default values.

Figure 5.2 shows the main tab that’s loaded up after loading the experimental data file. Some information specific to the experimental data file is automatically filled in, like the file name, number of experiments and the number of input and output variables. The compound names in the bottom left section are taken from the column headers of the experimental data file. All the other controls show their default state.
Choosing to load from an existing input data file is similar. An instance of the InputData class is constructed and the Read method is called to retrieve the data fields from the file. One of those data fields is the file name of the experimental data file, which then in turn is loaded as described above. This situation is shown in Figure 5.3. All of the controls are now set to the correct values instead of the defaults, most noticeable by looking at the weights next to the compound names.
5.3 Input Data tab

Figures Figure 5.2 and Figure 5.3 show the input data tab. It consists of five areas: four grouped areas and a picture with a button on top. The top left area is the experimental data group, which contains information taken from the experimental data file. Some of this information can’t be changed by the user, like the number of input and output variables, as they depend on the number of columns in the file. The user can choose to lower the number of experiments, i.e. not use all of the rows in the file.

The options group features two checkboxes that handle the Boolean values of the Catalytic and Isothermal class. The dropdown list below them is a ComboBox that lists the possible reactor types.

The top right group has two mutually exclusive radio buttons to indicate that either prediction mode or estimation mode is selected. There’s also a checkbox to turn weighted calculations on or off. This is only applicable to estimation mode, in prediction mode the checkbox is turned off and disabled.

The weights group contains a more interesting control, the DataGridView control. The DataGridView is an extremely flexible tool for dynamically displaying all sorts of data in tabular format. Its cells support a variety of other controls as well, like checkboxes and dropdown lists. In this case, only the simplest cell, the textbox cell, is used in both columns. The first column contains the compounds from the experimental data file, which can’t be edited by the user. The second column, the weight values, can be changed depending on whether or not weights are enabled in the options group.

The code sample below shows how the DataGridView control is populated with values taken from the correct objects. The names of the variables are self-explanatory.

```csharp
dataWeights.Rows.Clear(); // Remove all the rows in the DataGridView
for (int i = 0; i < expDataObj.NOutVars; i++)
```
```csharp
{  DataGridViewRow row = new DataGridViewRow(); // Create a row
    DataGridViewCell cell = new DataGridViewTextBoxCell(); // Create a text box cell
    cell.Value = expDataObj.Componds[i]; // Add the compound name to the cell
    row.Cells.Add(cell); // Add cell to row
    cell = new DataGridViewTextBoxCell(); // Create new text box cell
    cell.Value = InputMember.DoubleToString(inputDataObj.WeightInfoObj.Weights[i]); // Add weight to cell
    row.Cells.Add(cell); // Add cell to row
    dataWeights.Rows.Add(row); // Add row to DataGridView
} // Add a row with the correct cells for every output variable
```

The final area contains the button to fire up the µKE with current input data file. Calling an executable file requires the use of the System.Diagnostics namespace.

```csharp
using System.Diagnostics;

Calling the executable:

```csharp
Process.Start(PROGRAM, "\" + inputDataObj.File + "\");
```

Where PROGRAM is constant string that contains the executable’s name and the second argument contains the input data file path.

## 5.4 Reaction Network tab

### 5.4.1 Tab area

The reaction network tab shows all the information relevant to the network of chemical reactions, i.e. the reactions, the betas and the reaction orders. Figure 5.4 shows the reaction network tab after loading an input data file containing five reactions. Note that these controls will be empty when starting from an experimental data file.
Three DataGridViews are present on this tab. The first one shows the reaction network in its current state. It shows the reaction number in the first column, followed by the reactants, reaction arrow and products of each reaction. This control can’t be edited, it only serves as a visual aid for the other two DataGridViews, so the user knows which reaction he’s working on. Editing the network itself requires a more complicated control, which is discussed later in this chapter.

The second DataGridView is a simple control with two columns of editable textboxes to set the betas. The second column may be invisible depending on the state of the isothermal checkbox. If the reaction network is isothermal, the µKE only needs a single beta per reaction, so the second column is invisible.

The reaction orders DataGridView is more interesting, as it contains checkbox cells. Recall from paragraph 4.3.4.12 that each reaction order is accompanied by a character indicating whether or not it has to be estimated. This role is fulfilled by the checkboxes in the columns with an “Est” header, which translate to a ‘Y’ or ‘N’ when ticked or unticked respectively. The first column contains an extra checkbox per reaction that indicates if it’s an elementary step. When switched on, this unticks all the estimation boxes on that line and sets the reaction orders to a default value, while also disabling them.

5.4.2 Edit reaction network form

Clicking on the “Edit reaction network” button brings up a new form that allows the user to manually input and edit the reaction network. It contains a DataGridView consisting of several textbox cells and dropdown lists. Figure 5.5 shows this form for the five reaction input data file.
This control contains the same network as the reaction network DataGridView of the previous paragraph, but fully editable (with the exception of the ID number, which auto-increments). The dropdown lists are of type DataGridViewComboBoxCell. These cells contain all of the possible compounds to be used in the reaction network, as shown in Figure 5.6. The first five are the compounds from the experimental data file. The two that follow are intermediates, while “M” is the catalyst, which only appears if the catalytic checkbox is ticked. The “New intermediate” option allows the user to add a new intermediate to the network, which is added to the dropdown lists. The final option is the reaction arrow.
The following code sample shows how the program fills the dropdown lists.

```csharp
/// <summary>
/// Set the compound name cell
/// </summary>
/// <param name="row">The current working row</param>
/// <param name="col">The column number of the cell</param>
private void SetCompoundCell(int row, int col)
{
    DataGridViewComboBoxCell cell = new DataGridViewComboBoxCell(); // Create cell
    cell.Items.Add("-"); // Add the default selection
    foreach (string s in inputDataObj.CompoundInfoObj.CompoundNames) // Add all the compounds
    {
        cell.Items.Add(s);
    }
    foreach (string s in inputDataObj.IntermediatesObj.IntermediateNames) // Add all the intermediates
    {
        cell.Items.Add(s);
    }
    if (inputDataObj.CatalyticObj.IsCatalytic) // Add the catalyst if catalytic
    {
        cell.Items.Add(inputDataObj.CatalyticObj.Catalyst); // Add the "catalyst" option
    }
    cell.Items.Add("->"); // Add the arrow
    cell.Value = cell.Items[0]; // Set initial value;
    dataNetwork.Rows[row].Cells[col] = cell; // Set the cell
}
```

This control allows the user to add and remove reactions and build a network from the ground up. When the user is satisfied, he/she can use the "Save changes" button to call a method that validates all of the fields. If the constructed network isn’t valid, this method will generate the appropriate error messages. If everything is correct, a new instance of the ReactionNetwork class is constructed using this data, which is then passed back to the main form.
5.5 Parameters tab

The third and final tab of the GUI handles the parameters and plot options. The first group contains a DataGridView that allows the user to set all the values relating to the InitialParameterValues, ParameterLimits and ParameterEstimation classes (see paragraphs 4.3.4.13, 4.3.4.17 and 4.3.4.14). There’s a line entry for every parameter, including the catalyst and any reaction orders to be estimated.

The second group contains the plot options. Two checkboxes indicate if the parities and residuals have to be plotted. The user can choose which of the output variables have to be plotted using the DataGridView below.

The rightmost group features two buttons to communicate with the µKE. The “Initial Guess” button is used to run the µKE in prediction mode, so no parameters will be estimated. It uses the initial values on the left and generates output files and plots according to the plot options. Once this initial run is completed, the user can change plot options and use the plot button to generate new plots according to these options without having to run the µKE itself again. Once the user makes changes to the parameter values however, the “Initial Guess” button has to be used again. Note that to run the µKE in estimation mode, the user has to use the button on the InputData tab. The reason it’s situated there is that the weights are relevant to estimation mode, so it’s more practical to have them on the same tab.
5.6 Settings

The only components from the input data file we haven’t discussed yet are the control parameters for the solver and optimization routines. These can be found under the “MKE Settings” menu option.

Both options call a similar looking form that allows the user to set the control parameters.

- The optimization settings:
• The solver settings:

![Solver settings form](image)

*Figure 5.10  Solver settings form*

5.7 Handling the DataGridView

5.7.1 Overview

With its many features and possibilities and due to its omnipresence in our application, the DataGridView and the way its data changes are handled is worth taking a closer look at. Three types of cells are used throughout the GUI, i.e. textbox cells, checkbox cells and dropdown list cells. We’ll discuss an example of each type in the upcoming paragraphs.

5.7.2 Textbox cell

One of the DataGridViews that uses textbox cells is the weights control. The first column is read-only and contains the names of the compounds. Next to the names is a column of editable weights. The user can edit these values, and the code has to detect the change and set the correct values in the WeightInfo class instance.
Once the user has finished editing a value by pressing enter or leaving the cell, the \textit{CellValueChanged} event occurs. The corresponding event handler can be used to catch this event.

\begin{verbatim}
private void dataWeights_CellValueChanged(object sender, DataGridViewCellEventArgs e) {
    ...
}
\end{verbatim}

The handler has two arguments, the \textit{sender}, which is the DataGridView object (dataWeights) itself, and a second argument which contains extra information like the column and row index of the edited cell. This information can be used to extract the value of the cell, cast it to the correct data type and store it in the correct object. This is illustrated by the following code sample. By casting the value inside a try statement, it's possible to filter out invalid input using the correct catch statement.

\begin{verbatim}
private void dataWeights_CellValueChanged(object sender, DataGridViewCellEventArgs e) {
    // Cast the sender to DataGridView
    DataGridView data = (DataGridView)sender;
    if (data.Rows.Count == 0 || e.ColumnIndex == 0) {
        return; // Leave the handler
    } // If there are no rows or the change is in the first column
    // Use the event arguments to find the edited cell and store the value
    string value = data.Rows[e.RowIndex].Cells[e.ColumnIndex].Value.ToString();
    try {
        // Cast this cell's content to a double
        double d = InputMember.StringToDouble(value);
        // Add this double to the input object
        inputDataObj.WeightInfoObj.Weights[e.RowIndex] = d;
    } catch (FormatException) {
        // Set the original value back
        data.Rows[e.RowIndex].Cells[e.ColumnIndex].Value = InputMember.DoubleToString(inputDataObj.WeightInfoObj.Weights[e.RowIndex])
        // Show error message
        MessageBox.Show("The field has invalid input", "Format error");
    }
}
\end{verbatim}
5.7.3 Checkbox cell

For the checkbox cell, we take a closer look at the parameters DataGridView on the third tab. This control has several columns of textbox cells, but the final column contains checkboxes. After ticking or unticking a checkbox and leaving the cell, the same CellValueChanged event of the previous paragraph occurs. However, with a checkbox cell, we generally don’t want the changes to be detected when we leave the cell but immediately after changing the state of the checkbox. We use another event, the CurrentCellDirtyStateChanged event, to achieve the desired effect.

The CurrentCellDirtyStateChanged event occurs when a cell has been edited but the changes haven’t been saved yet. For a textbox cell, this would happen every time a character was added to or removed from that cell. So when setting the number 123 in a cell, this event would occur three times. It’s obvious we don’t want to use textbox cells in this manner. With a checkbox cell, it would occur every time the box is ticked or unticked, which is exactly how we want a checkbox cell to react. The following code sample shows how this event is used.

```csharp
private void dataParameters_CurrentCellDirtyStateChanged(object sender, EventArgs e)
{
    // Create a checkbox cell
    DataGridViewCheckBoxCell cell = new DataGridViewCheckBoxCell();
    // Cast the sender to DataGridView
    DataGridView data = (DataGridView)sender;
    if (data.CurrentCell.GetType() == cell.GetType())
    {
        // Commit the data, this calls the CellValueChanged event
        data.CommitEdit(DataGridViewDataErrorContexts.Commit);
    }
    // Compare types to check if the edited cell is a checkbox
}
```

During this event, the code checks if the edited cell is a checkbox cell. In that case the changes to the data are committed, which in turn calls the CellValueChanged event and allows it to extract the data and store it. This is shown in the next code sample.

```csharp
private void dataParameters_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
    // Cast the sender to DataGridView
    DataGridView data = (DataGridView)sender;
    ...
    // Store the cell value in a variable
    string value = data.Rows[e.RowIndex].Cells[e.ColumnIndex].Value.ToString();
    ...
    try
    {
        switch (e.ColumnIndex)
        {
        ...
```
case 4: // The last column
    bool b = Convert.ToBoolean(value);
    inputDataObj.ParamEstObj.Indicators[e.RowIndex] = b;
    break;
} // Use the column index to distinguish between cases

catch (FormatException)
{
    ...
    // Show error message
    MessageBox.Show("The field contains invalid data.", "Format error");
} // Handle an incorrect format

5.7.4 Dropdown list cell

A dropdown list inside a DataGridView is achieved by using a DataGridViewComboBoxCell. This type of cell is used extensively in the form that allows the user to edit the reaction network. This form was introduced in paragraph 5.4.2. Data changes are handled in a similar manner as the checkbox cells. Every time a different option is picked from the dropdown list, a CurrentCellDirtyStateChanged event occurs. In the event handler of this event, we check if the current cell is a dropdown list and commit the data changes if it is.

private void dataNetwork_CurrentCellDirtyStateChanged(object sender, EventArgs e)
{
    // Cast the sender to DataGridView
    DataGridView data = (DataGridView)sender;
    // Create cell of ComboBox type
    DataGridViewComboBoxCell cell = new DataGridViewComboBoxCell();
    if (data.CurrentCell.GetType() == cell.GetType())
    {
        data.CommitEdit(DataGridViewDataErrorContexts.Commit);
    } // Compare types and commit if applicable
}

The actual changes are again handled in a CellValueChanged event handler. This event handler is similar to the other cell types, and is therefore omitted from this section.
6 TESTING THE GUI

Testing the functions of the GUI was achieved by using existing input data files with available output results. Loading existing files proves the application can successfully read and interpret the files. Once all the data is collected in the interface, the user can edit, add or remove information as necessary. Saving the data and offering it as input to the µKE shows that the application is able to generate valid input data files. Finally, comparing the output results generated using our newly created input data files with the existing output results confirms that the µKE handles our files correctly.

In this chapter we discuss one example of an input data file used for testing.

![Figure 6.1 Existing input data file](image)

Loading this file fills the controls on the interface with the correct values. This is shown in the next set of figures. Note that this file is manually created and contains more information than strictly necessary. Both the µKE and the GUI ignore the extra data fields. The figures show that the data presented in the forms corresponds to the data in the existing input data file, indicating that the application can read files correctly.
Figure 6.2  Input data tab

Figure 6.3  Reaction network tab
Using the save option generates the following input data file. The data in the file is consistent with the data on the forms, indicating the application can write input data files correctly.
Figure 6.5 Generated input data file

Using the “Run µKE” or “Initial Guess” button calls the µKE with the recently generated input data file as parameter. The µKE shows its feedback via the command window.

Figure 6.6 Feedback of the µKE
When all the calculations are done, a message is shown informing us which compounds have been plotted. A comparison of the generated plots with the already existing plots is shown, with the existing plots on the left and the generated plots on the right.

Figure 6.7  Existing C3H8 parity plot

Figure 6.8  Generated C3H8 parity plot

Figure 6.9  Existing CO2 parity plot

Figure 6.10  Generated CO2 parity plot

Figure 6.11  Existing H2O parity plot

Figure 6.12  Generated H2O parity plot
All of the generated plots perfectly match the already existing plots. This indicates that the generated input data file was correctly processed by the µKE. From this test we conclude that the application can successfully load data files, create valid data files and correctly automate the µKE process. Obviously a single test only proves one case, so this test was repeated with a variety of different input data files to ensure the application behaves correctly in other cases.
7 CONCLUSION

A GUI for the existing µKE application has been developed. The user no longer has to worry about creating the correct input files and calling the µKE with the correct parameters. The developed application integrates the existing µKE so that they act as a single program from the user’s point of view. The use of specialised classes for each section of the input file means any future additions can be integrated easily, without affecting the consistency of the file.
BIBLIOGRAPHY


J. SHARP, Microsoft Visual C# 20008 Step by Step, Microsoft Press, 2008