Applied novel software development methodology for process engineering application

Afshin Sadrieh

School of Engineering and Information Technology
Murdoch University

2017
Declaration

I declare that this thesis is my own account of my research and contains, as its main content, work which has not previously been submitted for a degree at any tertiary education Institution.

To the best of my knowledge and belief this thesis contains no material previously published by any other person except where due acknowledgment has been made.

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university.

Afshin Sadrieh

Date: 21/04/2017
Abstract

Chemical processes are nonlinear continuous/discrete dynamic systems that are subject to considerable uncertainties and variations during their design and operation. These systems are designed to operate at an economically optimal steady-state. However, minor changes in process parameters’ values might cause deviations and elicit dynamic responses from processes. Controllability—defined as the ability of holding a process within a specified operating regime and the controllability assessment of each given process system—should be taken into account during the system design phase. This emphasises the necessity of effective software tools that could assist process engineers in their controllability evaluation.

Although there are few multipurpose tools available for this task, developing software tools for controllability analysis is a tedious and sophisticated undertaking. It involves elaboration from multiple disciplines, and the requirements of controllability assessments are so vast that it is almost impossible to create general software that covers all controllability measures and cases.

This thesis aims to systematically tackle the challenge of developing practical and high-quality software tools for controllability problems while reducing the required time and effort, regardless of the size and scale of the controllability problem.

Domain-specific language (DSL) methodology is proposed for this purpose. DSLs are programming languages designed to address the programming problems of a specific domain. Therefore, well-designed DSLs are simple, easy to use and capable of solving any problem defined in their domains. Based on DSL methodology, this study proposes a four-element framework to partition the software system into decoupled elements, and discusses the design and implementation steps of each element as well as communication between elements. The superiority of the
developed methodology based on DSL is compared with traditional programming techniques for controllability assessment of various case studies.

Essentially, the major advantage of the proposed methodology is the performance of the software product. Performance measures used in this study are total time to develop (TD) the software tool and its modifiability. Total time and effort to implement and use the result products presents up to five times improvement. Moreover, the result product’s modifiability is assessed by applying modifications, which also demonstrates up to five times improvement. All measures are tested on continuous stirred-tank reaction (CSTR) and forced-circulation evaporator (FCE) case studies.

In conclusion, this study significantly contributes to two fields. The first is DSL, since this thesis studies different types of DSLs and evaluates their applications in the controllability analysis. The second is the controllability evaluation, since this study examines a new methodology for software development in controllability assessment.
Acknowledgments

It is my great pleasure to acknowledge those who have help this thesis possible. Firstly, I would like to express my gratitude to my supervisor Professor Parisa A. Bahri for her advice, constant support, and comments throughout the course of this research. I also thank her for the interest she has shown in this study and in my technical development. I am proud to have had her as my supervisor.

I would like to acknowledge the financial support by school of Engineering and Information Technology. Also, I would like to thank the staff and students of this school for their support.

Completion of this research would have not been possible without the support, understanding and encouragement of my dearest family, especially my brother Arash, friends and colleagues. This thesis is also a result of moral supports by my friends, especially Tim, Ferhan, Mathew, Travis.

Last but not least, I would like to express my deepest gratitude to my beloved wife, Shiva, for her endless support, motivation and understanding throughout this thesis.
To my dearest in the world

My precious parents

My wonderful brother

My lovely Shiva
Publications


Table of Contents

Chapter 1 ........................................................................................................................................... 1
  1.1 Background .............................................................................................................................. 1
  1.2 Scope of the study .................................................................................................................... 2
  1.3 Structure of the thesis .............................................................................................................. 3
References ........................................................................................................................................ 5
Chapter 2 ........................................................................................................................................... 6
  2.1 Introduction ............................................................................................................................... 6
  2.2 Software tools .......................................................................................................................... 7
    2.2.1 General-purpose optimisation and process synthesis tools ............................................. 7
    2.2.2 Tools that enable balancing and flow-sheet simulation .................................................. 10
    2.2.3 Tools for process integration .......................................................................................... 15
  2.3 Software quality metrics ......................................................................................................... 19
  2.4 Software development challenges ........................................................................................... 20
    2.4.1 Software failure quantitative report .............................................................................. 21
  2.5 DSL definition ......................................................................................................................... 24
    2.5.1 Definition ....................................................................................................................... 24
    2.5.2 Formal definition ............................................................................................................ 25
    2.5.3 Different types of DSLs ................................................................................................. 29
      2.5.3.1 External DSLs ........................................................................................................ 29
      2.5.3.2 Internal DSLs ......................................................................................................... 29
    2.5.4 DSL elements .................................................................................................................. 29
      2.5.4.1 Semantic model ...................................................................................................... 29
      2.5.4.2 Code generation .................................................................................................... 31
    2.5.5 Language workbenches ..................................................................................................... 34
    2.5.6 Advantages and disadvantages of DSLs ......................................................................... 35
      2.5.7 Advantages ................................................................................................................... 36
      2.5.8 Disadvantages ............................................................................................................ 40
    2.5.9 DSL examples ................................................................................................................. 41
      2.5.9.1 Software ................................................................................................................. 41
      2.5.9.2 Hardware ............................................................................................................... 42
      2.5.9.3 Multimedia and office products .............................................................................. 42
      2.5.9.4 Telecommunications ............................................................................................ 42

Chapter 7 ............................................................................................................................. 141
  7.1 Introduction ............................................................................................................. 141
  7.2 Case study definition ............................................................................................. 142
    7.2.1 FCE ................................................................................................................. 143
  7.3 The Non-linear model ............................................................................................. 144
    7.3.1 FCE variables .................................................................................................. 144
    7.3.2 Mass balance equations ................................................................................... 146
      7.3.2.1 Liquid mass balance .................................................................................... 146
      7.3.2.2 Solute mass balance .................................................................................... 146
      7.3.2.3 Vapour mass balance .................................................................................. 146
    7.3.3 Energy balance equations ................................................................................ 147
      7.3.3.1 Liquid energy balance ................................................................................. 147
      7.3.3.2 Evaporator steam jacket .............................................................................. 148
      7.3.3.3 Condenser .................................................................................................... 149
    7.3.4 Linear model and state-space model ............................................................. 150
    7.3.5 Input disturbance ............................................................................................. 151
  7.4 Developing the DSL framework ............................................................................. 151
  7.5 Implementation ....................................................................................................... 152
    7.5.1 Solver .............................................................................................................. 153
    7.5.2 DSL definition ................................................................................................. 153
    7.5.3 Semantic model implementation ..................................................................... 154
    7.5.4 Code generator implementation ...................................................................... 154
    7.5.5 Code generator controller implementation ...................................................... 155
  7.6 Results and discussion ............................................................................................ 155
    7.6.1 FCE RGA index .............................................................................................. 158
    7.6.2 FCE RGA index with input disturbance ......................................................... 158
    7.6.3 Time and effort ................................................................................................ 160
    7.6.4 Effort comparison with CSTR case study ....................................................... 161
  7.7 Conclusion .............................................................................................................. 161
  References ........................................................................................................................... 162
Chapter 8 ............................................................................................................................. 163
  8.1 Summary of research contribution .......................................................................... 163
  8.2 Future work ............................................................................................................. 165
  Appendix 1 ......................................................................................................................... 167
List of Tables

Table 2.5-1. Domain-specific languages versus general programming languages .......... 28
Table 2.5-2. Language workbenches currently in use ......................................................... 34
Table 2.5-3. DSL benefits and their sources ................................................................. 37
Table 4.2-1 CSTR variables .............................................................................................. 79
Table 4.3-1. DSL objects .................................................................................................... 87
Table 5.5-1. CSTR Configurations RGA Index (R) ......................................................... 111
Table 5.5-2. Software products development specifications ................................................. 112
Table 6.2-1. Controllability assessment of CSTR case study including the Uncertainty feature. ................................................................................................................................. 121
Table 6.3-1. Various states of Auto-Splitter ....................................................................... 124
Table 6.4-1. Controllability assessment of CSTR case study including the Superstructure feature ................................................................................................................................. 130
Table 6.4-2. RGA Index of all configuration with ±%2 uncertainty ................................... 134
Table 6.4-3. Average of RGA index drift from 0% uncertainty for each percent of uncertainty value in input temperature for every configuration ...................................... 135
Table 6.5-1. Effort required to apply modification in each case ........................................ 136
Table 7.3-1. FCE Model Variables and their steady-state values ....................................... 145
Table 7.6-1. DSL defined objects ....................................................................................... 157
Table 7.6-2. Disturbance values, steady-state values and RGA index of FCE ..................... 158
Table 7.6-3. Results of evaluating FCE for RGA index. Disturbance applied to variables. 159
Table 7.6-4. Design and development comparison of FCE and CSTR case studies .......... 161
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>Application programming interface</td>
</tr>
<tr>
<td>ACP</td>
<td>Aspen Custom Modeler</td>
</tr>
<tr>
<td>BARON</td>
<td>Branch-and-Reduce Optimization Navigator</td>
</tr>
<tr>
<td>BDL</td>
<td>Behaviour Description Language</td>
</tr>
<tr>
<td>CC</td>
<td>Composite Curves</td>
</tr>
<tr>
<td>CPI</td>
<td>Centre for Process Integration</td>
</tr>
<tr>
<td>CPU</td>
<td>Central processing unit</td>
</tr>
<tr>
<td>CSTR</td>
<td>Continuous stirred-tank reactor</td>
</tr>
<tr>
<td>DETS</td>
<td>Distributed effluent treatment systems</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic link library</td>
</tr>
<tr>
<td>DOF</td>
<td>Degree of freedom</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-specific language</td>
</tr>
<tr>
<td>DXF</td>
<td>Drawing exchange format</td>
</tr>
<tr>
<td>ER</td>
<td>Emission Reduction</td>
</tr>
<tr>
<td>ETVP</td>
<td>Energy Targets for Various Processes</td>
</tr>
<tr>
<td>FCE</td>
<td>Forced-circulation evaporator</td>
</tr>
<tr>
<td>GORRP</td>
<td>Graph, Object, Relationship, Roll and Property</td>
</tr>
<tr>
<td>GPL</td>
<td>General-purpose language</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>HLA</td>
<td>High-level Analysis</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated development environment</td>
</tr>
<tr>
<td>IT</td>
<td>Internet technology</td>
</tr>
<tr>
<td>LP</td>
<td>Linear programming</td>
</tr>
<tr>
<td>MILP</td>
<td>Mixed-integer linear programming</td>
</tr>
<tr>
<td>MIMO</td>
<td>Multi-input multi-output</td>
</tr>
<tr>
<td>MINLP</td>
<td>Mixed-integer non-linear programming</td>
</tr>
<tr>
<td>MPS</td>
<td>Meta Programming System</td>
</tr>
<tr>
<td>NLP</td>
<td>Non-linear programming</td>
</tr>
<tr>
<td>OPPRR</td>
<td>Object Property Port Relationship Role</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional–integral–derivative</td>
</tr>
<tr>
<td>PIT</td>
<td>Process integration technology</td>
</tr>
<tr>
<td>PSE</td>
<td>Process system engineering</td>
</tr>
<tr>
<td>QCP</td>
<td>Quadratically constrained programming</td>
</tr>
<tr>
<td>RGA</td>
<td>Relative gain array</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>RHP</td>
<td>Right-half-plane</td>
</tr>
<tr>
<td>RPN</td>
<td>Robust performance number</td>
</tr>
<tr>
<td>SISO</td>
<td>Single-input single-output</td>
</tr>
<tr>
<td>TCS</td>
<td>Teta Consultancy Service</td>
</tr>
<tr>
<td>TD</td>
<td>Time to develop</td>
</tr>
<tr>
<td>TS</td>
<td>Total sites</td>
</tr>
<tr>
<td>UML</td>
<td>Unified modelling language</td>
</tr>
<tr>
<td>USO</td>
<td>Utility Systems</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Mark-up Language</td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td><strong>Actuator</strong></td>
<td>In a closed-loop control system, that part of the final control element that translates the controller output into an action by the control device.</td>
</tr>
<tr>
<td><strong>Closed loop</strong></td>
<td>A controller in automatic mode. The controller is driven by an error generated from reference minus the plant output or feedback signal. The controller then calculates the corrective action to apply to the plant to drive the plant output to match the reference, i.e. make the error zero.</td>
</tr>
<tr>
<td><strong>Controlled variable (CV)</strong></td>
<td>What you are trying to control.</td>
</tr>
<tr>
<td><strong>Left half plane</strong></td>
<td>In the complex plane, the half with negative real numbers. For continuous systems, the poles in the left half plane are stable.</td>
</tr>
<tr>
<td><strong>Linear System</strong></td>
<td>A system is said to be linear if it is scalable and obeys the principle of superposition.</td>
</tr>
<tr>
<td><strong>Manipulated Variable</strong></td>
<td>The physical variable that is adjusted by the controller output (via the actuator) to cause the desired change in process output.</td>
</tr>
<tr>
<td><strong>Pole</strong></td>
<td>A root of the transfer function denominator polynomial.</td>
</tr>
<tr>
<td><strong>Right-half-plane (RHP)</strong></td>
<td>In the complex plane, the half with positive real numbers. For continuous systems, the poles in the RHP are not stable.</td>
</tr>
<tr>
<td><strong>Single-input single-output (SISO)</strong></td>
<td>A system with one input signal and one output signal.</td>
</tr>
<tr>
<td><strong>State space</strong></td>
<td>systems the notation is common where the system is a matrix of first-order differential equations, is the state vector, is the system output and the system input.</td>
</tr>
<tr>
<td><strong>Steady-state</strong></td>
<td>The state reached once all of the dynamics of a system response have decayed.</td>
</tr>
<tr>
<td><strong>Transfer function</strong></td>
<td>A convenient way of representing a dynamic system. Rather than working with differential equations, which is difficult even for simple systems, they are converted into the Laplace domain. A transfer function is the ratio of output signal to input signal, so that the output of a system can be calculated by multiplying the input signal by the transfer function. A transfer function usually has numerator and denominator polynomials in powers of s, with the numerator polynomial being of an equal or lower order than the denominator.</td>
</tr>
<tr>
<td><strong>Zero</strong></td>
<td>A root of the transfer function numerator polynomial. Can be real or complex conjugate.</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.4-1. Portion of successful, failed and challenging software projects in 2004 22
Figure 2.4-2. Portion of succeeded, failed and challenging software projects (updated) 22
Figure 2.4-3. Issues with IT projects from business stockholders’ point of view 23
Figure 2.4-4. Contribution of factors that turn a software project into a failure 23
Figure 2.5-1. Semantic Model generated from DSL script 30
Figure 2.5-2. Simple state machine 32
Figure 4.2-1 CSTR unit operation 73
Figure 4.2-2 Mixer unit operation 80
Figure 4.2-3 Splitter unit operation 81
Figure 4.3-1 MetaEdit+ meta-model tool. a) Defining entities and types. b) Defining relations and bindings 85
Figure 4.3-2. MetaEdit+ symbol definition tool for the CSTR unit 85
Figure 4.3-3. MetaEdit+ Process system modeled using meta-model. 88
Figure 4.4-1. a) Petri-net based modeling tool elements. b) Clustering example 90
Figure 4.4-2. Differential equation representation using Petri-net based modeling tool 91
Figure 5.2-1. CSTR case study parallel configuration. 96
Figure 5.3-1 Controllability assessment requirements 98
Figure 5.4-1 General structure of the DSL framework 99
Figure 5.4-2 Semantic Model Class diagram 101
Figure 5.4-3. Sequence Diagram for DSL framework 104
Figure 5.4-4. Petri-net software structure. 106
Figure 5.4-5 Petri-net model of CSTR flow-sheet 107
Figure 5.4-6 screenshot of Result page of GPL product 108
Figure 5.5-1 Different CSTR configurations 110
Figure 6.2-1. Modified Elements for uncertainty implementation 119
Figure 6.2-2. Uncertainty symbol in the DSL framework 120
Figure 6.2-3. CSTR case study applied to DSL framework including uncertainty 122
Figure 6.3-1. Modified Elements for superstructure Implementation in DSL framework (hashed) 123
Figure 6.3-2. Auto-splitter designed in DSL framework 125
Figure 6.3-3 Modified Class diagram to include superstructure in GPL product. 127
Figure 6.4-1. CSTR case study applied to DSL framework including superstructure feature 129
Figure 6.4-2. Best process configuration from the controllability point of view 131
Figure 7.2-1 FCE Flow-sheet 143
Figure 7.6-1. FCE Case study modeled by DSL framework 156
Figure 7.6-2. Time required to develop the DSL framework 160
1.1 Background

Advances in computer hardware and software technologies have affected a wide range of scientific and industrial applications. Breakthroughs in computer hardware have introduced mega computational power, offering solutions that were not feasible decades ago [1]. Along with hardware improvements, noticeable software advances have also emerged. In fact, to use hardware power, it is necessary to provide efficient and practical software tools [1]. This is achieved by evolving a number of different programming paradigms and methodologies. As software development methods and tools improve, creating software requires less hardware knowledge; as a consequence, developers can spend more time on the problem itself rather than dealing with programming technicalities.

Among the domains influenced by computer breakthroughs is process engineering. A variety of software tools have been developed to help process engineers to model, simulate and control process systems. Although these tools assist process engineering, they also introduce some software development challenges. Arguably,
they present a steep learning curve for process engineers and expensive development and maintenance procedures for software developers. In fact, these issues are usually introduced because of the vast domain that these tools cover. In order to overcome these challenges, one approach is to limit the size of this domain. However, due to the wide scope of process engineering problems, it is not economically justifiable to develop individual software tools for each specific scope of process engineering using traditional software development methods.

Therefore, the aim of this research is to tackle the challenge of developing software tools for process engineering’s specific problems. For this purpose, the controllability assessment problems area was selected due to its narrow and well-defined boundaries.

In this study, domain-specific language (DSL), a new methodology for software development, is considered as a guideline to boost the development and maintenance efficiency of the software tool for controllability analysis. Although the DSL concept has recently been redefined and implemented, it has been available since the early stages of the computer science era. However, DSLs have only recently become more popular due to the fact that it bridges the gap between programmers and domain experts. Regardless of the vast numbers of successful DSL application use cases, this methodology has not been used in the process engineering domain. Thus, this study’s main emphasis is on implementing and evaluating DSL usage in process engineering, which makes this study novel in its field. The rest of this chapter explains the scope of the study and the thesis structure.

1.2 Scope of the study

As mentioned earlier, in the course of this study, controllability of chemical processes is the domain in which the new software tools were developed. More specifically, two controllability use cases were studied in this research:

1- analysing the combination of two CSTRs

2- analysing the forced-circulation evaporator (FCE) plant.
Three different software solutions were examined for these case studies using:

1- general-purpose language (GPL)
2- DSL
3- process engineering software tool.

The resulting tools were analysed against two software development metrics:

1- development time and effort
2- modifiability.

The scope of the research is defined as how these software products have been designed, developed and maintained for the controllability case studies.

1.3 Structure of the thesis

This thesis is organised into eight chapters. Chapter 2 is a detailed literature review that explains three major software tools for the process engineering domain and reviews the popular tools in three major software-developing tools; it also explains the software development challenges facing software development for process engineering. It then discusses DSL methodology and the history, definition and structure of DSLs. DSL structure is also used in chapters 5–7 to implement DSL tools. Chapter 2 reviews the advantages and disadvantages of DSLs and the currently available DSL tools. Chapter 3 is dedicated to controllability reviews, and illustrates the various tools and indices that are being used to evaluate a process system design from a controllability perspective; this chapter concludes that software tools are essential for controllability assessment to assist process designers in evaluating controllability of process system designs at the design stage.

The next three chapters address the design, development and modification of three software tools using three different methodologies. Chapter 4 outlines the case study model for which these tools have been defined. The model used to describe this case study in each of these three methods is then illustrated using each of these three
methods. Chapter 5 presents the design and development procedure for each of these software tools, as well as the effort required in each approach. As the next step of evaluating the software development methodologies, Chapter 6 applies one similar modification to each of the three software tools. It presents the affected parts as well as the effort required to implement this modification.

Chapter 7, however, deals with a more realistic/industrial controllability analysis case study. This chapter initially analyses the case study and presents the process model before explaining the procedure used to develop the DSL tools. This chapter also illustrates the effect of the case study size on the DSL development process.

Finally, Chapter 8 summarises the results and suggests future research opportunities.
References

Chapter 2

Software tools literature review

2.1 Introduction

The use of software tools is widespread within the process engineering domain. From performing tedious computational tasks for simulation and optimisation purposes to real-time controlling of unit operations in a plant, software tools are essential to any modern process engineering task. However, it has always been difficult to develop software tools for this scope. This difficulty has various causes, including the complexity and size of process engineering tasks. Similar to most other software tool challenges, the communication between software developers and process engineers is an important factor that, if not handled well, could cause fundamental problems.

This chapter presents a detailed overview of currently used software tools for process engineering, followed by a review of the causes and roots of software project failure. It then defines and explains DSLs.
2.2 Software tools

In this section, software tools that are used in the scope of process engineering are categorised into three major groups:

1- general-purpose optimisation and process synthesis tools
2- tools that enable balancing and flow-sheet simulation
3- tools for process integration.

A few software tools from each group are reviewed below, and the advantages and disadvantages of each are then discussed.

2.2.1 General-purpose optimisation and process synthesis tools

Problems in process synthesis pose a unique challenge that can be solved using the mathematical programming method. Mathematical models must be accurate to avoid errors and inaccurate solutions [1]. When designing a mathematical model, its various aspects must be factored in to give as accurate a representation as possible.

Mathematical models are generally used to describe and show how process optimisation problems operate. More specifically, they link variables and decisions (e.g., the rate of flow for a stream or the heat quantity resulting from pressurised steam).

Mathematical programming endeavours to assign relevant figures for these variables in order to fully accommodate the constraints, while also ensuring that a specific objective function that is related to these variables is either minimised or maximised.

A search space is specified as the variables’ limit and the objective function to define the ideal points. The type of variables (continuous or integer) as well as the type of constrains (linear or non-linear) determine the type of mathematical model. Typical models are linear programming (LP), Mixed-integer linear programming (MILP), non-linear programming (NLP) and Mixed-integer non-linear programming (MINLP) [2].
A high-level modelling system that is commonly used for optimisation and mathematical programming is the General Algebraic Modelling System (GAMS). The major optimisation problems that GAMS is created to model are mixed-integer, linear and non-linear problems. Massive models that require maintenance and are highly adaptable to novel situations can be built using a GAMS system [3]. GAMS caters for difficult-to-process and multi-scale modelling applications. It has proven to be very handy because it allows the user to manage massive, complex and unique problems that might require many corrections in order to create a high-level model. GAMS makes it possible to easily alternate and change the formulation of models; it also provides the option of changing from linear to non-linear models. In GAMS, like other algebraic description languages, algebraic statements define models, making it easy for both humans and machines to read. GAMS is fully capable of formulating models, irrespective of problem classes. GAMS delivers a variety of solver tools for each type of model [3]: for LP, quadratically constrained programming (QCP) and mixed-integer programming, it uses CPLEX; DICOPT is used for working out MINLP models. In the case of global optimisation of smooth constrained problems with continuous variables, or a combination of continuous and discrete variables, OQNLP is used and, finally, the Branch-and-Reduce Optimization Navigator (BARON) is used in the case of global optimality and to calculate non-convex optimisation problems. GAMS is proven to be efficient and is used to model different processes. Examples of such processes are the synthesis of water networks [4][5], production of biogas [6], mass exchange networks [7][8] and supply networks of biomass [9].

A second tool that can be used to calculate integer, linear and quadratic programming problems is LINDO [10]. It is very easy to work with LINDO because of its highly interactive modelling environment. LINDO optimisation engines can be connected to any existing application. What sets LINDO apart is its speed and ability to calculate large-scale integer and linear models. LINDO improves its portability by providing a dynamic link library (DLL) [11], which gives end users the ability to easily use it in Windows-based applications developed by C/C++/C#, Visual Basic or other languages that can carry DLL calls. The latest release of
LINDO is significantly improved—for example, it features global optimisation options, improved interfaces for other systems such as MATrixLABoratory (MATLAB) and Java, greater non-linear capabilities and better performance with integer and linear problems.

LINDO has released the leading full-featured solver DLL, which provides general integer and non-linear optimisation capabilities as LINDO API. This feature allows developers to integrate a general-purpose solver into any custom applications. The LINDO application program interface (API) allows the user to perform a variety of functions for non-linear problems, such as formulating, solving and modifying models. In order to make use of the non-linear capabilities, one needs the non-linear licence option. In order to come up with authentic global solutions for the non-linear and mixed-integer programs, the global solver uses different bounding and range-reduction techniques. LindoSystem has developed the LP, MINLP, LP and MILP optimisation programmer LINGO [10]. El-Halwagi demonstrates examples of process models that have been solved using LINGO [12]. The twin pair LINDO and LINGO are tools with wide mathematical programming formulation use and functionality, and with widespread necessity in process synthesis [13][14][15][16][17].

MATLAB is a programming language used for numerical computation [18]. It enables the user to work out problems systematically and efficiently, giving graphical solutions and results with minimum effort. MATLAB permits matrix manipulation, easily plots functions, implements algorithms and transfers data. One of its main advantages is the wide range of functions that it provides. These functions offer tried and tested solutions to a host of technical tasks, such as median, averages and standard deviation. Graphical results can be shown on output machines that run MATLAB. These machines can typically allow the user to have a constructive interaction with the graphs thanks to its graphical user interface (GUI)—because of this, the MATLAB programmer is able to create and build quality data analysis programs that even the most junior-level user can operate. MATLAB’s built-in routines permit function maximisation and minimisation [19].
MATLAB is currently being used to solve a dynamic range of process system engineering (PSE) problems. PSE problems found in modelling and parts simulation that have used MATLAB include the multivariable statistical process control that processes fault detections and diagnoses.

The free substitute of MATLAB is either OCTAVE [20] or SCILAB [21]. As opposed to MATLAB, these software tools are free, and provide the same problem-solving capabilities as MATLAB. They are both matrix-based programming languages. The main areas in which they are similar to MATLAB are the following: fully supported complex numbers, expanding via user-defined functions and modelling statements, capable math functions and vast function libraries.

2.2.2 Tools that enable balancing and flow-sheet simulation

The tools that are most commonly used for energy-saving analysis are flow-sheet simulation and balancing reconciliation tools. Design values, measurements and mathematical models are the basis of generating energy and mass models. The role of these tools can never be underestimated. They are key factors in both economic and technical activities where decisions around designing and planning procedures for real equipment are concerned. Klemes et al. [22] have provided one of the premier flow-sheet simulation overviews. The reconciliation technology and balancing data validation comprises a combination of processes that are integrated into software tools. The chief way to optimise and monitor industrial processes is with process data reconciliation [23]. Moreover, process data reconciliation can be used to monitor and give component diagnoses, ensure the online calibration of instrumentation and allow for condition-based maintenance. Heyen and Kalitventzeff [24] have established some of the process’s most important aims:

- regulating the accuracy of post-processing in case of unmeasured and measured data
- finding and recalibrating errors and deviations in measurement data that satisfies the constraints
- determining the precision of measured and unmeasured data.
AspenOne [25] is another application package that allows process engineers to design and simulate industrial processes. AspenOne provides optimum manufacturing and engineering operations. It is a comprehensive system that provides solution to various aspects of the manufacturing plant, such as control design, safety and economy estimation. This potentially offers significant cost-saving suggestions. One of the key aspects of AspenOne is the Aspen Plus. When conceptual designs and performance are being explored, the Aspen Plus is considered the process modelling tool of choice [26]. This highly efficient system has modules that map unit operations as its base. The process flow-sheet can be formed by connecting unit operations using material, heat streams and work streams. This software system incorporates an immense database of component and phase equilibrium data for conventional chemicals, electrolytes, solids and polymers. The default data on this database is provided by the United States National Institute of Standards and Technology, which is the governing body that provides the best available experimental property data.

The Aspen Plus works hand-in-hand with the Aspen Tech cost analysis software [27] and their heat exchanger design software [28][29]. Both of these applications are crucial. Used widely in the academic and industrial simulation process, the Aspen Plus is a key tool in the simulation of biomass gasification systems [26], cogeneration plants [30] and waste incineration processes [31][32].

Hyprotech has designed software known as HYSYS, which is used for steady-state and dynamic process simulation. HYSYS incorporates various tools that can be used for the following:

- evaluating the physical properties and the equilibrium phase of liquid–vapour
- planning and balancing heat and materials, optimising gas and oil procedures
- process equipment.

Aspen bought the rights to HYSYS in May 2002; Honeywell then purchased the rights from them and later rebranded the software as UniSim Design [33][34]. The
software of both firms share similarities in terms of their interface and application modes. The software provides the following:

- the subroutines used to determine the behaviour of varying plant equipment
- the GUI responsible for accepting the specifications of the display and case results
- the physical properties libraries of massive numbers of chemical species.

The chemical process can be explained using unit operations that are connected and linked by process streams. The software calculates and factors all the equations (energy, mass, equilibrium, etc.) and considers all their unique design parameters. The programs thus have a solid foundation of tough technology that has been used effectively in the gas and oil industry. A further advantage of Aspen HYSYS and UniSim is their flexible and interactive process modelling tool, which allows process engineers to carry out various operational processes and improvements, enabling the plants to continue sustainably [31]. The fluid package provides all the essential information for calculations of component pure flash and physical properties. To gain an accurate processing model during process simulation, the first step is to choose an appropriate thermodynamic model. If the chosen thermodynamic model does not accurately represent the physical model, optimised equipment determination, configuration and operation is no longer useful.

HYSYS is designed based on the minimum user data entry. The main input factors required for streams are the pressure, temperature and flowrate of the stream [7]. The flow-sheet within the HYSYS simulation environment can be alternated by the process engineer to reach the desired output. Typical uses of HYSYS and UniSim are as follows:

- comparing thermal coupling in crude distillation and delayed coking units [35]
- processing various biodiesel procedures [36]
- outlining and simulating reactors in order for the chlorination of acetone in its gaseous phase [37]
• optimising and simulating extractive distillation using H₂O as a solvent [38].

The Aspen Custom Modeler (ACP) is another software tool from the Aspen suite that provides flow-sheet design capabilities. It offers a GUI to design an interactive process flow-sheet that process engineers can then test and simulate. All Aspen products are integrated into one framework and can be used together to perform the best results.

gPROMS is an innovative process modelling package for the process industries [39]. Within a flow-sheeting environment, gPROMS is able to provide custom modelling capabilities. gPROMS’ process modelling, simulation and optimisation abilities are used to produce precise analytical data for decision support in product and process novelty, operation and design. gPROMS is a critical tool that has been used for the optimisation and modelling of many different applications [40][41][42][43].

Complex units can be optimised during any part of the process in gPROMS. gPROMS makes the development and maintenance of quality assurances easier by synchronising all the graphs and text views. This software has the ability to gain an overview of the entire lifecycle of the process. The gPROMS unit family comprises several products, and the gPROMS Model Builder is one of these. Within the Model Builder environment, engineering specialists can achieve custom modelling, process engineers can produce graphical flow-sheets and process operators can explicitly run the model. The foundation of gPROMS is an equation-oriented modelling system; this system is used for creating, validating and running the first-principles models of a flow-sheeting framework.

Another popular, often-used chemical process simulation software is CHEMCAD [44]. CHEMCAD consists of libraries made up of chemical components and methods with a thermodynamic nature, which enable the steady-state simulation of continuous chemical procedures. Improvements are constantly being made to CHEMCAD, and the latest permit the dynamic analysis of flow-sheets. Other CHEMCAD functions include training operators, operability checkouts and loop tuning (proportional, integral and derivative). In order to observe the behaviour of a
process that is exposed to various fluctuations such as temperature, pressure and product-rate changes, models for non-standard unit operations are used for the simulation. One of the key uses of CHEMCAD has been in the pollution-prevention sector [45]. CHEMCAD is also a fundamental tool that has been deployed to observe the integration of hydrogen production procedure notions [46][47].

A GUI displays and modifies the plant’s flow-sheet. A pull-down menu option allows for modification of the feed stream and unit operation. The potential for errors has been considered when entering data—in cases where data is missing or over-specified, an error-checking mechanism will offer the best solution. The interactive interface allows separate unit operations to proceed uniquely towards the flow-sheet for fast reference and studies. CHEMCAD also offers extra modules, including the following:

- CHEMCAD-BATCH, which permits the simulation of batch distillation processes
- CHEMCAD-THERM, which is used for planning and ranking shell and tube heat exchangers
- CHEMCAD-REACS, which allows stirred-tank reactors to be simulated.

CONVERT comes as part of the CHEMCAD software package and is a subroutine. The program makes a process flow diagram that can be translated to a batch of drawing exchange format files (DXF), which can be integrated into AutoCAD® software routines. Jecha et al. [48] have demonstrated the advantages of CHEMCAD in their work.

A computer simulation system that process engineers across different industries—from petroleum, solids processing, chemical and polymer to natural gas industries—is PRO/II [49]. PRO/II is complex, constituting several thermodynamic property prediction processes, a mega chemical component library and futuristic unit operation simulation software. PRO/II is a fully integrated system capable of calculating energy and mass balances for steady-state processes. PRO/II offers a
wide range of user aids to ease its learning curve. PRO/II’s simulation is used across the following fields:

- evaluating, recording and meeting environmental standards
- evaluating alternate plant configurations
- troubleshooting and bottleneck removal from processes plant
- increasing the profitability of a plant through better monitoring and optimisation
- revamping and modernising existing plants.

A high-pressure distillation simulation is an example of PRO/II usage [50].

2.2.3 Tools for process integration

Power generation and processing industries use many technologies. Among these, process integration technology (PIT) is a scientifically proven, regularly updated 30-year-old technology. Many educational and research institutes have contributed to PIT’s development—among them is the Centre for Process Integration (CPI) at the University of Manchester, which pioneered this technology in 1980. Pinch technology and heat and water integration are other names for PIT.

CPI has designed and developed many proportional–integral–derivative (PID) software tools, such as STAR [51], WATER [52], SPRINT [53] and WORK [54].

Energy recovery systems design uses SPRINT. Many individual processes on a particular site use this software tool for utilities selection and the associated design of heat exchangers. This tool offers the designer two choices: Retrofit and New Design. Retrofit is the default option. SPRINT applications can thus be summarised as follows:

1. choice, optimisation and loading of process utilities
2. new heat exchanger network design
3. designing retrofits with fewer modifications in heat exchanger networks
4. making new designs and designing retrofits for multiple utilities
5. designing networks interactively
6. simple model network simulation
7. minimum energy consumption planning
8. networks optimisation
9. providing network operability.

STAR is used for designing site utilities and co-generation systems and for analysing site processes, steam turbines, boiler house, gas turbines, cooling systems and local fired heaters. STAR provides special graphs called Composite Curves (CC). CC, Balanced Grand CC and total site profiles are the main outputs of STAR. When changes occur in site operations, such as a change of energy tools, STAR can be used to decrease the energy costs.

Sometimes, strict environmental regulations require low flue-gas emission, which can again be investigated using STAR. The input data files for STAR and SPRINT are the same.

STAR uses the following tools:

1. Utility Systems Optimisation (USO)
2. High-level Analysis (HLA)
3. Energy Targets for Various Processes (ETVP)
4. Total Sites (TS)
5. Emission Reduction (ER).

The applications and their uses are published in [29].

HEAT-Int [55] and SITE-int [56] are the upgraded versions of STAR and SPRINT. WORK is used for designing low-temperature processes. Costly refrigeration systems are used for heat rejection. The operational costs of refrigeration systems are the major costs in these systems. WORK can be used in complex refrigeration
systems analysis, cascade systems analysis, mixed refrigeration systems analysis and refrigerant composition optimisation. WORK is used in the following applications:

1. simplifying compound refrigeration systems
2. aiming less shaft work in small temperature cooling operations, cascade and mixed refrigeration systems
3. optimising the quantity and temperature of cooling (refrigeration) levels
4. composite optimisation in mixed refrigeration systems.

The process industry uses many software programs for designing water systems; WATER is the one of the best software tools for this purpose. WATER is used in cooling equipment, steam equipment, washing systems and mass movement operations. WATER checks the areas where it can carry out recycling, regeneration and reuse activities and will then try to reduce water usage.

Distributed effluent treatment systems (DETS) use some design-oriented methods and have minimum costs. Without creating network complexity problems for the designer, the automatic designing of effluent treatment networks, regeneration and water use can be achieved with this software. DETS are used for various applications that are very successful in the market, and can handle multiple contaminants.

The following can be achieved using WATER:

1. minimising water use
2. locating fresh water sources
3. automatic designing of water reuse networks
4. water regeneration
5. automatic design of effluent networks
6. taking care of sewer costs and pipe work in designing networks.
Pinch analysis is a comprehensive and well-defined strategy for plant energy maximisation. In most cases, a Microsoft Excel sheet is used for pinch analysis. Pinch analysis spreadsheets have the following features:

- key for the stream data
- calculations for all the tables and problems of the heat transference.

In new projects and retrofit projects for heat integration improvement, a variety of software can be used; Super Target is the most common for this purpose.

All users need not be masters in every aspect of Super Target, and some users can have minimal pinch expertise. For them, this tool could be very useful, as it can be easily operated and has an intuitive user interface. Pinch analysis is usually a time-consuming task, but it can be automated with the aid of a few advanced tools in this software. However, these tools should only be handled by experts. Data can be shared between Super Target, PRO/II, HYSYS and Aspen Plus. Super Target converts raw process data into pinch data using automatic data extraction systems. The user can change the default values if required. This software is used in academic projects, such as energy analysis in Saudi Arabian thermal desalination plants and in steam turbines [57]; it is used to study the process integration [58].

HEXTRAN is a software system designed to evaluate heat transfer systems [59]. Its main functionalities are to study new systems, measure running systems and optimise and report or address heat transfer issues. It makes the engineer capable of monitoring the performance of various exchangers or heat transfer networks by simulating integrated processes.

HEXTRAN provides support for many types of design analysis and operational work, including the following:

- network designs and individual exchanger
- pinch analysis
- exchanger zone analysis
- split flow
- area payout
- cleaning cycle optimisations.
HEXTRAN efficiently analyses the amount of profit and loss in the heat transfer process. In most heat transfer circuits, HEXTRAN is used for the following important circuits:

- to amplify the energy effectiveness, which directly or indirectly diminishes the input costs
- to amplify the yield and quality of the heat transferring devices
- explaining complex designs and providing a road map in case of operational situations
- providing designs for cost effectiveness and flexible processes.

This section has reviewed a number of popular software products for process engineering tasks. However, it has not covered all available software products in this domain—others include Ariane [60], Apros[61], ChromWorks[62], Wolfram SystemModeler[63], Petro-SIM[64] and MODEL.LA[65], among many more. Most of these are expensive commercial products and were not available for this study.

2.3 Software quality metrics

As mentioned in Chapter 1, the purpose of this study is to research the application of DSL methodology in the process engineering domain. However, to validate DSL product performance, some software quality measures are required in order to compare it with other software products. For this purpose, two classes of approaches were applied in this study:

1- measuring the time and effort required to develop the software (efficiency of the method)
2- evaluating the final software product against certain software quality characteristics.

In order to verify the first criteria, two indices were measured:

1- development time: by measuring the actual time required to develop the software, including design, implementation and testing time
2- **Source Line Of Code (SLOC):** by measuring the number of lines of code, excluding any comments or blank lines.

Conversely, several software quality characteristics to evaluate software products have been defined (portability, reliability, testability, etc.) [65]. However, not all of these characteristics are applicable in this study. Considering the importance of the measure during the software tool lifecycle and the resources available for this study, the following characteristics were selected [66]:

1- **Human-engineered:** Code possesses the characteristics of human engineering to the extent that it fulfils its purpose without wasting users’ time and energy, or degrading their morale. Two factors were used in this study to measure this index:
   a. **performance:** the time duration between when the user is finished entering the data into the software and when the software reports the final results. This must be measured on an identical machine for different software.
   b. **usage time:** the time that the user spends entering the input data into the software.

2- **Maintainability:** Code possesses the characteristics of maintainability to the extent that it facilitates updating to satisfy new requirements or to correct deficiencies.

Since the software products in this study are designed to measure controllability of process designs; in order to measure the human-engineered characteristics of these software products, the time required to perform a controllability analysis or change in the flow-sheet must also be measured. In addition, to measure the maintainability of a software product, the actual time required to implement a modification as well as the number of lines of code added, deleted or updated must also be measured.

2.4 **Software development challenges**
As the software tools in the previous section were reviewed, it was noticed that many other software tools were extinct or not yet prepared for public release. The following section discusses the potential reasons for this.

2.4.1 Software failure quantitative report
Software projects fail due to many factors. Generally, software fails when it does not achieve its long- and short-term goals. This section first presents some statistics about software project failure, followed by the main factors in software project failures. This information will help justify the use of a new methodology in software development.

Certain factors cause these software failures. Being delivered on time, costing no more than the estimated budget and performing in the manner needed are the three main factors that determine whether or not a software project failed.

A survey conducted by Dan Galorath [67] is now presented—first to show the financial dimensions of these failures and highlight the need for a more efficient approach to software development, and secondly, to categorise the factors that cause these failures so they can be addressed in the new approach [67]. This survey is combined several individual reports:

- Standish Group, Chaos University. This first report is by Standish (Figure 2.4-1). A new version is also available (Figure 2.4-2).
- Teta Consultancy Service (TCS) 2007. This report identifies failure parameters from the stockholders’ points of view (Figure 2.4-3).
- Avanade research report (2007). This report focuses on the reasons for software failure from the software engineering point of view (Figure 2.4-4).
- Bob Lawhorn report on software failure (March 2010):
  - weakly defined applications (miscommunication between developer and end user) is to blame for a 66% project failure rate, costing a minimum of $30 billion every year (Forrester Research)
- 60–80% of project failures are the result of poor requirement definitions, analysis and management (Meta Group)
- 40% of problems are identified by the end users (Gartner)
- 40% of all project costs are wasted due to rework (Carnegie Mellon University)
- up to 80% of project funding is spent on fixing self-proposed problems (Dynamic Markets Limited 2007 study).

![Figure 2.4-1. Portion of successful, failed and challenging software projects in 2004][67]

![Figure 2.4-2. Portion of succeeded, failed and challenging software projects (updated)[67]
As illustrated in these reports, a large number of software projects fail to satisfy their users. The factors that mostly contribute to produce failures are listed below [68]:

- requirement-gathering: The first task in developing any software is to gather the requirements. Although this is a time-consuming task and demands a certain amount of communication between the end user (domain expert) and programmer, if it is not done properly, programmers will have no correct input from end users and will build the software based on what they ‘think’ the software should do, instead of what it actually must do.
• lack of user involvement: The two main challenges in managing IT projects are lack of 1) executive support and 2) user involvement [68]. Users should be a part of the software development from the requirement extraction phase to the post-delivery maintenance stage. While this takes effort and time on the part of both end users and programmers, it will prevent the business community feeling as though it is separate from the software.

• team size: The size of the programming team should be chosen properly according to the scale of the project: 10 or fewer for small projects, 11 to 25 for medium projects and 26 or more for large-scale projects [68]. If not attended to, team size can become a source of issues, either in terms of communications or a delay in software delivery.

Other software failure factors include poor time allocation, incorrect testing and poor-quality management. However, these topics are irrelevant to this research, so further studies about them are left to the reader to pursue.

The next section presents a detailed review of DSLs. However, it is important to note that in this study, DSLs are compared with GPLs. GPLs are generally designed and evolved to either implement a new paradigm or to take advantage of hardware improvements. Moreover, using a GPL requires some level of knowledge about its programming paradigm (e.g., object-oriented, function-oriented) and the underlying computer science technologies used in that GPL (e.g., memory management, syntax, linking, etc.).

2.5 DSL definition

2.5.1 Definition

The term ‘DSL’ has been in use for a long time, but like most software notations, it does not have a fixed definition. Some languages are clearly DSL, while it is arguable whether the term can be applied to some others. However, several definitions are available in the DSL literature [69][70]. The common factor among these definitions is the opposition between DSLs and GPLs. A GPL is able to do almost everything that computers do (e.g., C++, Java, Python), while DSLs are mostly designed to deal with the problems of a specific domain (e.g., biology,
business). Moreover, generally speaking, GPLs are Turing complete, an index that determines that GPLs can be used to implement any computational algorithms that requires finite resources, while DSLs might or might not be Turing complete. Removing the constraint of being Turing complete can potentially reduce the number of commands, and consequently complexity and size of the language.

However, Martin Fowler et al. (2010) have defined DSL in the most widely accepted manner: ‘Domain-specific Language (noun): a computer programming language of limited expressiveness focused on a particular domain’ [69].

The above definition is based on four elements [69]:

- computer programming language: A DSL is a computer programming language in the first place—that is, a DSL should be able to communicate perfectly between humans and computers.
- language nature: A DSL, due to its language nature, must be fluent and simple. This does not only mean having limited expressiveness; the manner in which those expressions are combined is also important.
- limited expressiveness: The main idea behind using a DSL is to take advantage of its bounded notations, which are close to the vocabulary of the particular domain. However, this makes DSLs limited, as one cannot build an entire software system with a DSL. Moreover, this limitation makes DSLs easier to learn and use.
- domain focus: A DSL is useful if it can cover its particular domain programming problems. There would be no point in using a programming language with limited expressiveness if it could not handle its domain problems.

2.5.2 Formal definition
Section 2.5.1 offers a qualitative definition of a DSL; this section offers a quantitative approach. Although the qualitative definition helps DSLs be recognised, it does not provide any of their mathematical aspects. DSLs are strongly related to languages and programs, so these two concepts must first be defined.
To define a program, one can start with $p$, a conceptual representation of some computation that runs on a universal computer (Turing machine). Based on this, $P$ can be defined as the set of all available programs. A language, $l$, represents the syntax and semantics needed to express a subset of programs from $P$. Thus, if a program, $p$, is expressed in a language, $l$, then that representation is called the encoding of $p$ in $l$ and shown as $pl$.

However, if the language, $l$, is not Turing complete, it is obviously not possible to write all the programs in $P$ with $l$. The subset of programs that are expressible with $l$ donates as $P_l$. Moreover, not every program is expressed in the same manner in different languages. In fact, some languages are optimised and specialised for encoding programs of a particular subset of $P$.

Based on the last paragraph, one can easily define a domain. The subset of $P$ that is covered by a language $l$, is the domain of $l$ and is denoted as $D_l$. Although this definition is straightforward, it does not help in the context of DSLs, mainly since defining DSLs requires defining program, language and domain separately, while this definition ties domain to its language.

However, two approaches can be useful here [71]. One is to address the domain to a set of programs that are built to handle a group of particular problems (inductive). Expressed another way, this domain is defined based upon the commonalities of a class of programs. This approach mostly emphasises the existing software, not real-world problems.

The other approach, known as a deductive or top-down approach, is based on the real-world problem categorisation; ‘a domain is considered’ [71]. In fact, a domain, $D$, is defined as the class of computational problems for which one might want to build the software. Consequently, $P_D$ is defined as a subset of $P$ that deals with the problems in $D$. As a software engineer, it takes more effort to develop software if its domain is defined from a real-world problem (deductive), compared with when a domain is defined as the union of the common features from a group of software, and software is developed to imply those features. However, in the case of some domains for which not many, if any, software exists, the inductive definition is
mostly useless. Generally, there are several languages that can express $P_D$, but they also cover other parts of $P$.

Now that program, language and domain are theoretically defined, the DSL can be introduced. ‘A domain-specific language $l_D$ for a domain $D$ is a language that is specialised for encoding programs from $P_D$’ [71]. If this specialisation is implemented correctly, developing code with $l$ for $D$ should be more efficient in terms of size of the code, readability and analysability.

However, although some DSLs are capable of encoding programs out of the $D$ due to their wide expressivity (e.g., SQL), it is extremely inefficient to do so. Conversely, because of the Turing completeness of GPLs, it is also possible to write programs in $P_D$ with any GPL, but this still takes more effort compared with encoding the same program in a specialised DSL.

The crucial challenge in developing a DSL is to define its domain boundaries clearly, or to determine what should and should not be included in the domain. Although the answer to this question is the cornerstone of the whole DSL development process, it is, mostly, the experience of the domain experts that answers this question. One solution for this is to start from a fairly rational guess about the domain boundaries and build a DSL based on that. This DSL either under- or over-approximates the $P_D$. The next step is to redefine the domain so that the resulting language covers the whole $P_D$, but nothing more. By this iterative solution, we try to improve the DSL.

As mentioned before, DSLs are defined as the opposite of GPLs. Since almost every programmer is familiar with at least one GPL, using GPLs’ characteristics to define DSLs clears the discussion (Table 2.5-1) [71].
Table 2.5-1. DSLs versus GPLs

<table>
<thead>
<tr>
<th>Properties</th>
<th>GPLs</th>
<th>DSLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>large and complex</td>
<td>smaller and well defined</td>
</tr>
<tr>
<td>Language size</td>
<td>large</td>
<td>small</td>
</tr>
<tr>
<td>Turing completeness</td>
<td>always</td>
<td>often not</td>
</tr>
<tr>
<td>User-defined abstractions</td>
<td>sophisticated</td>
<td>limited</td>
</tr>
<tr>
<td>Execution</td>
<td>via intermediate GPL</td>
<td>native</td>
</tr>
<tr>
<td>Lifespan</td>
<td>years to decades</td>
<td>months to years (driven by context)</td>
</tr>
<tr>
<td>Designed by</td>
<td>guru or committee</td>
<td>a few engineers and domain experts</td>
</tr>
<tr>
<td>User community</td>
<td>large, anonymous and widespread</td>
<td>small, accessible and local</td>
</tr>
<tr>
<td>Evolution</td>
<td>slow, often standardised</td>
<td>fast-paced</td>
</tr>
<tr>
<td>Deprecation/incompatible changes</td>
<td>almost impossible</td>
<td>feasible</td>
</tr>
</tbody>
</table>

The fact that GPLs are Turing complete in most cases results in a vast and complex domain and, consequently, large languages. However, DSLs are rarely Turing complete, so they have a limited domain and bounded size. Apart from that, DSLs’ syntax is defined by domain professionals to handle computational issues of a well-defined domain—user-defined abstraction is thus more limited for them. Moreover, due to the localness and accessibility of the designers and users, DSLs are more flexible in terms of maintenance. In addition, since DSLs are designed to deal with a narrowed-down domain, the DSL is alive while that particular domain exists. This leads to shorter lifespan for DSLs compared with GPLs, which usually have a very long lifespan.
Although this table indicates a clear separation between DSLs and GPLs, the actual boundaries are blurred [72]. Many DSLs possess the characteristics given in the table’s middle column, but still fit the definition of a DSL. For example, SQL, HTML and Mathematica are some languages that are obviously specialised for a limited domain, but feature more characteristics from the middle column.

2.5.3 Different types of DSLs

2.5.3.1 External DSLs

The main feature of an external DSL is the freedom to choose its syntax. Most external DSLs’ syntax is separated from the syntax of the language with which it works. However, it is also common for an external DSL to use an existing syntax (e.g., XML) instead of building a new one. The cost of using custom syntax is to build and maintain language infrastructure tools needed for parsing the DSL script—these tools include interpreter, compilers and linkers. Examples of external DSLs are SQL and DEA.

2.5.3.2 Internal DSLs

Internal DSLs are a subset of a host language. They use the syntax of their host language, which is almost always a GPL. In fact, by using a limited subset of expressions of a GPL, internal DSLs still have the simplicity of a DSL without forcing their programmers to build and maintain language infrastructures. DSL scripts are valid codes in the host language environment, but styled so that they look more like a custom syntax than host code. Few GPLs that are suitable for acting as a host for internal DSLs—Ruby is a good example [73]. Due to its flexible syntax, it is a GPL used for this purpose. Lisp is another example of this class of GPLs [74]—it was the very first GPL used for this purpose in the UNIX community.

2.5.4 DSL elements

2.5.4.1 Semantic model

In the DSL scope, the semantic model is the representation of what DSLs describe [72]. This can be a Petri net model for a controller; it can be a business model for a DSL in a business area; it can be a set of equations for a mathematical DSL; or it can be a set of classes and their methods (Figure 2.5-1).
In fact, mapping should exist between every element of a DSL and its particular semantic model—a DSL script is another representation of its semantic model, which is meaningful for both DSL users and the computer. The relationship between a semantic model and a DSL is significant. As explained above, the semantic model is the engine of a DSL. It is safe to say that most of a DSL’s power comes from its semantic model. By using a semantic model, two concepts of language—parsing and semantic behaviour—become separate [69]. As a result, one can develop, debug and maintain each part without being concerned about the other parts. If the complete separated semantic model is in place, it is possible to test the correctness of the model without being concerned about the parsing process. In addition, the benefits of a well-designed DSL come more from its semantic model than the DSL itself. For example, the ability to change the system’s behaviour comes with the model, not the DSL [69]. Moreover, by having a loosely coupled DSL and semantic model, one can have more than one DSL exposing the model—a DSL is a thin wrapper over its semantic model [71].

However, it is a DSL’s duty to enhance the model capabilities—by this measure, it can be called an adjunct for the model [69].
2.5.4.2 Code generation

In the process of achieving the desired behaviour from a DSL script, one crucial step is to execute the semantic model and attain the results [75]. Based on the design of the semantic model, it can be executable, which is useful only if there is one way of interpreting the data inside the semantic model. The other solution for executing the code, however, is to generate new code from a semantic model—perhaps in another language—and then execute that new script of code[76].

The immediate advantage of including a code-generation process in the DSL framework is portability, which means the ability to use the DSL in different frameworks [75]. For example, if the target language (i.e., the language into which that code is generated) is XML or Java, any computer that supports Java virtual machine technology (which almost every computer does) can be the target for that DSL. Moreover, this process not only improves portability, but makes it possible to have different target environments. However, like any compilation process some extra elements and steps are necessary to develop a code generation. In fact, by adding a code-generation process, one sacrifices the simplicity of the DSL framework.

Generally speaking, code generation is categorised into two main groups: model-aware generation and model-ignorant generation [75][76]. If using the first group, one must have a separate code for the semantic model and the configuration file. The semantic model code loads at the pre-runtime level and represents the model behaviour. However, when paired with the semantic model code, there will be configuration file(s) that need to be maintained.

While the semantic model represents the general model behaviour, the configuration file(s) specifies the operation of the whole framework.
However, the next category covers cases where the configuration file and semantic model codes are bundled together and made into a single piece of code. The best example of this kind is a very simple state machine, explained in Figure 2.5-2 [71]. The generated code could look like this [72]:

![Simple state machine](image)
This single part of code can be loaded into any Central Processing Unit (CPU) and act as a simple switch. Since the semantic model and configuration file are separated, there are many cases where, due to lack of memory and other hardware resources, the first solution is either not feasible or very hard to implement.
2.5.5 Language workbenches

So far, the theoretical aspects of DSLs have been explained, along with some fundamental elements of a well-defined DSL framework. However, DSLs have been around almost from the beginning of the programming language era. Recently, a new set of tools has drawn plenty of attention: language workbenches. These could ‘change the game of DSL significantly’ [78]. Language workbenches provide a friendly environment in which to define and generate DSL scripts. They help both developers and end users (usually one group in the DSL scope) with a high level of tool support. Language workbenches provide an environment that not only helps developers to define the semantic model (the meta-model in the language workbenches scope), but also makes them capable of developing an integrated development environment (IDE) for their DSLs [71]. This tool support plays a crucial role in the DSL game, since it is a challenging task to provide parsers, scanners, code generators, smart editors and many other features that GPLs already offer. More importantly, they offer end users massive feature support, including (but not limited to) auto completion, code refactoring and real-time error detection. Each of these features makes the DSL more valuable and easier to use. At the same time, setting up each of them manually takes plenty of effort and means that many DSL products do not support them.

Table 2.5-2. Language workbenches currently in use

<table>
<thead>
<tr>
<th>Name</th>
<th>Graphical/textual</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Spoofax Language Workbench</td>
<td>Textual</td>
</tr>
<tr>
<td>Meta Programming System</td>
<td>Graphical</td>
</tr>
<tr>
<td>Intentional Workbench</td>
<td>Graphical</td>
</tr>
<tr>
<td>xtext</td>
<td>Textual</td>
</tr>
<tr>
<td>MetaEdit+</td>
<td>Graphical</td>
</tr>
<tr>
<td>Microsoft DSL Tool</td>
<td>Graphical</td>
</tr>
<tr>
<td>Eclipse Modelling</td>
<td>Graphical</td>
</tr>
</tbody>
</table>
A list of popular language workbenches is illustrated in Table 2.5-2. Traditionally, there was a trend for developing tool support for new grammar definition and textual DSLs. Among these, Xtext is a famous plug-in for the Eclipse environment. It is an open-source project, supporting grammar definition and structures to deal with the abstract syntax tree (AST) along with modern editor features such as auto-completion and syntax highlighting [79]. Spoofax is another text-based language workbench that has also garnered attention [80].

Conversely, a number of graphical language workbenches are available. Since most of the products rarely reach their second version, the MetaEdit+ from MetaCase has perhaps outlasted other language workbenches. It supports meta-modelling with a language called Graph, Object, Relationship, Role and Property (GORRP). Basically, it is an object-oriented modelling tool that supports defining the meta-model (the roles and syntaxes that define the semantic model) with a high level of abstraction [81]. Another well-known language workbench is the Intentional Domain Workbench from Intentional Software (http://www.intentsoft.com). They claim that their product is at the frontline of the new programming paradigm. Although they have published several articles [82] and presentations about their product, they also have a secrecy policy that makes it hard to evaluate their assertions. However, Jetbrain’s Meta Programming System (MPS) [83] is a free and open-source language workbench based on grammar definition and textual usage. The main feature of MPS is that it saves the DSL’s script as an AST, not a DSL code. This makes it possible to have more than one view of the DSL script (e.g., graph, table, matrix, etc.); and be able to edit DSL script in each of these views with the change cascading to other views [84]. Microsoft DSL tools [85] and Eclipse Modelling tools are well-known available language workbenches, mostly because of the framework in which they are integrated (.Net for Microsoft DSL tools and Eclipse for Eclipse Modelling tools). They both use unified modelling language (UML) as the meta-modelling language and generate code in their host framework.

2.5.6 Advantages and disadvantages of DSLs
This section discusses the pros and cons of using DSL technology during software development. It can be argued that frameworks, APIs and libraries provide the same
advantages—however, none of these technologies support all of these pros at the same time. For example, although using APIs improves productivity, their syntax is far from a domain expert’s language.

2.5.7 Advantages
Table 2.5-3 illustrates the benefits of DSLs and the sources of these benefits. Each benefit is explained in the following subsection.

2.5.7.1 Productivity
The first aspect of a well-designed DSL is its syntax noise cancellation. There is no need for extra semicolons, parentheses or unnecessary syntax noises; this simplification makes DSL script easier to understand. Moreover, DSL codes are mostly declarative [70], which means their users have to declare what they want the software system to do, not explain how the software system should do it. This also leads to a clearer code. A clear code is not only easy to write, but easy to read and modify [78]. In fact, using DSLs provides, in one sense, the same benefits of choosing meaningful variable names, code documentation and clear code construction [78].
Conversely, due to the expressiveness that DSLs offer, DSL scripts are generally shorter than GPL code in terms of SLOC. This also obviously helps programmers write, read and analyse code more fluently, which increases productivity.

2.5.7.2 Code quality
As mentioned in section 3.1, DSLs have a higher expressiveness and a lower degree of freedom for programmers, which makes writing incorrect programs harder and spotting mistakes in programs easier. DSLs also prevent code duplication due to their higher level of abstraction. This results in less effort for building DSL codes and future maintenance.

2.5.7.3 Requirement gathering
According to Forrester’s report, a lack of efficient communication between software developer and customer causes more project failure than any other factor. Essentially, developers must communicate with customers to gather software requirements, verify and validate their products and provide future maintenance.
Lack of clear communication not only leads to poorly defined software application, but poor requirement analysis and management. DSLs could ease these issues by involving the domain expert in the language-defining process. Obviously, if the domain experts use notations with which they are familiar, they can express and validate their language more fluently. In fact, the effort they once spent on understanding end users’ requirements can now go towards other aspects of software design and development.

2.5.7.4 Validation and verification
One of the challenges of dealing with software projects arises when a behaviour must be added or changed in the system, particularly when the software is fairly extended and vast. Applying documentation roles—for example, UML—makes it much easier to understand, analyse and change a piece of code. However, these methods usually have a long learning curve and do not always cover all programs. What DSLs offer is a self-documented code that a domain expert should be able to read and understand. The immediate benefit of this is that the software user has some idea of how the system works, which significantly improves communication efficiency between them and developers.

However, one could argue that the idea of self-documented languages goes back to COBOL programming language, which was a GPL and was originally designed to support accounting, banking and business domains. Although a huge effort went into embedding the accounting notations and phrases into the language, the end result was a ‘spaghetti factory’ that could not be used as a practical GPL. This means that self-documented programming languages do not provide the expressiveness of a DSL. Another argument against using these languages is that ‘by using a DSL we do not need programmers anymore’. According to Fowler et al. (2010), this is not true [77], since programmers are still needed to write the program, and domain experts to read and analyse the programs.

2.5.7.5 DSL as an analysing tool
Since DSLs are languages designed to express their domain concerns, if this is done correctly, they can be used as a communication tool—not only between programmers and domain experts, but also among domain experts. They provide a
systematic way of expressing the domain model that is not hampered by implementation details and allows domain experts to shift the unnecessary complexities of their models into runtime phase. Therefore, communication and discussion for an executable model can be more productive than a set of mathematical equations. Moreover, a DSL can be used as a learning tool for domain experts. Markus Voelter et al. [71] have claimed that a three-day workshop on DSL prototyping taught extensive information to domain experts. This benefit alone might be worth the DSL building overhead.

2.5.7.6 Better than APIs
There is an argument that APIs, frameworks and libraries also use specific domain notations—so what justifies developing a set of interpreting and compiling tools (e.g., scanner, compiler and linker) for a DSL? Firstly, there are many tools available, such as language workbenches, that provide both programmers and domain experts with the necessary equipment required for developing, interpreting and compiling tools. Secondly, DSLs come with supporting tools that make programming and modelling much easier. Code completion, debugging tools, refactoring and many other supporting tools improve the user’s experience. Based on JetBrains’s report about their webr and dnq Java extensions that are used for web applications and database extensions, for a similar task, customers are more productive when they use webr and dnq Java extensions than when they use J2EE APIs [86].

2.5.7.7 Portability
In many cases, it is important to run the DSL system on several different hardware and operating systems. As shown in section 2.5, one of the key elements of DSLs is their code-generation engine. Due to separation of the code-generation and semantic models, it is absolutely feasible to change the target platform of a DSL by only changing the code-generation engine.

Although each of these advantages alone is not convincing enough to justify using a DSL, in most cases, both programmers and domain experts benefit from more than one of these advantages, which makes using DSLs worthwhile.
2.5.8 Disadvantages

2.5.8.1 Building cost
The first and biggest challenge for using a DSL is building it. If a DSL is already available for that particular domain, it is justifiable to reuse it. However, due to dedicated usage of DSLs, in many cases, building new ones is unavoidable. Developing and maintaining a DSL, as with any other software product, is expensive. It requires involving both programmers and domain experts at the same time. One of the reasons DSL development is expensive to build is that it is a new technology—not everyone is used to defining grammars, building parsers and scanners for them and maintaining the whole system just for a narrow purpose. Using experienced programmers and possessing enough knowledge about the domain, as well as using supporting tools, reduces the costs of building DSLs [71]. Tools such as language workbenches are gamechangers here, since they dramatically decrease the cost of building new DSLs.

2.5.8.2 Learning new languages
Another concern about using DSL is the effort needed to learn it. Costumers mostly tend to stick to what they are already using, even if it is extremely inefficient, and not move to new technologies. However, as opposed to GPLs, DSL users only need to learn notations that are related to the domain. Moreover, if engineered well, writing DSL scripts are nothing more than modelling a domain problem with familiar notations [87].

2.5.8.3 Limiting adoption of changes
A DSL is a modelling tool too—it benefits both programmers and domain experts to abstract their thoughts, and helps them to represent their ideas in a particular domain. However, what if a new concept appears in the domain—can it be expressed by the DSL? Naturally, people try to fit new concepts into the DSL that is already in use, a problem that also applies to other modelling tools [78]. Users refuse to alter a DSL to make new changes. As with any other modelling tool, this can be avoided by checking for new requirements and updating the DSL regularly.
2.5.8.4 DSL hell
With language workbenches, it is easy to develop a new DSL. All that is needed is a grammar or meta-model and a language workbench. With a few clicks, a full operating DSL with an intelligent, language-aware IDE, a good level of abstraction for a semantic model and something that generates code in other environments will be in place [71]. Almost everyone is tempted to create a separate language for every single task. This leads to a situation where there are too many DSLs—none of which have been truly tested—overlapping each other, and none of them are completed. This can occur whenever technology makes a task too easy. However, it can be addressed by regular and well-organised communication in which team members reach an agreement to develop a limited set of DSLs.

2.5.9 DSL examples
As mentioned before, DSLs are as old as programming history, and there are hundreds of them spread across different scopes [70]. This section attempts to introduce and categorise some of these DSLs. Obviously, this study does not cover every DSL in all domains; however, this section explains different domains and their DSLs.

2.5.9.1 Software
The software domain itself is an attractive area for which to build DSLs. SQL and HTML are particularly well-known languages for database queries and web design. However, these languages lose their expressiveness as time passes, and become more like a GPL. Surveys conducted by E Horowitz et al. [88] have produced an overview of these DSLs. Moreover, a handful of DSLs are available for software architecture design and analysis that have been categorised by Medvidovic and Rosenblum [89]. Another group of DSLs that has been used to manage the control of an object-oriented software is known as Behaviour Description Language (BDL) [90]. BDL is a programming language intended for hardware description at various stages, ranging from the algorithm level to function level. In the field of system software, micro-language was developed as a DSL for dealing with operating system specialisations [91]. Finally, some attempts have been made to define, build and test a DSL in financial products [92][93][94].
2.5.9.2 Hardware
VHSIC Hardware Description Language (VHDL) is probably the most famous DSL in the hardware field; it is widely used to design and analyse hardware. Developing device drives for the extended device is a challenging task. In [95], a two-layer DSL approach is presented for making device drivers for video cards specifications. Teapot, as a DSL, is used to address the cache coherence protocols in [96]. The idea of using cache in modern CPUs is to improve performance and stability by providing one (or more) layer of fast-access memory between CPU and RAM. However, because of redundant data stored in that memory, it is challenging to control which data is stored and when CPU should cache the data.

2.5.9.3 Multimedia and office products
In the multimedia area, making 3D animation requires considerable effort. In [97], this issue is addressed by defining a DSL: Haskell and its extensions. fPIC is also a DSL for addressing drawing problems [98]. For the image-manipulation domain, Envision is proposed in [99]. Another area of multimedia DSL usage is web computing [100]. This provides both formal and informal treatment of DSLs for web computing. SrtuQL is another DSL for building websites, which is proposed in [101]. Excel, a program that changed the perception of programming for non-programmers, is a DSL in the spreadsheet domain. Using Excel’s macros and commands is so straightforward that the program is both very productive and popular. LaTeX is also a well-known DSL in the typesetting domain.

2.5.9.4 Telecommunications
Communication protocols present serious problems. The complexity of this area means that it requires certain tools to handle its issues. Promela++ is a DSL for protocol construction and validation stated in [89]. PRL5 is another DSL for addressing telecommunications switching problems [102].

2.5.9.5 Mathematics
Mathematica is arguably one of the most popular DSLs in the mathematics domain. Its usage is not limited to mathematics only; in some cases, engineers and computing and scientists might find it handy. Maple is another DSL that also addresses mathematics problems.
2.6 Conclusion

This chapter reviewed software tools available for process engineering. It also demonstrated that miscommunication between software developers and domain experts could be blamed for the majority of software failures.

DSL methodology was presented as an alternative to traditional software-developing methods, and represented a solution to communication issues. The pros and cons of the DSL methodology were then explained, followed by an overview of current-use DSL cases.

The following chapters develop DSL tools for both imaginary and real-word controllability problems, and study the characteristics of DSL methodology.
**References**


Chapter 2—Software tools literature review

Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (pp. 165-168). ACM.


3.1. Introduction

As the aim of this research is to evaluate the software engineering characteristics of developing DSL products for controllability problems, this chapter reviews currently available controllability definitions and measures in process engineering literature and provides an overview of the software tools that are being used to perform controllability analysis.

This begins with a brief overview of chemical processes to draw a general picture of the scope of this chapter. This is followed by a review of controllability definitions and currently available controllability measures and an overview of the superstructure concept. The chapter concludes with a review of software tools that are in use for controllability assessment.
3.2. Chemical processes

Chemical processes are linear/non-linear continuous/discrete dynamic systems that are prone to significant uncertainties and change. Although chemical systems are typically designed to function around an economically optimal steady-state, varying the value of the process parameters could potentially produce deviations and dynamic responses [1]. Moreover, changing the operating state—that is, the start-up and shutdown of a process—affects the dynamic operation of that process. However, The issues due to the changes of the process during its operation should be address by chemical engineer. Therefore, the operability, i.e. ability to adequately adapt to variations, is a key quality measure that must be taken into account at the design stage.

Operability consists of four major factors [2]:

1- Flexibility: an index that represents the feasibility of a chemical plant’s operation for a window of operating conditions.
2- Controllability: measures the dynamics of process systems. Controllability is explained in detail in the following section.
3- Reliability: represents how well the chemical plant is operable in the case of equipment failure (mechanical, electrical, etc.).
4- Safety: proposes an index to measure the hazards that are the consequence of any chemical plant failure.

This study’s focus is on the controllability aspect of operability topic, given that, regardless of the more than 30 methods and measures to assess controllability [1], there is no dedicated software tool or systematic instruction for controllability analysis. Moreover, controllability is a very important aspect of process systems, because chemical plants with a low controllability index are prone to safety issues—for example, the Chernobyl disaster could have been prevented if the plant’s controllability concerns had been addressed [3]. Conversely, controllability
measures have been defined in detailed mathematical form with clear boundaries, which are useful when applying DSL principles.

The next section is dedicated to controllability of process systems.

3.3. Controllability

3.3.1. Introduction

Controllability has been defined in several ways, with each definition founded on the context and requirements of the user. It was first defined in 1943, when Ziegler and Nichols [4] published an article on the controllability topic. They defined controllability as the ability of a process to reach and preserve a required state. However, Rosenbrock [5] proposed a different definition: a system is controllable, if it is possible to reach the specified goals of control; consequently, the controllability index of the system is determined based on the difficulty of applying control. Rosenbrock later defined ‘Functional controllability’ [5]: “a system is functionally controllable if, for any suitable vector $y$ of output functions defined for $t > 0$, a vector $u$ of inputs defined for $t > 0$ exits, which produces the output vector $y$ using the initial condition $x(0) = 0$”. This definition points to a more specific idea of ‘state controllability’ which is popular in the area of system-theory. State controllability is defined as the capability to transit a system from a known start state to any end state within a finite timeframe [6]. Moreover, a state is labelled controllable if, for any initial state $x(0) = x_0$, any time $t_f$ and final state $x_f$, there exists an input $u(t)$ such that $x(t_f) = x_f$[6]. Based on this definition, a system is defined controllable only if every states of the system are controllable.

Although state controllability is significant from the numerical point of view, it has some disadvantages. For example, it is obvious that not all design states are fully controllable. To cover these disadvantages, ‘Dynamic resilience’ presented by Morari as the quality of the regulatory and servo behaviour that can be gained by feedback [7]. ‘Structural controllability’ is another definition of controllability that is based on the structural information that deals with how the disturbances can affect
the process [8]. As an example, if streams in a process are not highly interconnected, then in the presence of disturbance, the flowrate of one of the streams does not spread over other streams and the process is called controllable. Finally, input–output (I/O) controllability defined by Skogestad as the ability to reach acceptable control performance [9]. Expressed another way, to maintain the outputs variables of a process within a specified limit from their reference values, available inputs and available measurements can be applied to address unknown but restricted changes, such as process changes and disturbances. The main concept of controllability that is currently used in system theory literature is I/O controllability.

Two other controllability definitions are also used occasionally. The first describes controllability as the effort required to keep process at the predefined steady-state. The other indicates that controllability is regarded as a characteristic of the process that shows the effort required to control the process to attain the anticipated requirements.

In summary, controllability can be defined as follows: “controllability is an inherent property of the process that accounts for the ease with which a continuous plant can be held within a specified operating regime despite bounded external disturbances and uncertainties” [1].

There are various reasons for considering the controllability analysis of a process through the design stage [4][1][10][11]. First, because of the high emphasis on the quality of the process products and increasing environmental regulations, processes are pushed to operate within narrow operational limits. Moreover, today’s economy implies significant uncertainty in market demands. In addition, to improve chemical process productivity, unit operations are more interconnected, resulting in complex and compound plants.

In order to accommodate the above, the process must be highly controllable. However, the traditional design method delays the controllability assessment to the last step that can create significant operational and economic errors.
Traditionally, chemical process design progresses over a sequence of evaluations and choices in a sequential manner [1]. It can be summarised via a three-step method: Determining the operational strategy, designing the nominal process and proposing the control system. Based on this sequential method, the control system design delayed until the main characteristics of the process have been recognised. If it is not possible to design a suitable control system, the process itself must be redesigned, resulting in repetition between the process and the control system design phase. Clearly, this repetition is unfavourable due to the effort it requires. Consequently, it is essential to study the controllability property in an initial phase of the process design.

Controllability assessment, defined as the finest possible control value regardless of the controller, leads the control system design throughout the chemical process design. According to the literature, chemical process controllability can be determined by various features. For example, plant design and detailed process dynamics, disturbance features, uncertainty effects, actuator limitations and parameters measurement location [1]. It is essential to show how these features change the process controllability.

Due to the complexity and scale of process design controllability problems, to perform controllability assessments—whether they take place during the design stage or later—the process system design information must be imported to a general-purpose process engineering software tool and analysed against controllability measures.

As mentioned at the beginning of this section, controllability has been defined in various ways and, based on each definition, a group of indices has been defined to measure controllability. The following two subsections presents a brief overview of linear and non-linear controllability indices.

3.3.2. Linear controllability indices
3.3.2.1.1. Poles, zeroes and time delays

Consider the following input–output model of the process system:

\[
y(s) = G(s)u(s)
\]

Eq. 3.3-1

Where \( y(s) \in \mathbb{R}^p \) are the controlled variables, \( u(s) \in \mathbb{R}^p \) are the manipulated variables and \( G(s) \in \mathbb{R}^{m \times p} \) is the transfer function matrix.

Some properties of the above model will apply limitations on the achievable performance through feedback control. These limitations are inherent properties of the process system design and independent of the controller algorithm.

Morari categorised these limitations as follows [7]:

1- feedback controllers intend to be the inverse of the plant transfer function
2- processes inevitability limit the best achievable control performance.

Based on the above statements, the following conclusion can be made [8]:

1- Any right-half-plane (RHP) zero of the process model is translated as the RHP pole in the controller transfer function \( G^{-1}(s) \) and therefore make the controller unstable. Furthermore, the closer the RHP zeroes are to the origin, the harder it is to control the process.
2- Time delays in the \( G(s) \) impose an upper bound on the gain crossover frequency of the controller.

The above conclusions have been used as design heuristics for single-input single-output (SISO) systems at the design stage. Although some works have been done to apply RHP zeroes and time delays for multi-input multi-output (MIMO) systems, expanding them to MIMO is not always possible nor useful, because each variable has its own associated direction and these measures do not consider the interaction between closed loops.
3.3.2.1.2. Singular value and condition number

As mentioned, the interaction between different inputs and outputs plays a very important role in assessing the controllability of a MIMO system.

To quantify these interactions, singular value (SV) and condition number (CN) are defined as follows:

At any given frequency, \( \omega \), the gain of the system \( G(j\omega) \) can be represented using its SV \( \sigma \) as follows:

\[
\sigma_i = \sqrt{\lambda_i(G^*G)}
\]

\textit{Eq. 3.3-2}

Where \( G^*G \) is the complex conjugate transpose and \( \lambda_i \) is the eigenvalue of the \( G \).

The CN is then defined as [12]:

\[
CN(G) \equiv \frac{\sigma}{\sigma_i}
\]

\textit{Eq. 3.3-3}

Where \( \sigma \) and \( \sigma_i \) represent the maximum and minimum singular values, respectively. Usually, the CN is calculated at \( \omega = 0 \) and is an index to measure the relationship between input variables and output variables. A large CN indicates that changes in one particular input are not effective enough to drive its corresponding output, and therefore produce controllability issues.

However, since the CN is highly dependent on scaling the outputs and inputs, normalisation must be applied to generate an effective measure for interaction analysis.

3.3.2.1.3. Relative gain array

Bristol [13] presented another index to measure the interaction between the closed loops of a process system called the relative gain array (RGA). This standard measure can be summarised with the following formula:
\[ \text{RGA}(G) = \Lambda \equiv G \times (G^{-1})^T \]

\textit{Eq. 3.3-4}

Where \( G \) is the transfer function matrix of the closed-loop system and \( \times \) represents element-by-element multiplication.

The elements per column and per row of the RGA will always add up to one. If one RGA value per column is close to one and the remaining values are insignificant, either the interaction in the entire plant is low, or it indicates one-way interaction. In this case, if the inputs and outputs corresponding to the single large entry in each column are paired, the system will perform well in terms of the controllability index.

However, the explained RGA rules do not apply to cases with negative RGA values [14].

To be able to compare process systems using one index, the following is defined as the ‘RGA index’ of a two-by-two system and represents the controllability of that system:

\[ R = \begin{cases} \lambda_{11} \geq 0 : (\lambda_{11} - 1)^2 \\ \lambda_{11} < 0 : (\lambda_{12} - 1)^2 \end{cases} \]

\textit{Eq. 3.3-5}

Based on how \( R \) is defined, the closer \( R \) is to one, the lower the level of interaction between possible control loops.

Linear controllability measures are not limited to the indices explained above, and other popular measures can be named as the robust performance number (RPN) [15], Singular Value Decomposition (SVD) [16], Disturbance Cost (DB) [17] and Partial Disturbance Gain[12].

3.3.3. Non-linear measures
For the majority of cases, a controllability evaluation based on a linearised model of a non-linear system results in sufficient accuracy [18][19]. Basic, static non-linear compensators can be designed with the aim of cancelling the majority of the process non-linearity. Afterwards, it would be possible for the compensated system to be
assessed using linear techniques. This technique can be used in cases where regulatory performance is near a precise steady-state point. However, this is not true for systems representing a large level of non-linearity. Expressed another way, assessments made from these linearised techniques are only valid around a steady-state point, although in many cases, processes display non-linear behaviour that is not easily rectifiable using basic linear transformations. There might be consequences for the dynamic performance of systems as a result of the aforementioned non-linear factors. Therefore, understanding the complex non-linear behaviour of the process is crucial to assessing the results of the operational conditions and parameters.

A variety of techniques and algorithms have been introduced to evaluate the interactions between process design and process control [20]. One group of solutions is based on optimisation methods for synthesis and design that injects controllability into the design problem. For example, the flexibility index introduced by Swaney and Grossmann [21] measures the flexibility (which can be used to measure controllability) of a process at its steady-state by maximising a measure labelled the flexibility index. This index can also be used to measure controllability. Dimitriadis and Pistikopoulos [22] extended this approach for dynamic systems; later on, this index was further expanded by Mohideen et al. [23] by implementing an economic objective. They formulated a process and control system design with an integrated optimisation framework, within which process features and control system characteristics were determined at the same time. Rigorous dynamic models, pre-specified disturbances and PID controllers were used, while significant economic benefits were reported. Bansal et al. [24] extended this index.

Luyben and Floudas [25] implemented a multi-objective optimisation approach. This involved setting up a metrics matrix from dynamic control performance characteristics. Conversely, White et al. [26] introduced an approach to assess the switchability of a process design—that is, its ability to move through steady-states. Their approach was based on finding the best switching trajectory for the plant by
Chapter 3—Controllability literature review

solving an optimisation control problem. One character of this approach is the ability to provide indices specifying the plant design as decision factor.

Bahri et al. [27] introduced a back-off optimisation formulation to inspect the given design’s disturbance rejection capability and to find the optimal back-off design with the intention of rejecting the specified disturbance at the steady-state. One advantage of their work’s optimisation formulation is the ability to include parameters specifying the plant design as decision variables without control design. In their later work, Bahri et al. [28] improved their work to include dynamic systems and control design.

Generally, the literature features two major categories on integrated design and control methods. The first set of approaches emphasise steady-state operation. They seek to develop steady-state designs that are simultaneously economically optimal and dynamically operable around specified steady-states. This approach is applied by using a balance between two main measures: an economic measure and a controllability performance index. The ultimate decision to indicate the ‘best’ design; however, this is usually not deterministic, as it depends on the relative weights used for the deferent objectives.

The drawback of these approaches is usually caused by the weaknesses of the performance indices used. For example, the controllability indicators might not directly and unambiguously relate to real performance requirements [29]. Moreover, the key weakness is that the solutions are only reliable for the specified steady-states because the final index is defined around a steady-state, although the solution is non-linear-based—to change the set point, it is necessary to rerun the optimisation and achieve a new solution. In addition, to check the validity of the results, it is often necessary to analyse the closed loop using dynamic simulations.

Dynamic approaches represent the second set of non-linear approaches [23][26][28][30]. They emphasise the fact that all processes are naturally dynamic and, generally, dynamic operations are unavoidable (or in some cases desirable to)
Chapter 3—Controllability literature review

steady-state operation. Hence, they deploy dynamic performance measurement during the design phase by using dynamic models. As a result, the uncertainty associated with controllability performance is avoided. Moreover, these methods are not limited to the working area around steady-states, meaning that the final decisions concluded are more resilient to disturbances as they are valid over a large area of the operation.

For the purposes of this study, the RGA index has been chosen as the controllability index to apply to process system case studies for three main reasons:

1- The RGA index has been defined and used in control literature since 1960, and thus has a wide range of applications in industrial and real-world use cases. Therefore, it can be used as a reference index for different use cases to study the effort and performance of the DSL methodology in process engineering applications.

2- The RGA index of a MIMO process system is related to both its topology (i.e., the configuration with which the unit operations are connected to each other) and the physical properties of each unit operation. This property of the RGA is described in Chapter 6, where the modifiability of a DSL product is studied through the selection of the superstructure (explained in the next section) of a process system design by software products.

3- In general, the RGA index is relatively simple to implement. This simplicity is useful in this study, since in the following chapters, the same case study is solved by different programming paradigms. As a result, an index is needed that can be implemented in object-oriented programming languages (e.g., C#) as well as general-purpose optimisation software (e.g., MATLAB).

However, as the RGA is a linear index, using it outside the steady-state region might produce invalid results. Nevertheless, since this study seeks to research the performance of DSL methodology, the RGA benefits outweigh its disadvantages over other controllability indices.
3.3.4. Superstructure selection

Generally, there are two main approaches available to perform controllability analysis of a flow-sheet configuration. The first is sequential, and entails dividing the flow-sheet into its primary elements and trying to improve the controllability index of the design by modifying the flow-sheet topology. These modifications are usually based on rational guesses and, as a result, this approach is subjective and cannot be mathematically formulated.

The other approach is to use superstructure, first summarised by Yeomans and Grossmann (1989). In this approach, all the possible configurations are initially created in one large flow-sheet; extra discrete variables are then added in the process system design explaining whether each unit operation is available or not. The flow-sheet also indicates various connection configurations between unit operations on the flow-sheet. Based on these equations, an optimisation problem is defined where the objective function is to improve the various design factors. By solving this optimisation problem (usually it becomes a MILP or MINLP), the best process system design is achieved [31]. If the objective function is defined as the controllability of the design, then the result is the best configuration of unit operations from the controllability point of view. This approach requires high engagement of the software tool.

The main advantage of using superstructure selection is that it will summarise the information of multiple process system designs into one flow-sheet. This not only makes the flow-sheet more readable, but also defines a systematic algorithm with which to choose the best design.

Superstructure selection is used in many cases in the literature. For example, a superstructure framework has been proposed to model heat exchanger networks [32][33] and distillation sequences [34], for controllability and flexibility assessment [28][35] and for CO2 transportation [36].
3.3.5. Software tools for controllability analysis

Computational complexity is often one of the most challenging aspects of controllability assessment, especially in the case of a high (>100) number of controlled and manipulated variables. Generally, three methods are used to take advantage of computers for controllability analysis:

1- using optimisation and modelling software tools: As mentioned in section 2.2.1, a number of software tools provide linear algebra, optimisation, modelling and simulation functionalities. Therefore, based on the controllability index and scale, it is possible to deploy the mathematical model of the process system to the suitable software tool and perform the controllability analysis. This approach requires sufficient knowledge of the software tool that will be used. Since software tools are not specifically designed for controllability analysis, their learning curve is relatively steep. Moreover, is it hard to modify any script developed using these software tools. However, this is the most popular approach for deploying computers in controllability analysis. Use of this method is well illustrated in [37][38][39][40].

2- developing software products to evaluate controllability cases using GPLs: Another approach is to design and develop a software product using GPL specifically for one or a group of controllability problems. With this approach, the process model is implemented at the target GPL, which requires additional components to provide mathematical functionalities. In addition to proficiency knowledge about the process model of the controllability problem, this method demands high programming and software engineering skills. Therefore, this method is not as popular as the first. Examples of this method presented in [41][42][43].

3- developing a software framework for controllability analysis: Similar to software frameworks developed for other specific tasks (e.g., optimisation, modelling, linear algebra, etc.), it is possible to design and implement a
software product that is capable of evaluating number of process designs against various controllability indices. The only software that fits this group is Process Controllability ToolBox (PCTB) [44].

3.4. Conclusion

This chapter offered a general review of controllability analysis, its definitions and its currently used measures; it also explained the reasoning behind choosing RGA as the controllability index for this research. It was argued that, to prevent reiterations through processes design and controller design, controllability analysis should take place during the process design phase. In addition, this chapter presented the superstructure analysis of a flow-sheet. Finally, it reviewed current software tools methods that are used in the controllability domain. The next chapter provides the case study information that will be used in subsequent chapters, as well as the modelling tools and techniques employed for these purposes.
References


Chapter 3—Controllability literature review


Chapter 4

Modelling tools

4.1 Introduction

As stated in chapters 2 and 3, software tools and development methods can affect the procedure of analysing process systems by changing the efficiency and analysis cost. Therefore, the choice of development methods and tools is important. Moreover, the modelling method and its characteristics are also important, as these will be used in the desired software. It is thus key to define and declare the software and modelling tools prior to the design and development phase.

This chapter defines and explains the modelling tools and methods and the case study’s mathematical model used in future chapters.

Section 2 briefly describes the mathematical model of a system consisting of a CSTR, mixer and splitter unit operations. Section 3 explains the modelling methodology and software used in the DSL framework development procedure, and section 4 explains the Petri net modelling tool as the base model for developing a GPL framework in future chapters.
4.2 Unit operations

The following subsections provide information on three unit operations: CSTR, mixer and splitter. These are the flow-sheet elements that are assessed throughout Chapter 5 and 6.

4.2.1 CSTR model

A CSTR is a useful unit operation in the chemical industry. It is a vessel within which the ingredients of a chemical reaction are mixed to the ideal level and in which the required reaction will take place [1]. The chemical reaction that occurs in a CSTR is either exothermic or endothermic, requiring energy to be either removed from or added to the system to maintain a constant temperature and reaction rate. For control purposes, it is also desirable to eliminate the temperature and concentration gradient in a CSTR by mixing the materials. CSTRs usually run at a steady-state, and the system is modelled with no noticeable variation in concentration, temperature or reaction rate throughout the vessel [1]. Assuming the system is well mixed and operating at a steady-state, it can be concluded that the concentration and temperature at the exit point are equal to the temperature and concentration within the vessel. Since it is assumed that the level of liquid inside the CSTR is constant, the flowrate of the output stream is equal to the flowrate of the input stream of the vessel. The following section presents the mathematical model for an ideal CSTR.

4.2.1.1 Process description

Since a CSTR is a common unit operation in chemical and petrochemical plants, various configurations of CSTRs have been modelled and analysed. However, the first practical modelling was conducted by Pottman and Seborg [2] back in 1992, mainly for simulation purposes.

In the example shown in Figure 4.2-1, inside a CSTR, an irreversible exothermic chemical reaction \( A \rightarrow B \) occurs, which is cooled down by a coolant stream. The feed enters the reactor at the temperature of \( T_0 \) and concentration of \( C_{A0} \) at constant volumetric flowrate \( q \). Assuming the steady-
state condition, the product is collected from the reactor at the same volumetric flowrate \( q \).

CSTRs are usually cooled down (or heated up in the case of endothermic reactions) using a coolant (or heating) fluid that circulates at a high velocity within the jacket surrounding the reactor. The coolant volumetric flowrate is assumed to be constant at \( m_c \), with its input temperature at \( T_{c0} \) and exit temperature at \( T_C \).

\[ \text{Feed} \]
\[ q, C_0, T_0 \]

\[ \text{Coolant in} \]
\[ m_c, T_{c0} \]

\[ \text{CSTR} \]

\[ \text{Coolant out} \]
\[ m_c, T_c \]

\[ \text{Product} \]
\[ q, C_A, T \]

---

**Figure 4.2-1 CSTR unit operation**

---

### 4.2.1.2 Mathematical modelling

The following assumptions are made to obtain the simplified model equations of the CSTR shown in Figure 4.2-1:

1. perfect mixing in the reactor and the jacket
2. constant volume in the reactor and the jacket
3. constant mixture density, independent of mixture temperature and concentration
4. constant heat capacity, independent of mixture temperature and concentration.

The mathematical model for this process is formulated using mass and energy balances.

4.2.1.3 Mass balance [3]

\[ V \frac{dC_{A}}{dt} = q(C_{0} - C_{A}) - r_{A} \]

\text{Eq. 4.2-1}

Where \( C_{A} \) and \( C_{0} \) are the product concentration of component A in the reactor and inlet stream, respectively, \( V \) is the volume of the tank and \( r_{A} \) is the rate of reaction. The Arrhenius expression is normally used for the rate of reaction. A first-order reaction results in the following expression:

\[ r_{A} = k_{0} \exp \left( -\frac{E}{RT} \right) C_{A} \cdot V \]

\text{Eq. 4.2-2}

Where, \( k_{0} \) is the reaction rate constant, \( E \) is the activation energy, \( R \) is the ideal gas constant and \( T \) is the reactor temperature on the absolute scale.

4.2.1.4 Energy balance [3]

\[ V \frac{dT}{dt} = q \cdot \rho \cdot C_{p} (T_{0} - T) + DH \cdot r_{A} - \text{cool} \]

\text{Eq. 4.2-3}

where Cool is defined as follows:
Cool = \( Ua (T - T_{c0}) = m_c C_{pc} (T_{c0} - T_c) \)

\[ T_{c0} = \frac{UaT}{m_c C_{pc} + Ua} + m_c C_{pc} \frac{T_c}{m_c C_{pc} + Ua} \]

From equations 4.2—2 and 4.2—5, Cool can be calculated as follows:

\[ Cool = Ua \left( T - \left( \frac{UaT}{m_c C_{pc} + Ua} + \frac{m_c C_{pc} T_c}{m_c C_{pc} + Ua} \right) \right) \]

And DH is defined as follows:

\[ DH = -\frac{\Delta H}{\rho C_p} \]

where \((-\Delta H)\) is the heat of the reaction.

By substituting Cool in Eq. 4.2-3, the energy balance becomes the following:

\[ \frac{dT}{dt} = \frac{\Delta H}{\rho C_p} \cdot k_0 \exp \left( -\frac{E}{RT} \right) C_A + \frac{q \rho C_p}{V} (T_0 - T) + \frac{-Ua}{V} \left( T - \frac{UaT}{m_c C_{pc} + Ua} \right) - \frac{m_c C_{pc} T_c}{m_c C_{pc} + Ua} \]

where \( Ua \) is the heat transfer coefficient multiplied by heat transfer area, \( T_0 \) is the feed temperature, \( T_{c0} \) is the inlet coolant temperature, \( \rho \) is the inlet density, \( m_c \) is the coolant flowrate, \( C_p \) is the specific heat of the inlet stream and the liquid in the CSTR and \( C_{pc} \) is specific heat of the coolant. From Eq. 4.2-1, Eq. 4.2-2 and Eq. 4.2-7, the mass balance and energy balance equations of the CSTR are obtained as follows:
\[
\frac{dC_A}{dt} = \frac{q}{V} (C_{A0} - C_A) - k_0 C_A \exp \left( -\frac{E}{RT} \right) 
\]

Eq. 4.2-8

\[
\frac{dT}{dt} = \frac{\Delta H}{\rho C_p} k_0 \exp \left( -\frac{E}{RT} \right) C_A + \frac{q \rho C_p}{V} (T_0 - T) + \frac{-Ua}{V} \left( T - \frac{UaT}{m_c C_p U_a} \right) - \frac{m_c C_{pc} T_c}{m_c C_{pc} U_a} 
\]

Eq. 4.2-9

where \( T_0 \) and \( T_c \) are the feed and coolant temperatures on the absolute scale, respectively, \( q \) is the feed flowrate, \( m_c \) is the coolant flowrate, \( C_{pc} \) an \( C_p \) are the feed and coolant specific heats, respectively, and \( \rho \) is the liquid density.

Eq. 4.2-10 and Eq. 4.2-11 are represented in the standard-state variable form as follows:

\[
\frac{dC_A}{dt} = f_1(C_A, T, m_c) = \frac{q}{V} (C_{A0} - C_A) - k_0 C_A \exp \left( -\frac{E}{RT} \right) 
\]

Eq. 4.2-10

\[
\frac{dT}{dt} = f_2(C_A, T, m_c) = \frac{\Delta H}{\rho C_p} k_0 \exp \left( -\frac{E}{RT} \right) C_A + \frac{q \rho C_p}{V} (T_0 - T) + \frac{-Ua}{V} \left( T - \frac{UaT}{m_c C_p U_a} \right) - \frac{m_c C_{pc} T_c}{m_c C_{pc} U_a} 
\]

Eq. 4.2-11

4.2.1.5 Linearisation
Throughout this study, the linearised model of the CSTR is used in various sections. The linearised model is cast into the state variable form as follows:

\[
\ddot{x} = Ax + Bu 
\]

\[
y = Cx 
\]
where matrices $A$ and $B$ represent the Jacobian matrices that correspond with the nominal values of the state variables $(x)$, and input variables $(u)$ and $y$ represent the output variables; the output matrix is represented as $C$:

$$
x = \begin{bmatrix} C_A \\ T \end{bmatrix},
  y = \begin{bmatrix} C_A - C_{As} \\ T - T_s \end{bmatrix}
$$

$$
u = m_c$$

where $C_{As}$ and $T_s$ are the steady-state values of the product concentration and reactor/product temperature.

The Jacobian matrix $A$ is given as follows:

$$
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}
$$

where

$$
A_{11} = -\frac{q}{V} - K_s \\
A_{12} = -C_{As} \dot{K}_s \\
A_{21} = \frac{\Delta H}{\rho \cdot C_p} K_s \\
A_{22} = \left(-\frac{q \cdot \rho \cdot C_p}{V}\right) + \frac{\Delta H}{\rho \cdot C_p} \cdot C_{As} \dot{K}_s + \left(-\frac{Ua}{V}\right) + \left(\frac{Ua^2}{V(m_c C_p c + Ua)}\right)
$$

$$K_s = K_0 \exp \left(-\frac{E}{RT_s}\right)$$

$$\dot{K}_s = K_0 \exp \left(-\frac{E}{RT_s}\right) \left(\frac{E}{RT_s^2}\right)$$
The Jacobian matrix $B$ is given by the following:

\[
B = (B_{11} \quad B_{12}) = \begin{bmatrix} \frac{\partial f_1}{\partial u} & \frac{\partial f_2}{\partial u} \end{bmatrix}
\]

\[Eq. \ 4.2-15\]

where

\[
B_{11} = 0
\]

\[
B_{12} = -c_p a^2 (-T_0 + T_c) \frac{(m_{cs} c_p + Ua)^2 V}{(m_{cs} c_p + Ua)^2 V}
\]

The output matrix $C$ is given by the following:

\[
C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\]

\[Eq. \ 4.2-16\]

The steady-state operating data are provided in Table 4.2-1 [3].
### Table 4.2-1 CSTR variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_A$</td>
<td>Product concentration</td>
<td>0.3</td>
<td>mol./l</td>
</tr>
<tr>
<td>$T$</td>
<td>Reactor temperature</td>
<td>280</td>
<td>K</td>
</tr>
<tr>
<td>$m_c$</td>
<td>Coolant flowrate</td>
<td>0.35</td>
<td>kg/min</td>
</tr>
<tr>
<td>$q$</td>
<td>Feed flowrate</td>
<td>0.3</td>
<td>l/min</td>
</tr>
<tr>
<td>$T_0$</td>
<td>Feed temperature</td>
<td>300</td>
<td>K</td>
</tr>
<tr>
<td>$T_{i0}$</td>
<td>Inlet coolant temperature</td>
<td>250</td>
<td>K</td>
</tr>
<tr>
<td>$V$</td>
<td>CSTR volume</td>
<td>2.5</td>
<td>l</td>
</tr>
<tr>
<td>$U_a$</td>
<td>Heat transfer term</td>
<td>0.35</td>
<td>cal./(min. K)</td>
</tr>
<tr>
<td>$k_0$</td>
<td>Reaction rate constant</td>
<td>2.7x10$^8$</td>
<td>Min.$^{-1}$</td>
</tr>
<tr>
<td>$E/R$</td>
<td>Activation energy term</td>
<td>6000</td>
<td>K</td>
</tr>
<tr>
<td>$-\Delta H$</td>
<td>Heat of reaction</td>
<td>-5</td>
<td>cal./mol.</td>
</tr>
<tr>
<td>$\rho$, $\rho_{pc}$</td>
<td>Liquid densities</td>
<td>1</td>
<td>kg/l</td>
</tr>
<tr>
<td>$C_p$, $C_{pc}$</td>
<td>Specific heat</td>
<td>1</td>
<td>cal./(kg K)</td>
</tr>
</tbody>
</table>

In addition to CSTR, the flow-sheets studied in chapters 5 and 6 contain mixer and splitter unit operations; these unit operations are explained here.

It is assumed that no chemical reaction occurs in either of these unit operations. No dynamics assumed for that mixer and the splitter are working in their steady-state.

#### 4.2.2 Mixer

A mixer (Figure 4.2-2) is a basic unit operation that merges two inlet streams into one outlet stream. The model for the mixer is as follows:

\[
Q = Q_1 + Q_2
\]

\[
T = \frac{Q_1T_1 + Q_2T_2}{Q}
\]

\[
C = \frac{Q_1C_1 + Q_2C_2}{Q}
\]

Eq. 4.2-17
where $Q$, $Q_1$, $Q_2$, $T$, $T_1$, $T_2$, $C$, $C_1$ and $C_2$ are outlet flowrate, top inlet flowrate, bottom inlet flowrate, outlet temperature, top inlet temperature, bottom inlet temperature, outlet concentration, top inlet concentration and bottom inlet concentration, respectively.

### 4.2.3 Splitter

Simular to a mixer, a splitter is a basic unit operation that divides an inlet stream into two outlet streams. Splitters (Figure 4.2-3) use an index ($R$) to determine the portion of each output flowrate to the input flowrate. The model for the splitter is as follows:

\[
\begin{align*}
T_1 &= T_2 = T \\
C_1 &= C_2 = C \\
Q_1 &= Q \cdot R \\
Q_2 &= Q(1 - R)
\end{align*}
\]

\textit{Eq. 4.2-18}

where $Q$, $Q_1$, $Q_2$, $T$, $T_1$, $T_2$, $C$, $C_1$, $C_2$ and $R$ are inlet flowrate, top outlet flowrate, bottom outlet flowrate, outlet temperature, top outlet temperature, bottom outlet temperature, outlet concentration, top outlet concentration, bottom outlet concentration and ratio between top and bottom outlet flowrate, respectively.
The next section explains the modelling tool used to model the case studies. This modelling tool is specifically provided by MetaEdit+, which is the language workbench used in this study.

4.3 MetaEdit+

This study used MetaEdit+ as the language workbench to develop DSL products [4]. The main advantages of MetaEdit+ are its user-friendly interface, tool support for meta-model designing and various views for the model. The first version of MetaEdit+ was developed back in 1996 as a research project at the University of Jyväskylä, Finland [5].

MetaEdit+ has several industrial usages. For example, Nokia estimated that applying DSL developed by MetaEdit+ increased productivity by a factor of 10 [6]. Similar results have been obtained in various industries, such as telecommunications, fishing, farming, insurance, railway systems, home automations and wearable sports components [7].

Developing DSL with MetaEdit+ consists of two steps: first, the meta-model of the DSL should be designed and implemented; then, based on the meta-model, the actual model of the problem is generated.
4.3.1 OPPRR modelling tool
MetaEdit+ uses an OPPRR (Object Properties Port Relation Role) modelling tool to illustrate the meta-model. The entities in this modelling tool are as follows [8]:

1. Object: An object is an element that can be placed on its own in a graph. All instances of objects support reuse functionality—an existing object can be reused in other graphs by using the ‘Add Existing’ function.
2. Relationship: A relationship is an explicit connection between two or more objects. Relationships attach to objects via roles.
3. Role: A role specifies how an object participates in a relationship.
4. Port: A port is an optional specification of a specific object part to which a role can connect. Normally, roles connect directly to objects, and the semantics of the connection are provided by the role type. If you want a given role type to be able to connect to different places on an object with different semantics, you can add ports to the object’s symbol. For example, an Amplifier object might have a port for analogue input, a port for digital input and an analogue output port. Roles connecting to each of these will have different semantics. Ports are defined for an object type, and all instances share those same ports.
5. Property: A property is a describing or qualifying characteristic associated with the other types, such as a name, an identifier or a description.

This model is defined formally as a 9-tuple $M = (O, P, \Lambda, R, \Gamma, r, p, \rho, \phi)$, where

- $O = \{o_1, o_2, \ldots, o_n\}$ is a non-empty, finite set of objects
- $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of properties
- $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ is a finite set of ports
- $R = \{r_1, r_2, \ldots, r_n\}$ is a finite set of relations
- $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ is a non-empty, finite set of roles
p: P → (O ∪ Λ ∪ R ∪ Γ) is a function indicating the properties of each entity
ρ: Λ → O is a function indicating valid ports of an object
φ: Γ → R is a function indicating valid roles of a relation
r: R → ×_{i=2}^{n} (K, Λ), where K = \{(o, p)\mid \forall o \in O, (o, p) \in Γ\}, which is the function defining the relations in the model.

In the MetaEdit+ environment, the first five elements of the model (O, P, Λ, R, Γ) are called types, while the next four elements (r, p, ρ, φ) are called bindings.

To model the CSTR case study using this modelling language, unit operations (e.g., CSTRs, mixer, splitter) and streams (e.g., inlet, product outlet, coolant stream) are defined as objects, while the connections among these objects are defined as relations.

For example, the meta-model to model the CSTR case study is as follows:
O = \{<unit operations, streams>\}
P = \{<properties of objects (CSTRs’ volume, splitters’ portion, inlet concentration, etc.)>\}
Λ = \{<connection ports of unit operations, connection ports of streams>\}
R = \{<connections between unit operations and streams>\}
Γ = \{<destination and source roles of connections between objects>\}
p = \{<associating properties to objects>\}
ρ = \{<associating ports to relevant objects>\}
φ = \{<associating roles to relevant stream connections>\}
r = \{<defining connections, involved unit operations and their roles>\}.

Using this meta-model, the CSTR case study is modelled and represented in Appendix 1.

MetaEdit+ provides a powerful toolset to define the meta-model. The procedure to define the DSL using MetaEdit+ consists of two phases: first, defining the types (Figure 4.3-1.b), and then defining the bindings (Figure 4.3-1.a), which determine how the relations apply to objects. In addition, MetaEdit+ provides tools to generate graphical representations (Figure 4.3-2) of types in the meta-model.
Table 4.3-1 shows the graphical representations of objects in the proposed CSTR DSL framework.

<table>
<thead>
<tr>
<th>Component symbol</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="CSTR symbol" /></td>
<td>CSTR</td>
<td>Represents the CSTR unit operations; stores values indicating specifications of the CSTR and the reaction happening at the unit (Volume, Ka, Ua, etc.).</td>
</tr>
<tr>
<td><img src="image" alt="Mixer symbol" /></td>
<td>Mixer</td>
<td>Represents the mixer unit.</td>
</tr>
<tr>
<td><img src="image" alt="Splitter symbol" /></td>
<td>Splitter</td>
<td>Represents the splitter unit; stores the index indicating the portion of its outputs’ flowrate.</td>
</tr>
<tr>
<td><img src="image" alt="Coolant inlet symbol" /></td>
<td>Coolant inlet</td>
<td>Stores physical details of the coolant inlet, such as flowrate, temperature, density.</td>
</tr>
<tr>
<td><img src="image" alt="Product inlet symbol" /></td>
<td>Product inlet</td>
<td>Stores specifications of the inlet streams, such as concentration, temperature and flowrate.</td>
</tr>
<tr>
<td><img src="image" alt="Product outlet symbol" /></td>
<td>Product outlet</td>
<td>Stores specifications of the outlet streams, such as concentration, temperature and flowrate.</td>
</tr>
</tbody>
</table>
Due to the strong modelling capabilities of the OPPRR tool, it is used as the semantic model in this study.

Using this meta-model, every configuration and design of the CSTR system can be modelled. Each model can be demonstrated using various
representations, including graph, matrix or XML file. If any one of the representations is updated, the others will also take on these changes. MetaEdit+ provides a user-friendly graphical environment to produce and edit models using a defined meta-model set in both graph (Figure 4.3-3) and matrix representations. It also generates the XML representation of the model that is used as the input for the other elements of a DSL framework.
<table>
<thead>
<tr>
<th>Component symbol</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="CSTR symbol" /></td>
<td>CSTR</td>
<td>Represents the CSTR unit operations; stores values indicating specifications of the CSTR and the reaction happening at the unit (Volume, Ka, Ua, etc.).</td>
</tr>
<tr>
<td><img src="image" alt="Mixer symbol" /></td>
<td>Mixer</td>
<td>Represents the mixer unit.</td>
</tr>
<tr>
<td><img src="image" alt="Splitter symbol" /></td>
<td>Splitter</td>
<td>Represents the splitter unit; stores the index indicating the portion of its outputs’ flowrate.</td>
</tr>
<tr>
<td><img src="image" alt="Coolant inlet" /></td>
<td>Coolant inlet</td>
<td>Stores physical details of the coolant inlet, such as flowrate, temperature, density.</td>
</tr>
<tr>
<td><img src="image" alt="Product inlet" /></td>
<td>Product inlet</td>
<td>Stores specifications of the inlet streams, such as concentration, temperature and flowrate.</td>
</tr>
<tr>
<td><img src="image" alt="Product outlet" /></td>
<td>Product outlet</td>
<td>Stores specifications of the outlet streams, such as concentration, temperature and flowrate.</td>
</tr>
</tbody>
</table>
4.4 Petri net

As previously mentioned, modelling and presentation tools that are used in future chapters are explained in this chapter. Petri net is a popular modelling and presentation tool that is used in GPL software tool to present the flowsheet and unit operations. A brief description of this tool is presented here, followed by the modelling techniques used to model differential equations with this tool.

4.4.1 Introduction

Petri net, introduced for the first time in 1962 by Dr Carl Adam Petri [9], is one of the most practical modelling tools in many disciplines, including computer science and systems engineering. The main advantage of Petri nets is that they represent a graphical representation of the dynamic behaviour of systems as well as the mathematical behaviour of the system. System behaviour analysis and modelling is achieved via Petri nets’ theoretical aspect, while their graphical representation enables visualisation of the changes in the state of the system. This combination is the main reason for
the significant success of Petri nets. As a result, their footprint is visible in various kinds of dynamic event-driven systems, including computer networks [10], communication systems [11][12], manufacturing plants [13][14][15]; command and control systems [16], real-time computing systems [17][18]; logistic networks [19] and workflows [20][21], to mention only a few important examples.

Petri nets were originally defined for modelling discrete event-based systems, but some studies have adapted them in continuous systems. However, there is still research to be done on the use of Petri nets for analysis of continuous systems.

4.4.2 Modelling

This modelling tool is a particular bipartite-directed graph populated by three main types of object [22]: places, transitions and directed arcs. A directed arc connects places to places and transitions or transitions to places. If there exists an arc, \( a_i \), that starts from a place, \( p_i \), and ends in a transition, \( t \), then \( p_i \) is called \( t \) input; and if there exists an arc, \( a_{out} \), that starts from a transition, \( t \), and ends in a place, \( p_o \), then \( p_o \) is called \( t \) output. Each transition has one and only one output, but at least one input. Places are divided into three groups: manipulated, states and output places. Manipulated places cannot be the output for any transition and output places cannot be the input for any transition. Transitions are also divided into two categories: algebraic and differential transitions. The output of an algebraic transition must be an output place.

The above definition offers another view of state-space model; places thus represent variables (manipulated, state and output variables) and transitions represent the equations (differential and algebraic).

This graph is formally defined as a 7-tuple \( V = (U, S, Y, A, D, I, O) \) where:

- \( U = \{u_1, u_2, \ldots, u_n\} \): is a non-empty, finite set of manipulated places
- \( S = \{s_1, s_2, \ldots, s_n\} \): is a finite set of state places
- \( Y = \{y_1, y_2, \ldots, y_n\} \): is a non-empty, finite set of output places
A = \{a_1, a_2, \ldots, a_n\}: is a non-empty, finite set of algebraic transitions
D = \{d_1, d_2, \ldots, d_n\}: is a finite set of differential transitions
I = (U \cup S) \times (A \cup D) \to R: is an input function that defines directed arcs from places to transitions where R is a real number
O = (D \times S \to R) \cup (A \times Y \to R): is an output function that defines directed arcs from transitions to places where R is a real number.

The graphical notation for each object is represented in Figure 4.4-1 (a).

For modelling a linear system with Petri net, each variable of the model is mapped to its particular place (e.g., manipulated variables to manipulated places, etc.). Equations will also be mapped to their transitions (e.g., differential equations to differential transitions, etc.). To model a differential equation in the form of \(\frac{dy}{dt} = \sum a_i x_i\) where \(x_i\) is continuous variables and is connected to a differential transition using a weighted arc, and the weight of the arc is the coefficient of the variable \(a_i\). The differential transition is connected to the corresponding place of the left-side variable \(y\) using an arc with the weight of one.

Modelling algebraic equations is similar to modelling differential equations, but instead of using a differential transition, an algebraic transition is used. Figure 4.4-2 shows an example of modelling a simple differential equation:

\[
\frac{dy}{dt} = 2x_1 + 3x_2
\]
4.4.3 Clustering
While modelling a complex process with this tool, having too many places, transitions and arcs is a possibility. In this case, the clustering option is added, which clusters a group of objects and names them as a separate unit. By doing this, only the manipulated and output places of that unit are visible (Figure 4.4-1. b). Moreover, by using clustering, it is also possible to define a unit once and use it many times.

4.5 Summary
This chapter has explored the process model of a CSTR, mixer and splitter as well as various modelling tools and methodologies used in later chapters.

Section 4.2 explained the mass and energy balance of CSTR, mixer and splitter unit operations in detail; sections 4.3 and 4.4 illustrated the modelling tools used in the DLS and GPL software tools. CSTR processes will be modelled using these tools.

The modelling tools explained in this chapter are used in the following two chapters to design and implement software products. The next chapter studies the structure and design of the DSL software product and compares it to GPL and software tool products.
References


Chapter 5

Software structure

5.1 Introduction

As stated in Chapter 3, lack of assessment tools for controllability analysis at the design stage could lead to environmental disasters and economical drawbacks. However, due to a variety of controllability measures, developing software tools for this area is a challenging task.

To develop gold-standard software, using well-designed, practical software architecture and structure is essential. In fact, structure is one of the most important aspects of a software tool. Software structure can improve the software’s maintainability—that is, its ability to adopt future changes [1]. Well-designed architecture in a software tool means that every distinct duty of the software is
assigned to only one element of the software; there is minimum overlap with other elements. In this way, both developing and maintaining tasks require less effort.

This chapter explains a new methodology and structure design for developing software tools for controllability analysis. This is achieved by illustrating the architecture of DSL software. Firstly, the general requirements for software in this area are outlined; the DSL system’s top-view structure is then explained, with emphasis on software elements and connections. The structure and specification of each software system element is then discussed.

For comparison reasons, the structure of two other software development methodologies is also described. Software development methodologies are then applied to develop a software system for the CSTR case study explained in Chapter 4. Finally, the software product of each method is clarified and discussed.

5.2 Controllability case study

The case study in this chapter consists of two CSTRs [2]. The inlet stream was divided into two streams using a splitter and sent to both CSTRs; the outlet of the CSTRs was mixed with the mixer to form the process outlet (Figure 5.2-1).
The order of the units in this flow-sheet could change. As stated in Chapter 3, the controllability index of a process system design might vary by changing the design itself—that is, changing the order of units. Therefore, the software tools should be able to analyse different unit configurations (i.e. in different flow-sheets). The four configurations designed and used for this chapter are explained in detail in section 5.5. From the degree of freedom (DOF) analysis perspective, using the model explained in section 4.2, each of these four configurations is represented by four equations (CSTRs’ mass and energy balance) and eight variables \( Q_i, T_i, C_i, M_{c1}, M_{c2}, T_{ci}, C_m \) and \( T_m \), so the DOF can be calculated as follows:

\[
DOF = 8 - 4 = 4
\]

Manipulated, controlled and disturbance variables can be specified as follows:

\( M_{c1} \) and \( M_{c2} \) = manipulated variables.
C_m and T_m = controlled variables.

Q_i, T_i, C_i and T_ci = disturbance variables

The model has a DOF of zero, so a unique solution is possible.

5.3 Requirements

As stated in Chapter 2, requirement analysis is a particularly important stage of any software system development [3][4].

In this case, the requirements are summarised in four parts (Figure 5.3-1):

1) entering the flow-sheet data

2) evaluating the process design based on an appropriate controllability measure

3) saving the design

4) loading previous works.

At the first stage, the user should be able to enter the process design flow-sheet information into the software. The most important factor at this stage is for the user to adopt notations and symbols that are meaningful and familiar to them—this will significantly reduce the learning curve of the software system. The flow-sheet entering process is followed by an evaluation process, which is the main requirement for this software group. The procedure of evaluating the process system design is explained in subsequent sections. Another feature that increases the software efficiency is the ability to save and load a process flow-sheet.
5.4 Design and structure

The structure of a software product affects the total effort required to implement and maintain it. Other software quality metrics of the software product, such as user-friendliness and performance, can also be influenced by the software structure. The structure of each software product is thus represented in this section [5].

All of these software systems were specifically designed based on requirements stated in section 5.3.

5.4.1 DSL approach

Chapter 2 outlines the general structure of a DSL system. However, this section explains the specific structure of the DSL software developed for the CSTR case study.
Figure 5.4-1 illustrates the software system structure. It consists of four decoupled elements, each with a well-defined duty. The main concern in this architecture is maintainability. This structure addresses this issue by separating elements and generalising communication media among them.

This structure offers four elements:

- **GUI**: The major task of this element is to collect the information about the flow-sheet and to present the evaluation results to the user.

- **Semantic model**: As stated in Chapter 2, the semantic model is basically the data structure for storing and analysing the process design. The backbone of the semantic model for this software system is the object-oriented implementation of the meta-model (Figure 5.4-2). The object set in the OPPRR meta-model is interpreted as a set of classes [5]. In addition, the relations in the OPPRR meta-model are interpreted as the design topology.
As emphasised in Chapter 2, the semantic model plays a key role in any DSL software system. This data structure provides the software framework with two main tasks: managing the properties of each operational unit (e.g., the volume of CSTR, the portion of splitter), and storing the topology of the process system design (i.e., the order in which unit operations are connected together). It is essential that the order of unit operations be known by the software system in order to calculate correct steady-state values. To implement this property, a master class is defined as a unit, which has a rank property and set of port connections. These ports help to implement the relations in the OPPRR meta-model. For example, the output port of the CSTR might connect to one of the input ports of a mixer. All of the unit operations are inherited from this class.
Figure 5.4-2 Semantic model class diagram
Code generator: This part of the framework provides the mathematical model of the design in the form of code scripts that are valid for the solver software, which will evaluate the model by processing these code scripts. The output of the code generator contains 10 steps to find the RGA index (Algorithm 1).

Algorithm 1) Solution process to attain RGA:

1. Calculate each unit at steady-state
2. Define manipulated and controlled variables
3. Declare the non-linear equations for each unit
4. Calculate C matrix in the state-space model
5. Use C matrix to calculate the temperature, concentration and flowrate of each unit
6. Generate the linear set of equations from the non-linear equations
7. Form the state-space model, fetch the transfer function
8. Calculate the steady-state gain matrix (K)
9. Calculate RGA matrix (Λ)
10. Calculate R value.

As the first step, the steady-state values of each unit operation are calculated (Algorithm 2). This algorithm starts from all inlet streams, ranks them as zero, adds them to the elem set and increases the rank variable by one. The same procedure then occurs for every element in elem set until that set is empty. The goal of this algorithm is to determine the steady-state calculation order of the unit operations. After calculating the steady-state values, the non-linear model of the process is declared to the solver. The next step is to generate the state-space model of the process system. Firstly, the C (Eq. 4.2-16) matrix in the state-space model is generated based on the influence of each CSTR’s concentration and temperature (state variables)
on the outlet concentration and temperature (controlled variables). The next step is to generate matrices A (Eq. 4.2-14) and B (Eq. 4.2-15) of the state-space model.

Algorithm 2) Ranking the objects in the flow-sheet:

\[
\begin{align*}
\text{rank} &= 0 \\
\text{elem} &= \emptyset \\
\text{Rank every inlet stream as } 0 \\
\text{Repeat} \\
\text{elem} &= \text{All objects with the rank } = \text{rank} \\
\text{Rank every output object of each elements of the elem with rank}+1 \\
\text{Increase rank by one} \\
\text{Until elem }=\emptyset
\end{align*}
\]

- Solver: The final step to assess the flow-sheet design is to evaluate its mathematical model to attain the controllability index. This task is achieved by the solver software in this structure. In this case, the solver is MATLAB software. MATLAB is used here for two main reasons:
  - MATLAB provides comprehensive support for linear and non-linear mathematics, which are highly regarded in this study. In case the study is extended to include non-linear controllability indices, MATLAB also provides optimisation and simulation solutions.
  - MATLAB is a very popular tool among process engineers, and its syntax and commands are well known. It is also available in most universities within Australia. Therefore, using MATLAB would make it easier for other researchers to reproduce the results of this study.
Figure 5.4-3. Sequence diagram for the DSL framework
Figure 5.4-3 shows the sequence of actions by which a process system design evaluates using this software framework.

This process starts by entering the flow-sheet into the GUI. The user draws the flow-sheet to enter the details of the process system design into the software and then triggers the evaluation process by calling the fetchSemanticModel function from the semantic model.

The next step is to calculate the steady-state of each unit in the process system design. However, the order in which the steady-state is calculated is important, since units’ inputs come from other units’ outputs. As a result, the attributes of one unit operation can affect the steady-state of the next one. For example, if the upstream of a CSTR is a mixer, the temperature of mixer’s output is the inlet temperature for the CSTR.

To calculate the correct steady-state of the unit operations, the semantic model calls the rankObjects function, which uses the algorithm described in Algorithm 1, to order the objects in the design. The input of this algorithm is the relation set OPPRR metamodel, and the output is a set of numbers assigned to each unit indicating which unit’s steady-state should be calculated first. The steady-state of the objects with smaller ranks should be calculated first.

After calculating the steady-state of the units, the code generator generates the code scripts necessary to evaluate the process system using the generateEvaluationCode function, which is followed by the solver executing these scripts to obtain the controllability index of the process system design. Finally, this index is replaced in the semantic model and presented to the user.

5.4.2 GPL approach

The second software development methodology used in this study is based on the Petri net modelling tool explained in Chapter 4.
The software structure is straightforward. It consists of three elements (Figure 5.4-4): GUI, Petri net model and mathematical model.

1) GUI: The GUI is responsible for collecting the user’s information on the flow-sheet in the form of a Petri net design from the user. The user defines the input, output and state variables along with the connections among them in this element. The principles explained in section 5.3 were used to develop the GUI element.

2) Petri net model: This element is the data structure that holds the essential information about the design. The classes of this element are designed based on the Petri net modelling tool. It consists of places, arcs and transition classes. Each transition represents a linear equation in the actual model, and the transition type indicates whether the equation is algebraic or differential. Places in this structure represent variables in the model.

3) Mathematical model: This element converts the information from the Petri net model into mathematical equations. As previously explained, each transition represents a linear equation that has one variable on the left and a linear combination of variables on the right. The transfer function, RGA matrix and RGA index are calculated.

As a GPL was used (C#) to implement the software, this methodology is referred to as GPL in this study.
Figure 5.4-5 shows the Petri net representation of the flow-sheet displayed in Figure 5.2-1. The place names are configurable, and in this setup, \( C_f, T_f \) and \( Q_f \) represent the inlet stream concentration, temperature and flowrate, respectively. \( M \) also represents the flowrate of the coolant for each CSTR. The type of variables is shown by the letter in the variable circles: D for disturbance, U for manipulated and Y for controlled variables. This representation is then analysed to calculate the transfer function and RGA index. Figure 5.4-6 shows the results of the flow-sheet controllability analysis displayed in Figure 5.2-1. It also shows the state-space model values (i.e. matrix A, B, C and D), process variables (i.e. matrix C, U, and X), transfer function (matrix G) and the \( \Lambda \) matrix from the input flow-sheet.
5.4.3 Software tool approach

As in the previous approach, commercial mathematical modelling software (MATLAB) was used to develop a software solution to evaluate the controllability index of the case study.

The process of calculating the controllability index is similar to that of other approaches. Firstly, the steady-state of each CSTR is calculated. Then, the linear model is generated and, based on the linear model, state-space and steady-state gain are formed. The RGA index is calculated using the steady-state matrix.

Although it is possible to implement GUI elements in MATLAB (e.g., button, form, etc.), relatively significant effort is required to do so. For example, to draw any shape on a MATLAB GUI, the shape equations must be modelled and drawn on the screen. In addition, modifying these shapes or adding text to them in runtime is a tedious task. Moreover, not all of the MATLAB versions support GUI design and editing. As the result, designing and implementing a tool that includes a GUI does not suit the purpose of this study, which is to use GUI to reduce the effort required to modify flow-sheets; therefore, in this study, the MATLAB...
software tool was implemented as a scripting language and not used to design and implement GUI.

The MATLAB script, which was developed for this purpose, is available in Appendix 2.

5.5 Results
Four different CSTR configurations were studied using the methodologies explained in this chapter. The configurations are as follows:

1- Parallel: The inlet stream is divided into two streams, each into one CSTR, and the CSTR’s outputs mixed to form the outlet (Figure 5.5-1.a).

2- Series 1: The inlet stream enters one CSTR and its output forms the input for the second CSTR. The outlet is the output of the second CSTR (Figure 5.5-1.c).

3- Series 2: The inlet goes straight to the first CSTR, but the output of this CSTR is divided into two streams: one goes into the second CSTR, and the other mixes with the outlet of the second CSTR to form the outlet (Figure 5.5-1.d).

4- Series 3: The inlet is initially divided into two streams; one goes into the first CSTR, and its output connects directly to the input of the second CSTR. The output of the second CSTR mixes with the other part of the inlet to form the outlet (Figure 5.5-1.b).

Appendix 3 presents an evaluated state space, transfer function and Λ matrix for each configuration.
Figure 5.5-1 Different CSTR configurations
To measure the time required to change the parameters of a single flow-sheet, the coolant flowrates of CSTR1 and CSTR2 were changed. Further, to measure the time required to change from one configuration to another, the four states represented in Figure 5.5-1 were implemented using three software products. Table 5.5-1 represents the RGA index (Eq.3.3.5) of these four configurations with a different coolant flowrate for each. The calculated RGA indices are identical when using the three software products. Appendix 4 represents the steady-state values and maximum value of $\Lambda$ matrix.

Table 5.5-1. CSTR Configurations RGA Index (R)

<table>
<thead>
<tr>
<th></th>
<th>CSTR1 Coolant flowrate</th>
<th>CSTR2 Coolant flowrate</th>
<th>Flow-sheet (a)</th>
<th>Flow-sheet (b)</th>
<th>Flow-sheet (c)</th>
<th>Flow-sheet (d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.35</td>
<td>0.35</td>
<td>0.1327</td>
<td>0.5292</td>
<td>0.6301</td>
<td>0.0231</td>
<td></td>
</tr>
<tr>
<td>0.37</td>
<td>0.33</td>
<td>0.1877</td>
<td>0.5306</td>
<td>0.6370</td>
<td>0.0227</td>
<td></td>
</tr>
<tr>
<td>0.4</td>
<td>0.3</td>
<td>0.3357</td>
<td>0.5345</td>
<td>0.6485</td>
<td>0.0219</td>
<td></td>
</tr>
<tr>
<td>0.33</td>
<td>0.37</td>
<td>0.0958</td>
<td>0.5287</td>
<td>0.6238</td>
<td>0.0233</td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>0.4</td>
<td>0.0606</td>
<td>0.5294</td>
<td>0.6210</td>
<td>0.0232</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5-2 compares the development and usage factors of the three software products. Development time is the time spent to design, implement and test each software tool. This index and SLOC can be used as another index to measure the effort required to develop each of these software tools. ‘Update flowrate values’ is the time required to alter an existing flow-sheet configuration. ‘Change
configuration’ represents the time required to enter one flow-sheet into each software product. These two indices are useful for comparing how user-friendly each tool is. Finally, the runtime represents how fast each evaluation was and was measured by automatically starting and stopping a system clock on the same PC used to run each software tool. Runtime is efficiency index for the final product.

This information is used to compare these software tools in the following section.

<table>
<thead>
<tr>
<th>Software product</th>
<th>Development time (h)</th>
<th>SLOC</th>
<th>Update flowrate values (s)</th>
<th>Change configuration (m)</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSL</td>
<td>34</td>
<td>271</td>
<td>12</td>
<td>1</td>
<td>1160</td>
</tr>
<tr>
<td>GPL</td>
<td>162</td>
<td>1440</td>
<td>13</td>
<td>5</td>
<td>900</td>
</tr>
<tr>
<td>MATLAB</td>
<td>21</td>
<td>59</td>
<td>30</td>
<td>17</td>
<td>1100</td>
</tr>
</tbody>
</table>

5.6 Discussion

This section compares the results of the DSL, GPL and MATLAB approaches. However, since the development procedure of the three software elements are independent of the problem that they are solving the same results will be valid for other process engineering problems.

5.6.1 MATLAB product versus DSL Product

It is faster to develop a software tool with MATLAB due to its flexibility and wide coverage of mathematical tools compared with the DSL product, as the development time and the SLOC are smaller for the MATLAB product. In fact, the two main challenges for developing a software tool with MATLAB are modelling the problem in the form of a mathematical equation and learning how to select and use MATLAB functions to obtain the desired data from the model. This approach is straightforward, and does not usually involve concerns that lie within the
software engineering scope, such as loosely coupling, modularity and documentation. Generally, due to their well-designed and well-implemented algorithms, MATLAB products possess faster runtimes compared with DSL products (the runtime of the MATLAB product developed for this study was slightly shorter than the runtime of the DSL product, as shown in Table 5.5-2).

However, entering the data for a process system design into a DSL product is faster than entering it into a MATLAB software tool, as the results show that changing configuration is much faster when using a DSL product compared with a MATLAB product. This is mainly because of the GUI that DSL product provides; this GUI also reduces the learning curve of a DSL product. Therefore, it is easier for the user to learn how to handle CSTR, splitter and mixer icons rather than textual MATLAB functions. Apart from this, relatively small changes are easier to implement in a DSL product than in a MATLAB product (as experienced when modifying the flowrate values, shown in Table 5.5-2); this is also because the GUI provides quick access to data from different parts from the model. Moreover, more fundamental changes (e.g., changing the configuration of CSTRs) are easier to apply. This is because the semantic model and the code generator module provide flexibility.

**5.6.2 DSL product versus GPL product**

GPLs have been used to develop software tools for many years. However, it is faster to develop software tools with DSL rather than GPL (development time and SLOC are almost five times bigger for GPL); this is mostly due to the tool support the DSL offers. In this study, developing a GUI for a GPL product required huge effort, whereas it was a built-in option for the DSL. In addition, a GPL programmer must provide his own classes and modules to solve mathematical equations—a challenging task—whereas, by using a DSL architecture, the code generator layer uses other software tools to perform these calculations. Along with the initial effort needed to develop a software tool with GPL, the end product does not have the quality of the tested DSL tools. For example, in this study, due to
inefficient GUI, it took more effort to change from one flow-sheet to another compared with the DSL product. However, when changing the model characteristics (e.g., coolant flowrate), both products were identical in terms of the time needed to apply the changes.

5.7 Conclusion

This chapter presented a software development approach-based DSL methodology to develop a software tool for evaluating process system designs from a controllability point of view. It showed that the architecture and design of the DSL software tool provides a framework to develop software tools for domain-specific problems, such as controllability problems. To illustrate the capabilities of this approach, three software tools were developed for the two-CSTR case study, and the results from these three software products were then compared in terms of the time and effort needed to develop and use them. The next chapter evaluates the long-term benefits of using the DSL.
References


Chapter 6
Modification implementation

6.1. Introduction
As explained in Chapter 2, computer software products are subjected to a wide range of modification over the course of their lifecycle. These modifications not only apply in various levels within the software architecture, but also have different magnitude.

In this chapter, the three software products implemented in the previous chapter are modified in order to study the effects of adding two new features to them: uncertainty and superstructure selection. This chapter also illustrates the effort required to apply these modifications and the procedures used to apply them.

The structure of this chapter is as follows: firstly, it presents an introduction to various types of uncertainty and their associated effects. It then illustrates the procedure of modifying each software product and explains the superstructure-selection feature and the process of applying it to the software products. Both
modifications are combined and applied to the software tools, and the results are then shared and discussed.

6.2. Uncertainty

Every process system naturally holds some degree of uncertainty and variability. Therefore, process models, which are a qualitative and quantitative translation of a process system, should also represent these phenomena. Failure of a process model to represent the process uncertainty results in incorrect decisions being made at both the design and control stages. For example, although one model might be evaluated as controllable at a certain steady-state, shifting its input to a different region could cause the whole system to become unstable. In fact, the same issue was one of the causes of the Chernobyl disaster [1].

6.2.1. Different types of uncertainty

Based on the nature of the source of uncertainty in a process, a suitable classification can be proposed as follows [2]:

- **Model-inherent uncertainty**: This contains kinetic constants, physical characteristics and transfer coefficients. Information regarding this category of uncertainty is typically obtained from trial and pilot-plant data; a usual report form can be completed via some estimations of a probability distribution function.

- **Process-inherent uncertainty**: Physical quantities (i.e. Flowrate, temperature, stream pressure, etc.) variation forms this category. These uncertainties typically illustrated by a probability distributional form gathered from (online) data.

- **External uncertainty**: Variations that are controlled from outside of the plant, such as product price and demands, input availability, etc. falls into this category.
The purpose of this study is to illustrate the methodology of adding *process-inherent* uncertainty to the software tool.

In real chemical plants, because of variation in the steady-state of upper-hand unit operations, external uncertainty (disturbance) is often likely to occur. This is due to the connections among the unit operations (i.e., the inlet stream of one unit operation is mostly formed by outlets of others). This variation causes many effects on the controllability index of the whole plant. Therefore, it is essential to consider this phenomenon at the design stage.

This section discusses how this feature (considering the disturbance of the variables in controllability analysis) is added into three software tools explained in the last chapter.

6.2.2. DSL
This section outlines the process of implementing the ability to measure the input disturbance uncertainty effect on the controllability index. Firstly, the model explained in section 4.2.1 must be modified to be able to represent the uncertainty of the process system. Adding a new property to the stream object will address this issue. This property indicates the upper and lower bands of the uncertainty that will apply to the temperature, concentration and flowrate of the stream. The next step is to modify the software itself, which consists of two steps:

1. simulating the disturbance on the inlet stream by changing its value within a certain boundary
2. calculating the controllability index with the new inlet stream values.

To apply these changes, two elements (out of four) of the DSL-based framework should be modified; these are hashed in Figure 6.2-1:
1. GUI: The change to this element is simply to add a symbol that shows the disturbance value applied to the inlet stream. This is achieved by defining a symbol and using it on the flow-sheet design (Figure 6.2-2).

2. Code generator: the code generator element is the element that will simulate the disturbance behaviour of the inlet stream for the framework by changing the inlet properties values within the defined boundaries. When the new value is calculated, the code generator automatically generates a new set of scripts and runs the solver to calculate the new controllability index based on the new values of the inlet properties.
6.2.3. GPL
To implement this new feature in the GPL-based software, the same steps as those explained in section 5.4.2 should be taken.

To do this, the stream class is modified so that each parameter of the stream (temperature, concentration, flowrate, etc.) can vary inside the defined boundaries. The user declares this boundary, along with which parameters from the stream have the uncertain value.

To implement this functionality, a new class called Parameter is developed; this holds and varies the values of the stream characteristics.

For the next modification, the sequence of evaluating the design should change by adding an extra loop through which the different values of each parameter are then generated. The new parameter values make a new steady-state of the design and a new controllability index that is automatically calculated by the software.
6.2.4. MATLAB product

To add this feature to the MATLAB product, an extra loop that covers all other parts of the code must be added—through this loop, the values of the selected parameter of the inlet stream (temperature, concentration, flowrate, etc.) vary to add disturbance to each parameter. To add uncertainty to other parameters, the same loop should be placed in the scripts—this adds nested loops to the software. This makes the software harder to read, modify and understand. The modified code is presented in Appendix 5.

Table 6.2-1. Controllability assessment of CSTR case study, including the uncertainty feature.

<table>
<thead>
<tr>
<th>Λ(1,1)</th>
<th>Λ(1,2)</th>
<th>Λ(2,1)</th>
<th>Λ(2,2)</th>
<th>R</th>
<th>Uncertainty(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.57</td>
<td>-2.57</td>
<td>-2.57</td>
<td>3.57</td>
<td>6.6049</td>
<td>20</td>
</tr>
<tr>
<td>4.06</td>
<td>-3.06</td>
<td>-3.06</td>
<td>4.06</td>
<td>9.3636</td>
<td>-20</td>
</tr>
<tr>
<td>3.63</td>
<td>-2.63</td>
<td>-2.63</td>
<td>3.63</td>
<td>6.9169</td>
<td>12</td>
</tr>
<tr>
<td>3.89</td>
<td>-2.89</td>
<td>-2.89</td>
<td>3.89</td>
<td>8.3521</td>
<td>-12</td>
</tr>
<tr>
<td>3.69</td>
<td>-2.69</td>
<td>-2.69</td>
<td>3.69</td>
<td>7.2361</td>
<td>5</td>
</tr>
<tr>
<td>3.79</td>
<td>-2.79</td>
<td>-2.79</td>
<td>3.79</td>
<td>7.7841</td>
<td>-5</td>
</tr>
<tr>
<td>3.72</td>
<td>-2.72</td>
<td>-2.72</td>
<td>3.72</td>
<td>7.3984</td>
<td>2</td>
</tr>
<tr>
<td>3.76</td>
<td>-2.76</td>
<td>-2.76</td>
<td>3.76</td>
<td>7.6176</td>
<td>-2</td>
</tr>
<tr>
<td>3.74</td>
<td>-2.74</td>
<td>-2.74</td>
<td>3.74</td>
<td>7.5076</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2-1 shows the different RGA index for different uncertainty amounts applied to the inlet temperature on the flow-sheet presented in Figure 6.2-3.
The first four columns show different values of \( \lambda \), calculated using Eq. 3.3-4. The fifth column represents the R value, which is calculated using Eq. 3.3-5, and the last column shows the amount of uncertainty (disturbance) applied to the inlet temperature.

These results do not indicate any high correlation between the amount of disturbance on the temperature and the RGA index. Further, the choice of control loops is not affected. Therefore, this flow-sheet is resilient to the inlet stream temperature disturbance.

6.3. Superstructure selection
The following sections explain the procedure of implementing the superstructure-selection feature (section 3.3.4) to the three software tools, and then gives the results.

6.3.1. DSL
The model outlined in section 4.2 must be modified so that it represents the superstructure feature. This is done by adding an extra object to the model, called the ‘auto-splitter’. This object acts almost like a normal splitter, with the only
difference being that the ratio between its outputs’ flowrates is not constant and can vary dynamically. This feature alters the topology of the flow-sheet by changing the stream connection among the unit operations. The auto-splitter can be at one of the states that have been explained in Table 6.3-1. The next step is to modify the software itself. To do this, three out of the four elements of the DSL framework must be altered (Figure 6.3-1).

*Figure 6.3-1. Modified elements for superstructure implementation in the DSL framework (hashed)*
To implement this feature, the following changes must be applied to the DSL framework:

1. **GUI:** A new symbol for the auto-splitter should be added to the framework (Figure 6.3-2). Similar to a normal splitter, this has one input and two outputs that can all connect to other streams.

### Table 6.3-1 Various states of the auto-splitter

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllDown</td>
<td>All of the input flow goes through the bottom output of the auto-splitter.</td>
</tr>
<tr>
<td>Half</td>
<td>Two-thirds of the input flowrate pass through the bottom output and one-third passes through the top output.</td>
</tr>
<tr>
<td>Equal</td>
<td>The input flow is divided equally between the outputs.</td>
</tr>
<tr>
<td>Twice</td>
<td>Two-thirds of the input flowrate pass through the top output and one-third passes through the bottom output.</td>
</tr>
<tr>
<td>AllUp</td>
<td>All of the input flow goes through the top output of the auto-splitter.</td>
</tr>
</tbody>
</table>
Chapter 6—Modification implementation

2. Semantic model: To use information from the auto-splitter, this information must be saved in the semantic model. The important data about the auto-splitter is its topological place (i.e., which stream connects to its input and which streams are its outputs) as well as the states that each instance can have.

3. Code generator: Changes applied to this element represent the auto-splitter’s behaviour. By using an auto-splitter, multiple designs can be generated from one flow-sheet and this is achieved by generating multiple codes for the solver to evaluate. Expressed another way, the code generator is modified so that it can generate a set of scripts for the solver, each representing one state of each auto-splitter.

Figure 6.3-2. Auto-splitter designed in the DSL framework
6.3.2. GPL
The principle of implementing the superstructure selection in the GPL-based software is the same as for the DSL-based framework. A new entity should be added to the system that can give the designs dynamic behaviour—to implement this feature, the flow-sheet should change at the runtime, and this ability is provided by a new element that is added to the GPL software. This element should act in a similar way to the auto-splitter in the DSL-based framework by dynamically changing the process design.

A new class is implemented for this purpose: TAutoSplitter. This is inherited from the splitter class and has an extra attribute: state. The class diagram of the software then changes from Figure 5.4-2 to Figure 6.3-3. By using the state attribute, the user can specify different output ratios for each instance of this object. Possible states are the same as for the auto-splitter in the DSL mode and explained in Table 6.3-1.

The process of evaluating the design remains unchanged, except when the user finishes entering the flow-sheet data into the software; then, the software automatically changes the state of the auto-splitters, resulting in various process configurations. In each of these configurations, the auto-splitter has a fixed state, and it is therefore possible to evaluate the RGA index as before.
Chapter 6—Modification implementation

Figure 6.3-3 Modified class diagram to include superstructure in the GPL product
6.3.3. MATLAB product
Similar to the two other software systems, in order to add the superstructure feature to MATLAB product, an entity called the auto-splitter is implemented; this entity can change its behaviour during the runtime. In this case, it is implemented as a MATLAB function, which results in a stream and a state as arguments and returns two streams as the outputs. The flowrate, temperature and concentration of the outputs are determined based on the input stream and the input state.

However, the main difference in this case is that to control this entity, an extra control loop is required for each auto-splitter to change its state. Further, to vary the location of auto-splitters in this software tool, two different nodes of the code must be modified. This issue is not inherited from the software implementation—regardless of the software tool used, if it is function-oriented, at least two nodes of the code should be modified to apply this change.

In the next section, the case study illustrated in section 4.2 is applied to these three modified software tools, and the results given.

6.4 Results
The results illustrated in this section are the outcome of the two modifications explained in previous sections and are available in link provided at Appendix 3.

Table 6.4-1 shows the RGA index for different designs from the superstructure design illustrated in Figure 6.4-1. The first two columns represent the state of the two auto-splitters. The third column is the positive value from the \( \lambda \) matrix, calculated using Eq. 3.3-4, and fourth column is the RGA index, calculated using Eq. 3.3-5. In these two and the next two columns, the symbol ‘NaN’ is used for the configurations that do not represent a valid configuration (i.e., bypassing one of the CSTRs).
Figure 6.4-1. CSTR case study applied to the DSL framework, including superstructure feature
Table 6.4-1. Controllability assessment of the CSTR case study, including the superstructure feature

<table>
<thead>
<tr>
<th>AutoSplitter 1</th>
<th>AutoSplitter 2</th>
<th>$\lambda$</th>
<th>$R$</th>
<th>$T$</th>
<th>$C_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllDown</td>
<td>AllDown</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>Half</td>
<td>AllDown</td>
<td>1.62</td>
<td>0.3844</td>
<td>$m_{c1}$</td>
<td>$m_{c2}$</td>
</tr>
<tr>
<td>Equal</td>
<td>AllDown</td>
<td>7.6</td>
<td>43.56</td>
<td>$m_{c1}$</td>
<td>$m_{c2}$</td>
</tr>
<tr>
<td>Twice</td>
<td>AllDown</td>
<td>2.02</td>
<td>1.0404</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllDown</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Half</td>
<td>1.16</td>
<td>0.0256</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>Half</td>
<td>Half</td>
<td>3.29</td>
<td>5.2441</td>
<td>$m_{c1}$</td>
<td>$m_{c2}$</td>
</tr>
<tr>
<td>Equal</td>
<td>Half</td>
<td>14.2</td>
<td>174.24</td>
<td>$m_{c1}$</td>
<td>$m_{c2}$</td>
</tr>
<tr>
<td>Twice</td>
<td>Half</td>
<td>2.69</td>
<td>2.8561</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>AllUp</td>
<td>Half</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Equal</td>
<td>1.14</td>
<td>0.0196</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>Half</td>
<td>Equal</td>
<td>5.75</td>
<td>22.5625</td>
<td>$m_{c1}$</td>
<td>$m_{c2}$</td>
</tr>
<tr>
<td>Equal</td>
<td>Equal</td>
<td>25.4</td>
<td>593.4096</td>
<td>$m_{c1}$</td>
<td>$m_{c2}$</td>
</tr>
<tr>
<td>Twice</td>
<td>Equal</td>
<td>3.31</td>
<td>5.3361</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>AllUp</td>
<td>Equal</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Twice</td>
<td>1.15</td>
<td>0.0225</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>Half</td>
<td>Twice</td>
<td>14.4</td>
<td>178.4896</td>
<td>$m_{c1}$</td>
<td>$m_{c2}$</td>
</tr>
<tr>
<td>Equal</td>
<td>Twice</td>
<td>122</td>
<td>14733.1</td>
<td>$m_{c1}$</td>
<td>$m_{c2}$</td>
</tr>
<tr>
<td>Twice</td>
<td>Twice</td>
<td>4.39</td>
<td>11.4921</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>AllUp</td>
<td>Twice</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>AllUp</td>
<td>1.79</td>
<td>0.6241</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>Half</td>
<td>AllUp</td>
<td>17.8</td>
<td>283.2489</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>Equal</td>
<td>AllUp</td>
<td>19.4</td>
<td>338.56</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>Twice</td>
<td>AllUp</td>
<td>14.4</td>
<td>179.0244</td>
<td>$m_{c2}$</td>
<td>$m_{c1}$</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllUp</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
For example, in the configuration in which both auto-splitters are in the ‘AllDown’ state, the second CSTR (CSTR2) is ignored in the design, so the resulting RGA measurement is not feasible (i.e., there is only one manipulated variable available). The next column is the RGA index calculated with Eq. 3.3-5. The last two columns represent the order of pairing of controlled and manipulated variables. For example, in the second row, the temperature and concentration of the outlet are controlled by the coolant flowrate of the CSTR1 and CSTR2, respectively.

According to these results, the best design from a controllability perspective is the one in which the first auto-splitter passes all inlets through the bottom output and the second auto-splitter divides its inlet equally through its outputs (Figure 6.4-2).

6.4.1 Superstructure and uncertainty
This section presents the results from evaluating multiple process designs by applying various uncertainties. Designs were fetched from the previously presented superstructure, and uncertainty values of ±2%, ±5%, ±12% and ±20% were applied
to the inlet temperature. For each configuration, these four sets of uncertainty were applied and the RGA index measured.

Table 6.4-2 illustrates the results for the case study, where ±2% uncertainty in inlet temperature was applied to each design. This table is similar to Table 6.4-1, with the only difference being that three RGA indices were calculated instead of one. In addition, the last two columns represent the absolute and relative differences between the RGA index with and without uncertainty. ‘Drift from 0 uncertainty’ and ‘Drift percentage’ values were calculated based on Eq. 6.4-1 and Eq. 6.4-2, respectively. In other words, the last column in Table 6.4-2 shows the sensitivity of each configuration to the ±2% disturbance in input temperature.

\[
\textbf{Drift from 0 uncertainty} = |\text{RGA}_{\text{uncertainty 0}} - \max(\text{RGA}_{\text{uncertainty+2}}, \text{RGA}_{\text{uncertainty-2}})| \\
\text{Eq. 6.4-1}
\]

\[
\textbf{Drift Percentage} = \frac{\text{drift from 0 uncertainty}}{\text{RGA}_{\text{uncertainty 0}}} \times 100 \\
\text{Eq. 6.4-2}
\]

Three other tables showing the RGA index for ±5%, ±12% and ±20% uncertainty in inlet temperature are presented in Appendix 6.

Table 6.4-3 was generated by merging all drift information from the four uncertainty cases. This table summarises the sensitivity of different configurations produced from the superstructure to different values of uncertainty. Further, the ‘Average’ column of this table shows additional information about each configuration. It indicates that the minimum average drift of the RGA index from a zero uncertainty situation is when AutoSplitter1 is in an ‘AllDown’ state and AutoSplitter2 is in an ‘AllUp’ state. This means that this configuration is the least sensitive to the presence of uncertainty from an RGA index perspective.

Although the ‘AllDown-Equal’ configuration has the best RGA index without any uncertainty, its RGA index increases up to 160% of its original value with the
presence of uncertainty, while on average, it increases by 6.45% for the ‘AllDown’, ‘AllUp’ configuration.
Table 6.4-2. RGA index of all configuration with ±2% uncertainty

<table>
<thead>
<tr>
<th>AutoSplitter</th>
<th>AutoSplitter</th>
<th>Uncertainty</th>
<th>λ</th>
<th>RGA Index</th>
<th>Uncertainty</th>
<th>λ</th>
<th>RGA Index</th>
<th>Uncertainty</th>
<th>λ</th>
<th>RGA Index</th>
<th>Drift from 0</th>
<th>Maximum Drift Percentage%</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllDown</td>
<td>AllDown</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+2%</td>
<td>NaN</td>
<td>NaN</td>
<td>-2%</td>
<td>NaN</td>
<td>NaN</td>
<td>0.25</td>
<td>0.010</td>
</tr>
<tr>
<td>Half</td>
<td>AllDown</td>
<td>0%</td>
<td>1.49</td>
<td>0.2401</td>
<td>+2%</td>
<td>1.48</td>
<td>0.2304</td>
<td>-2%</td>
<td>1.5</td>
<td>NaN</td>
<td>0.25</td>
<td>0.010</td>
</tr>
<tr>
<td>Equal</td>
<td>AllDown</td>
<td>0%</td>
<td>3.74</td>
<td>7.5076</td>
<td>+2%</td>
<td>3.72</td>
<td>7.3984</td>
<td>-2%</td>
<td>3.76</td>
<td>7.6176</td>
<td>0.110</td>
<td>1.465</td>
</tr>
<tr>
<td>Twice</td>
<td>AllDown</td>
<td>0%</td>
<td>2.58</td>
<td>2.4964</td>
<td>+2%</td>
<td>2.55</td>
<td>2.4025</td>
<td>-2%</td>
<td>2.62</td>
<td>2.6244</td>
<td>0.128</td>
<td>5.127</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllDown</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+2%</td>
<td>NaN</td>
<td>NaN</td>
<td>-2%</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Half</td>
<td>0%</td>
<td>1.15</td>
<td>0.0225</td>
<td>+2%</td>
<td>1.14</td>
<td>0.0196</td>
<td>-2%</td>
<td>1.16</td>
<td>0.0256</td>
<td>0.003</td>
<td>13.778</td>
</tr>
<tr>
<td>Half</td>
<td>Half</td>
<td>0%</td>
<td>2.84</td>
<td>3.3856</td>
<td>+2%</td>
<td>2.81</td>
<td>3.2761</td>
<td>-2%</td>
<td>2.88</td>
<td>3.5344</td>
<td>0.149</td>
<td>4.395</td>
</tr>
<tr>
<td>Equal</td>
<td>Half</td>
<td>0%</td>
<td>6.17</td>
<td>26.7289</td>
<td>+2%</td>
<td>6.11</td>
<td>26.1211</td>
<td>-2%</td>
<td>6.24</td>
<td>27.4576</td>
<td>0.729</td>
<td>2.726</td>
</tr>
<tr>
<td>Twice</td>
<td>Half</td>
<td>0%</td>
<td>3.72</td>
<td>7.3984</td>
<td>+2%</td>
<td>3.67</td>
<td>7.1289</td>
<td>-2%</td>
<td>3.78</td>
<td>7.7284</td>
<td>0.330</td>
<td>4.460</td>
</tr>
<tr>
<td>AllUp</td>
<td>Half</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+2%</td>
<td>NaN</td>
<td>NaN</td>
<td>-2%</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Equal</td>
<td>0%</td>
<td>1.13</td>
<td>0.0169</td>
<td>+2%</td>
<td>1.12</td>
<td>0.0144</td>
<td>-2%</td>
<td>1.13</td>
<td>0.0169</td>
<td>0.002</td>
<td>14.793</td>
</tr>
<tr>
<td>Half</td>
<td>Equal</td>
<td>0%</td>
<td>4.67</td>
<td>13.4689</td>
<td>+2%</td>
<td>4.57</td>
<td>12.7449</td>
<td>-2%</td>
<td>4.78</td>
<td>14.2844</td>
<td>0.820</td>
<td>6.084</td>
</tr>
<tr>
<td>Equal</td>
<td>Equal</td>
<td>0%</td>
<td>9.14</td>
<td>66.2596</td>
<td>+2%</td>
<td>9.08</td>
<td>63.8004</td>
<td>-2%</td>
<td>9.32</td>
<td>69.2224</td>
<td>2.963</td>
<td>4.472</td>
</tr>
<tr>
<td>Twice</td>
<td>Equal</td>
<td>0%</td>
<td>4.77</td>
<td>14.2129</td>
<td>+2%</td>
<td>4.71</td>
<td>13.7641</td>
<td>-2%</td>
<td>4.84</td>
<td>14.7456</td>
<td>0.533</td>
<td>3.748</td>
</tr>
<tr>
<td>AllUp</td>
<td>Equal</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+2%</td>
<td>NaN</td>
<td>NaN</td>
<td>-2%</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Twice</td>
<td>0%</td>
<td>1.14</td>
<td>0.0196</td>
<td>+2%</td>
<td>1.13</td>
<td>0.0169</td>
<td>-2%</td>
<td>1.15</td>
<td>0.0225</td>
<td>0.003</td>
<td>14.796</td>
</tr>
<tr>
<td>Half</td>
<td>Twice</td>
<td>0%</td>
<td>9.98</td>
<td>80.6404</td>
<td>+2%</td>
<td>9.58</td>
<td>73.6164</td>
<td>-2%</td>
<td>10.44</td>
<td>89.1136</td>
<td>8.473</td>
<td>10.507</td>
</tr>
<tr>
<td>Equal</td>
<td>Twice</td>
<td>0%</td>
<td>17.54</td>
<td>273.5716</td>
<td>+2%</td>
<td>16.87</td>
<td>251.857</td>
<td>-2%</td>
<td>18.29</td>
<td>298.944</td>
<td>25.373</td>
<td>9.275</td>
</tr>
<tr>
<td>Twice</td>
<td>Twice</td>
<td>0%</td>
<td>6.55</td>
<td>30.8025</td>
<td>+2%</td>
<td>6.5</td>
<td>30.25</td>
<td>-2%</td>
<td>6.62</td>
<td>31.5844</td>
<td>0.782</td>
<td>2.538</td>
</tr>
<tr>
<td>AllUp</td>
<td>Twice</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+2%</td>
<td>NaN</td>
<td>NaN</td>
<td>-2%</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>AllUp</td>
<td>0%</td>
<td>1.79</td>
<td>0.6241</td>
<td>+2%</td>
<td>1.8</td>
<td>0.64</td>
<td>-2%</td>
<td>1.79</td>
<td>0.6241</td>
<td>0.016</td>
<td>2.548</td>
</tr>
<tr>
<td>Equal</td>
<td>AllUp</td>
<td>0%</td>
<td>22.91</td>
<td>480.0481</td>
<td>+2%</td>
<td>24.39</td>
<td>547.092</td>
<td>-2%</td>
<td>21.5</td>
<td>420.25</td>
<td>67.044</td>
<td>13.966</td>
</tr>
<tr>
<td>Twice</td>
<td>AllUp</td>
<td>0%</td>
<td>17.62</td>
<td>276.2244</td>
<td>+2%</td>
<td>18.65</td>
<td>311.523</td>
<td>-2%</td>
<td>16.64</td>
<td>244.61</td>
<td>35.298</td>
<td>12.779</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllUp</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+2%</td>
<td>NaN</td>
<td>NaN</td>
<td>-2%</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Table 6.4-3. Average of RGA index drift from 0% uncertainty for each per cent of uncertainty value in input temperature for every configuration

<table>
<thead>
<tr>
<th>AutoSplitter1</th>
<th>AutoSplitter2</th>
<th>2 (%)</th>
<th>5 (%)</th>
<th>12 (%)</th>
<th>20 (%)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>AllDown</td>
<td>4.12</td>
<td>8.33</td>
<td>31.14</td>
<td>75.97</td>
<td>29.89</td>
</tr>
<tr>
<td>Equal</td>
<td>AllDown</td>
<td>1.47</td>
<td>3.68</td>
<td>10.57</td>
<td>24.72</td>
<td>10.11</td>
</tr>
<tr>
<td>Twice</td>
<td>AllDown</td>
<td>5.13</td>
<td>13.06</td>
<td>36.11</td>
<td>97.42</td>
<td>37.93</td>
</tr>
<tr>
<td>AllDown</td>
<td>Half</td>
<td>13.78</td>
<td>28.44</td>
<td>56.25</td>
<td>135.11</td>
<td>58.40</td>
</tr>
<tr>
<td>Half</td>
<td>Half</td>
<td>4.40</td>
<td>12.31</td>
<td>47.37</td>
<td>102.75</td>
<td>41.71</td>
</tr>
<tr>
<td>Equal</td>
<td>Half</td>
<td>2.73</td>
<td>7.89</td>
<td>48.03</td>
<td>60.51</td>
<td>29.79</td>
</tr>
<tr>
<td>Twice</td>
<td>Half</td>
<td>4.46</td>
<td>12.11</td>
<td>30.42</td>
<td>81.06</td>
<td>32.01</td>
</tr>
<tr>
<td>AllDown</td>
<td>Equal</td>
<td>0.00</td>
<td>15.98</td>
<td>65.31</td>
<td>160.95</td>
<td>60.56</td>
</tr>
<tr>
<td>Half</td>
<td>Equal</td>
<td>6.08</td>
<td>16.43</td>
<td>81.54</td>
<td>186.32</td>
<td>72.59</td>
</tr>
<tr>
<td>Equal</td>
<td>Equal</td>
<td>4.47</td>
<td>12.92</td>
<td>187.02</td>
<td>138.08</td>
<td>85.63</td>
</tr>
<tr>
<td>Twice</td>
<td>Equal</td>
<td>3.75</td>
<td>10.33</td>
<td>24.74</td>
<td>58.08</td>
<td>24.23</td>
</tr>
<tr>
<td>AllDown</td>
<td>Twice</td>
<td>14.80</td>
<td>30.61</td>
<td>77.78</td>
<td>169.90</td>
<td>73.27</td>
</tr>
<tr>
<td>Half</td>
<td>Twice</td>
<td>10.51</td>
<td>32.58</td>
<td>424.26</td>
<td>1665.31</td>
<td>533.16</td>
</tr>
<tr>
<td>Equal</td>
<td>Twice</td>
<td>9.27</td>
<td>29.47</td>
<td>53.16</td>
<td>1569.42</td>
<td>415.33</td>
</tr>
<tr>
<td>Twice</td>
<td>Twice</td>
<td>2.54</td>
<td>5.48</td>
<td>12.77</td>
<td>17.65</td>
<td>9.61</td>
</tr>
<tr>
<td><strong>AllDown</strong></td>
<td><strong>AllUp</strong></td>
<td><strong>2.55</strong></td>
<td><strong>5.13</strong></td>
<td><strong>5.13</strong></td>
<td><strong>13.06</strong></td>
<td><strong>6.47</strong></td>
</tr>
<tr>
<td>Half</td>
<td>AllUp</td>
<td>14.67</td>
<td>40.50</td>
<td>121.89</td>
<td>272.96</td>
<td>112.50</td>
</tr>
<tr>
<td>Equal</td>
<td>AllUp</td>
<td>13.97</td>
<td>38.12</td>
<td>113.26</td>
<td>249.66</td>
<td>103.75</td>
</tr>
</tbody>
</table>

6.5 Discussion

This section presents and discusses the modification results. The modified software products were measured using three indices: the effort needed to apply the changes, the efficiency of using modified products and the runtime. These data are for three cases: applying superstructure-selection modification, embedding
uncertainty and combining these features. The measurement process of each index is the same as that given in Chapter 5.

For the superstructure-selection feature, more fundamental elements of each software tool should be changed, while adding uncertainty involves fewer elements in each case.

Similar to the approach described in Chapter 5, the modifications were applied using three software development methodologies. Table 6.5-1 shows the results.

Table 6.5-1. Effort required to apply modification in each case

<table>
<thead>
<tr>
<th>Modification</th>
<th>Software product</th>
<th>Modification time (Hour)</th>
<th>Number of lines of code added/updated</th>
<th>Changing time for auto-splitter position/uncertainty value (minutes)</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td>20.5</td>
<td>70</td>
<td>10</td>
<td>27510</td>
<td></td>
</tr>
<tr>
<td>DSL</td>
<td>4</td>
<td>28</td>
<td>0.5</td>
<td>29440</td>
<td></td>
</tr>
<tr>
<td>GPL</td>
<td>1.5</td>
<td>23</td>
<td>2</td>
<td>22590</td>
<td></td>
</tr>
<tr>
<td>MATLAB</td>
<td>3</td>
<td>25</td>
<td>2</td>
<td>3360</td>
<td></td>
</tr>
<tr>
<td>DSL</td>
<td>1</td>
<td>18</td>
<td>0.5</td>
<td>3480</td>
<td></td>
</tr>
<tr>
<td>GPL</td>
<td>0.75</td>
<td>18</td>
<td>0.5</td>
<td>2770</td>
<td></td>
</tr>
<tr>
<td>MATLAB</td>
<td>2</td>
<td>8</td>
<td>13.2</td>
<td>55180</td>
<td></td>
</tr>
<tr>
<td>DSL</td>
<td>0.5</td>
<td>4</td>
<td>1</td>
<td>58330</td>
<td></td>
</tr>
<tr>
<td>GPL</td>
<td>0.5</td>
<td>4</td>
<td>2.5</td>
<td>45530</td>
<td></td>
</tr>
</tbody>
</table>

6.5.1 DSL versus MATLAB product

It is possible to compare the novel methodology of developing software for the particular process engineering application and the existing methods from three perspectives: the time and effort needed to implement the modifications, the time needed to use the modified product and the runtime of the modified product.

It is faster to apply changes to a DSL-based product compared with a MATLAB product. This is mainly due to the structure of the DSL-based frameworks. The DSL structure helps the software product absorb modifications more easily;
expressed another way, elements of the DSL framework are not only loosely coupled, which means changing one does not affect others, but the DSL structure is flexible to the point where it can absorb new requirements.

Conversely, due to the design and implementation of the MATLAB product, which is function-oriented, modification requires more effort. Particularly in this case, it was not only necessary to implement a new script for the auto-splitter, but also to add extra streams and connections to implement the modification, which required updating other parts of the application.

However, in the uncertainty case, in which the changes were mostly limited to the code generator element of the DSL-based framework, the effort required to implement the modification was similar for DSL and MATLAB products; this shows the power of DSL technology in the modification case, which is supported by its structure.

The DSL-based framework preserved its user-friendly characteristics after modification due to the presence of a GUI, which was not supported by the MATLAB product.

The lowest layer of the DSL-based framework, the solver, used the same tools as the MATLAB product, which made the runtime almost identical in both cases. This indicates that runtime mainly depends on the solver.

6.5.2 DSL versus GPL
As illustrated, to modify the DSL-based framework, modifying its elements is essential. Since this framework is closely connected with other programs (e.g., solver), changes should satisfy the communication protocols among different layers. However, in GPL-based software, it is easier to apply changes than it is in a DSL-based framework, due to the direct access to memory. Different elements of GPL-based software communicate with each other through memory, while in some
cases, elements of a DSL-based framework should use other media, such as hard drive or network communications, to satisfy the protocol of using these media; programmers must maintain the required protocol and generate codes to use them.

It is also apparent that as the modification involves fewer elements in the DSL-based framework (e.g., uncertainty case), this reduces the overall effort needed to apply modifications, and closes the gap between these two software development methodologies. This is because the interface between the software elements remains unchanged.

Conversely, the time needed to evaluate the design using a DSL-based framework remains low after modification. This is due to the high-quality tools available for this methodology. In this study, the MetaEdit+ offered a GUI toolset that boosted the software development design and implementation process. As illustrated in Chapter 4, the advanced GUI toolset increases the efficiency of designing GUI elements, and the resulting software thus interacts with its user using known notations and symbols with which they are familiar.

The runtime of GPL-based software is slightly better compared with a DSL-based tool. This can be explained based on the structure and design of DSL and GPL products. In the case of the DSL-based framework, information must be transferred among different processes. As discussed previously, this data transfer occurs via either hard drive or network card, both of which are relatively slow compared with data transfer via memory media—the media used in the GPL-based framework.

6.6 Conclusion
This chapter explored the ability to implement and develop new features in the DSL-based framework. It also discussed the process of adding two desirable features to this product in a controllability analysis. Further, to compare the
capacities of DSL methodologies in terms of modifiability, the same changes were applied to two other software products: the GPL-based framework and the MATLAB product. The results were evaluated in terms of the effort needed to apply these changes and the time needed to evaluate any flow-sheet using the modified products.

The chapter also illustrated how the structure of the DSL products can ease future modifications due to loosely coupled elements in this software system; quality modifications can thus be implemented with less effort.
References


Chapter 7

FCE Case Study

7.1 Introduction

So far, this thesis has illustrated the application of DSL technology for developing software for the CSTR case study. As mentioned in Chapter 6, the CSTR process is relatively simple; it was thus chosen to demonstrate the principles of DSL technology.

Depending on the software architecture and development method, the problem size can significantly affect the effort required to develop a software system [1][2].

This chapter focuses on researching the effort required to develop and modify a DSL framework for a real-world chemical process system—an FCE process—to study the effect of problem size on the development process for DSL products.

Sections 7.2 and 7.3 explain the basic principles, structure and various types of evaporators used in the industry. This is followed by the mathematical model, assumptions and non-linear, linear and state-space model of the FCE. Section 7.4 illustrates the general design principles that were followed in the design stage of
this framework; section 7.5 describes each step and gives the results of implementation. Section 7.6 presents the results of evaluating the case study from the controllability point of view, compares them with other software development methods and briefly discusses the efficiency of each method. Section 7.7 concludes the chapter.

7.2 Case study definition

The evaporation process has been used since the time of early civilisations for a wide range of applications, including obtaining salt from seawater [3] and extracting perfumes from flowers and potash from the ashes of the burnt plants [4]. Today, the food, refining and mining industries use the evaporation method as their main separation process.

In the evaporation process, product concentration is achieved by boiling out a solvent (generally water). The remaining end-product should have best solute/solids balance according to the required product quality and economics [3]. This unit operation is used widely in the food industry, chemicals, paper and pulp, pharmaceuticals and beverages [4].

There are several types of evaporator are available (e.g., batch pan, forced circulation, natural circulation, etc.) in the literature [3]. For the purpose of this study, the FCE was chosen for its wide industry usage. Moreover, the main purpose of this chapter is to assess the DSL methodology in terms of scalability in comparison with other software development methodologies, as the FCE model is comparatively larger than the CSTR model [3].

FCEs are typically more expensive than film evaporators due to the need of large recirculating pumps and bore circulating pipework [4]. Therefore, it is important to consider FCE controllability to reduce the operation cost.
7.2.1 **FCE**

The FCE (Figure 7.2-1) was developed for processing liquids that are susceptible to scaling or crystallising. The fresh feed is mixed with the hot recycle stream and pumped into a vertical evaporator, while being heated up with steam passed around the outside walls of the tube. The heated mixture is brought to boiling point and then passes into the separation vessel, where the liquid and vapour are separated. After this, a relatively small fraction of the liquid is drawn off as the product and the rest are circulated and mixed with the input feed. The vapour is cooled down in the cooler with water. The next two sections present the non-linear FCE model, based on the work of Newell and Lee [5].

![Figure 7.2-1 FCE flow-sheet [5]](image-url)
Chapter 7—FCE case study

7.3 The Non-linear model
The FCE control structure and design principles are discussed in the following sections.

7.3.1 FCE variables
A list of variables, including description, standard steady-state values and engineering units, are represented in Table 7.3-1 and Figure 7.2-1. The solvent is water, which is non-volatile.
Table 7.3-1 FCE model variables and their steady-state values [5]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Feed flowrate</td>
<td>10.0</td>
<td>kg/min.</td>
</tr>
<tr>
<td>F2</td>
<td>Product flowrate</td>
<td>2.0</td>
<td>kg/min.</td>
</tr>
<tr>
<td>F3</td>
<td>Circulation flowrate</td>
<td>50.0</td>
<td>kg/min.</td>
</tr>
<tr>
<td>F4</td>
<td>Vapour flowrate</td>
<td>8.0</td>
<td>kg/min.</td>
</tr>
<tr>
<td>F5</td>
<td>Condensate flowrate</td>
<td>8.0</td>
<td>kg/min.</td>
</tr>
<tr>
<td>X1</td>
<td>Feed composition</td>
<td>5.0</td>
<td>%</td>
</tr>
<tr>
<td>X2</td>
<td>Product composition</td>
<td>25.0</td>
<td>%</td>
</tr>
<tr>
<td>T1</td>
<td>Feed temperature</td>
<td>40.0</td>
<td>°C</td>
</tr>
<tr>
<td>T2</td>
<td>Product temperature</td>
<td>84.6</td>
<td>°C</td>
</tr>
<tr>
<td>T3</td>
<td>Vapour temperature</td>
<td>80.6</td>
<td>°C</td>
</tr>
<tr>
<td>L2</td>
<td>Separator level</td>
<td>1.0</td>
<td>m</td>
</tr>
<tr>
<td>P2</td>
<td>Operating pressure</td>
<td>50.5</td>
<td>kPa</td>
</tr>
<tr>
<td>F100</td>
<td>Steam flowrate</td>
<td>9.3</td>
<td>kg/min.</td>
</tr>
<tr>
<td>T100</td>
<td>Steam temperature</td>
<td>119.9</td>
<td>°C</td>
</tr>
<tr>
<td>P100</td>
<td>Steam pressure</td>
<td>194.7</td>
<td>kPa</td>
</tr>
<tr>
<td>Q100</td>
<td>Evaporator duty</td>
<td>339.0</td>
<td>kW</td>
</tr>
<tr>
<td>F200</td>
<td>Cooling water flowrate</td>
<td>208.0</td>
<td>kg/min.</td>
</tr>
<tr>
<td>T200</td>
<td>Cooling water inlet</td>
<td>25.0</td>
<td>°C</td>
</tr>
<tr>
<td>T201</td>
<td>Cooling water outlet</td>
<td>46.1</td>
<td>°C</td>
</tr>
<tr>
<td>Q200</td>
<td>Condenser duty</td>
<td>307.9</td>
<td>kW</td>
</tr>
</tbody>
</table>
7.3.2 Mass balance equations

7.3.2.1 Liquid mass balance
Applying mass balance to the liquid in the separator and evaporator yields [5]:

\[ \rho A \frac{dL_2}{dt} = F_1 - F_4 - F_2 \]

\text{Eq. 7.3-1}\]

where \( \rho \) is the liquid density and \( A \) is surface area of the separator. The term \( \rho A \) is assumed to be constant and its value is 20 kg/m.

The level of liquid is assumed to be constant in the separator. Thus [5]:

\[ F_2 = F_1 - F_4 \]

\text{Eq. 7.3-2}\]

7.3.2.2 Solute mass balance
A mass balance on the solute in the liquid phase yields [5]:

\[ M \frac{dX_2}{dt} = F_1X_1 - F_2X_2 \]

\text{Eq. 7.3-3}\]

where \( M \) is the liquid hold up in the evaporator and separator is assumed to be constant at 20 kg.

7.3.2.3 Vapour mass balance
Mass balance on the process vapour in the system results in the following [5]:

\[ C \frac{dP_2}{dt} = F_4 - F_5 \]

\text{Eq. 7.3-4}\]

where \( C \) is the conversion factor between mass and pressure of the vapour and is assumed to have the constant value of 4 kg/kPa.
7.3.3 Energy balance equations

7.3.3.1 Liquid energy balance

Due to the high circulation rate, it is safe to assume that the process liquid is always at its boiling point and is perfectly mixed. The liquid temperature can be obtained from the linearisation of the saturated liquid line for water near the standard steady-state value [5]:

\[ T_2 = 0.5616 P_2 + 0.3126 X_2 + 48.43 \]

Eq. 7.3-5

Further, the vapour temperature can be calculated from the linearisation of the saturated liquid line for water near the standard steady-state value [5]:

\[ T_3 = 0.507 P_2 + 55.0 \]

Eq. 7.3-6

Assuming the very fast dynamics of the energy transfer, and assuming no heat transfer to or from the environment in any parts of the system [5]:

\[ F_4 = \frac{Q_{100} - F_1 C_P (T_2 - T_1)}{\lambda} \]

Eq. 7.3-7

where:

- \( C_P \) is the heat capacity of the liquid and is assumed to be constant at 0.07 kW/K (kg/min.)
- \( \lambda \) is the latent heat of liquid vaporisation and is assumed to be constant at 38.5 kW/(kg/min.).
7.3.3.2 Evaporator steam jacket

The steam pressure $P_{100}$ relates to steam temperature $(T_{100})$ and is considered to be a possible manipulated variable. The following equation relates $T_{100}$ to $P_{100}$ by linearising the saturated steam temperature–pressure relationship near the standard steady-state value [5]:

$$T_{100} = 0.1538P_{100} + 90$$

*Eq. 7.3-8*

The rate of heat transfer to the boiling liquid is given by the following [5]:

$$Q_{100} = U A_1 (T_{100} - T_2)$$

*Eq. 7.3-9*

where $U A_1$ is the product of heat transfer coefficient and the heat transfer area in evaporator and can be calculated using the following formula [5]:

$$U A_1 = 0.16(F_1 + F_3)$$

*Eq. 7.3-10*

The steam flowrate can be calculated from the following [5]:

$$F_{100} = \frac{Q_{100}}{\lambda_s}$$

*Eq. 7.3-11*

where $\lambda_s$ is the latent heat of stream at saturated conditions and is assumed to be constant at the value of 36.6 kW/(kg/min.).
7.3.3.3 Condenser
Energy balance for the cooling water stream yields as follows [5]:

\[ Q_{200} = F_{200} C_p (T_{201} - T_{200}) \]

Eq. 7.3-12

where \( C_p \) is the heat capacity of the cooling water and is assumed to be constant at 0.07 kW/(kg/min.).

The heat transfer rate in the condenser is as follows [5]:

\[ Q_{200} = U_{A2} (T_3 - 0.5 (T_{200} + T_{201})) \]

Eq. 7.3-13

where \( U_{A2} \) is the product of the heat transfer coefficient and the heat transfer area in the condenser and is assumed to be constant at 6.84 kW/K.

Equations Eq. 7.3-12 and Eq. 7.3-13 can be combined to eliminate \( T_{201} \), as follows [5]:

\[ Q_{200} = \frac{U_{A2} (T_3 - T_{200})}{1 + U_{A2} / (2 C_p F_{200})} \]

Eq. 7.3-14

\( T_{201} \) can then be calculated as follows [5]:

\[ T_{201} = T_{200} + \frac{Q_{200}}{F_{200} C_p} \]

Eq. 7.3-15

The condensate flowrate is calculated as follows [5]:
\[ F5 = \frac{Q200}{\lambda} \]

where \( \lambda \) is the heat of vaporisation of condensate assumed constant at 38.5 kW/(kg/min.).

The above non-linear model includes 12 equations and 19 variables, so the DOF can be calculated as follows:

\[
\text{DOF} = \text{Number of variables} - \text{number of equations} \\
= 19 - 12 \\
= 7
\]

by specifying manipulated, controlled and disturbance variables as follows:

P100 and F200 = manipulated variables

X2, P2 = controlled variables

F3, X1, F1, T1 and T200 = disturbance variables

The model will have a DOF of zero, so a unique solution is possible.

7.3.4 Linear model and state-space model

Using the non-linear model generated in the previous section, the linear and the state-space models of the system are produced.

To calculate the state-space using a non-linear model, initially, all variables and constants with their known values and equations were declared. State variables were assumed to be constant (steady-state assumption); based on this assumption, the system of equations was solved to generate the steady-state values of all variables. The linear model was then calculated by linearisation around steady-state values.
As explained, the method of generating the state space is unaffected by the model size, and as long as the non-linear model exists, it is possible to find the state-space model around the steady-state point. The state-space model is essential for calculating the controllability index (in this case, RGA).

### 7.3.5 Input disturbance

The process is subject to input uncertainties, which means the values of the disturbance variables are not fixed and might vary within a range through the unit operation. To cater for this phenomena, the manipulated variables are modified so that the controlled variables remain steady. If these disturbances are left unhandled, the state-space model data can be dragged from the actual values of the process system. Later in this chapter, section 7.3.5 offers a simulation of the input disturbance effect, and then explains, from a software development perspective, how to apply this effect to the DSL framework.

### 7.4 Developing the DSL framework

So far, the case study specification has been explained. This section discusses the DSL framework structure.

Similar to the CSTR case study, this DSL framework is also formed by four elements: DSL GUI, semantic model, code generator and solver. The duties of each element are explained in the next section.

It is important to emphasise that this case study is relatively large, from a model size perspective, compared with the CSTR case study. Therefore, in this study, the focus is on the flexibility of the DSL methodology in dealing with larger systems.

Further, in the previous case study, it was possible to model each unit operation (e.g., CSTR, splitter, mixer, etc.) separately and merge these models to reach the final model for the whole process system. However, in this case, a number of mathematical equations of the model describe the behaviour of more than one individual unit operation (e.g., solute mass balance), forcing the model to bundle two or more unit operations together. At the same time, the model contains
specific equations describing aspects of the behaviour of single unit operations. Moreover, the existence of a recycle stream prevents the framework from calculating each unit operation’s steady-state separately and then merging them together. Therefore, for the DSL framework, being able to analyse the whole model of the process system design as one single entity is essential.

The FCE was chosen as the case study for this chapter due to its more complex model compared with the CSTR case study. This chapter demonstrates that the model size does not impose any limitations on the DSL framework capabilities—to analyse a bigger case study, the same design and implementation principles apply, but on a larger scale.

Similar to the DSL framework explored in Chapter 5, the requirements from the DSL framework are categorised in three major groups:

1. gathering design information via GUI
2. evaluating process system design with input disturbance
3. basic user interface capabilities (saving, loading, etc.).

As explained in section 7.3, it is essential to provide the non-linear model of the whole process system design as an input to the DSL framework development process. In addition, input uncertainties, which are declared by the user, are applied to the disturbance variables.

Similar to the CSTR case, the requirements are clearly defined and limited to a specific scope (evaporator model); the software product thus fits into the definition of a DSL system.

7.5 Implementation

Now that the design principles of the software framework have been addressed, the development process of the framework can be studied. This section explains, at each step, which element will be implemented and how, and the connections between the elements.
7.5.1 Solver
The first step for developing the DSL framework is to develop the desired code in the solver environment to achieve the end result. In this case, the expected result from the framework is the RGA index of the process system design, and the solver is MATLAB software. The input for this step is the non-linear model of the process system explained in sections 7.3.2 and 7.3.3.

The solver script developed for this case study is shown in Appendix 7. Firstly, the differential equations, variables and constants are defined. In the next step using the values presented in Table 7.3-1, the actual steady-state values for the state variables are calculated by zeroing the differential equations and solving them to find the steady-state values of the variable. The next step is to linearise the equations by differentiating the equations around the steady-state operating point, and then substituting the steady-state values in the linear model to get the state-space model. At this point, the transfer function is fetched from the state-space model and used to calculate the controllability index RGA.

During this step, the feasibility of solving the problem is examined by developing a prototype solution for the problem—in this case, a MATLAB script. This helps isolate the process of implementing the code generator from solving the actual problem.

7.5.2 DSL definition
In this step, the DSL notations and rules are designed. In this part, it is essential to use notations typically used in the field of process engineering in order to preserve the main advantages of using the DSL methodology, which is to make the end product more user-friendly. Moreover, the DSL should be able to gather all the information needed to fully describe the process system design.

In this case, the necessary data from the design is the non-linear model of the process system and the input disturbance values for disturbance variables. The DSL should also be able to collect and illustrate this information from/to the user. Symbols and notations developed for this step are shown in Table 7.6-1.
Conversely, the output of the DSL model is an XML file that contains all the information collected from the user. The structure of the output file strongly depends on how the DSL objects are defined and the relationship among the objects.

In this study, the DSL is defined using the MetaEdit+ language workbench, which was illustrated in Chapter 4.

### 7.5.3 Semantic model implementation

Based on the properties, relationships and connections among the objects in the DSL, a data structure can be implemented. As explained in previous chapters, the semantic model is responsible for keeping track of the process system model specification. At this stage, the semantic model should be designed and implemented so that other elements of the framework can access the model specifications. The FCE process system is modelled based on the tool introduced in Chapter 4 and presented in Appendix 8.

As the next step, a parser is developed to fetch the output data from the DSL element and set it in the semantic model. In this study, to make the evaluation process simpler and more traceable, an XML file is used as an intermediate media, and then parsed into the semantic model. However, in real-time cases in which performance is a big concern, it is possible to link the DSL element to the semantic model directly, so that the data from the process system model fills straight into the semantic model.

### 7.5.4 Code generator implementation

The next step to complete this framework is to implement the code generator. The code generator acts as a system that takes the semantic model as the input and generates a set of instructions that are valid in the solver environment. This new set of instructions generates the end result—in this case, an RGA index.

The crucial role of the prototype generated in the first step is visible in this step. If the structure of the code generator output is similar to that file, it is guaranteed that the results from running the generated code within the solver environment will
also lead to the desirable results, which will greatly reduce the probability of error production in the whole framework.

7.5.5 Code generator controller implementation
So far, the framework is capable of assessing single process system design, without any modification to the flow-sheet. However, it is desirable to be able to apply relatively small changes in the process model and for the framework to still be capable of analysing it. The DSL product explained in this chapter is capable of analysing the process system that contains defined levels of input disturbance. This is achieved by changing the semantic model data, so that the disturbance is applied to the process system and then followed by running the code generator on the new semantic model. This leads to different generated scripts and a different RGA in the case of applied disturbance.

7.6 Results and discussion
This section illustrates the results of evaluating the FCE process system with a DSL framework. Figure 7.6-1 and Table 7.6-1 describe the objects and connections in the DSL framework; sections 7.6.1 and 7.6.2 present the results of the RGA evaluation of the FCE. These results then can be used to determine the best control loop for this process. Appendix 9 shows a sample of calculated state space, transfer function and $\Lambda$ matrix. Completer set of evaluated results are available in link provided at Appendix 3.
Figure 7.6-1. FCE case study modelled by the DSL framework
### Table 7.6-1. DSL defined objects

<table>
<thead>
<tr>
<th>Component Symbol</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Evaporator" /></td>
<td>Evaporator</td>
<td>Represents the evaporator unit. Stores the values of amount of liquid in the evaporator (M), the pressure (P2) of the evaporator and the C constant.</td>
</tr>
<tr>
<td><img src="image" alt="Condenser" /></td>
<td>Condenser</td>
<td>Represents the condenser unit in the model. Stores the value of $UA2$ parameter.</td>
</tr>
<tr>
<td><img src="image" alt="Pump" /></td>
<td>Pump</td>
<td>Represents the pump unit. Stores the value of pump output (F2) flowrate.</td>
</tr>
<tr>
<td><img src="image" alt="Separator" /></td>
<td>Separator</td>
<td>Represents the separator unit. Stores the ρA value.</td>
</tr>
<tr>
<td><img src="image" alt="Mixer" /></td>
<td>Mixer</td>
<td>Represents the mixer unit.</td>
</tr>
<tr>
<td><img src="image" alt="Splitter" /></td>
<td>Splitter</td>
<td>Represents the splitter unit. Stores the index indicating the ratio of its outputs’ flowrates.</td>
</tr>
<tr>
<td><img src="image" alt="Steam inlet" /></td>
<td>Steam inlet</td>
<td>Stores physical properties of the hot steam inlet, such as pressure (P100), temperature (T100) and density, heat capacity and $\lambda_s$.</td>
</tr>
<tr>
<td><img src="image" alt="Coolant inlet" /></td>
<td>Coolant inlet</td>
<td>Stores physical properties of the coolant inlet, such as flowrate (F200), temperature (T200), density and heat capacity.</td>
</tr>
<tr>
<td><img src="image" alt="Coolant outlet" /></td>
<td>Coolant outlet</td>
<td>Stores physical properties of the coolant outlet, such as flowrate (F200), temperature (T201), density and heat capacity.</td>
</tr>
<tr>
<td><img src="image" alt="Condensate" /></td>
<td>Condensate</td>
<td>Stores the physical properties of the condensate outstream, such as flowrate, temperature (T2), density and heat capacity.</td>
</tr>
<tr>
<td><img src="image" alt="Product inlet" /></td>
<td>Product inlet</td>
<td>Stores specifications of the inlet streams, such as concentration (X1), temperature (T1) and flowrate (F1).</td>
</tr>
<tr>
<td><img src="image" alt="Product outlet" /></td>
<td>Product outlet</td>
<td>Stores specifications of the outlet streams, such as concentration (X2), temperature (T2) and flowrate (F2).</td>
</tr>
</tbody>
</table>
7.6.1. FCE RGA index
Using the implemented framework, the RGA index (Eq. 3.3-5) for the FCE case study has been evaluated. Table 7.6-2 shows the results. The first three columns indicate the values of the disturbance variables, the next two columns display the steady-state values of the manipulated variables and the last two columns show the $\lambda$ (between P2 and P100) and the RGA (R) index of the process system, respectively.

<table>
<thead>
<tr>
<th>Disturbance values</th>
<th>Manipulated variables steady-state values</th>
<th>$\lambda$</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>T200 (°C)</td>
<td>T1 (°C)</td>
<td>F1 (kg/min.)</td>
<td>X1 (%)</td>
</tr>
<tr>
<td>25</td>
<td>40</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

7.6.2. FCE RGA index with input disturbance
As a part of the requirements explained in section 7.3.5, the DSL framework should be capable of simulating disturbance by applying various values to the input variables of the process system. Table 7.6-3 represents the results of evaluating the FCE from a controllability point of view after applying disturbance. The first three columns of the table indicate the disturbance percentage on each disturbance variable, and the next two columns represent the values of the disturbance variables after applying the disturbance. As explained in section 7.3.5, to keep the controlled variables fixed at the set point, manipulated variables have been modified, and their new values are presented in third group of columns. The last two groups of columns in the table represent the $\lambda$ (between P2 and P100) and the RGA index of the process system design, respectively.
Table 7.6-3. Results of evaluating the FCE for the RGA index—disturbance applied to variables

<table>
<thead>
<tr>
<th>Percentage(%)</th>
<th>Disturbance Values</th>
<th>Steady state values</th>
<th>λ</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td>F1</td>
<td>X1</td>
<td>T1(C)</td>
</tr>
<tr>
<td>-10</td>
<td>0</td>
<td>-7</td>
<td>-15</td>
<td>36</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>-7</td>
<td>-15</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>-10</td>
<td>0</td>
<td>-15</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>7</td>
<td>-15</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-15</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>-15</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>-10</td>
<td>7</td>
<td>-15</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>7</td>
<td>-15</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>-10</td>
<td>0</td>
<td>7</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-7</td>
<td>7</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-15</td>
<td>44</td>
</tr>
<tr>
<td>-10</td>
<td>0</td>
<td>0</td>
<td>-15</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-15</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>44</td>
</tr>
<tr>
<td>-10</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>-10</td>
<td>7</td>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>-10</td>
<td>0</td>
<td>-7</td>
<td>15</td>
<td>36</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-7</td>
<td>15</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-7</td>
<td>15</td>
<td>44</td>
</tr>
<tr>
<td>-10</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>36</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>44</td>
</tr>
<tr>
<td>-10</td>
<td>0</td>
<td>7</td>
<td>15</td>
<td>36</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>15</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>7</td>
<td>15</td>
<td>44</td>
</tr>
</tbody>
</table>
7.6.3. Time and effort

Figure 7.6-2 illustrates the time required to develop the elements of the DSL framework for the FCE system.

![Development time graph]

According to these results, developing the solver script is the most time-consuming task compared with tasks for other parts of the framework. Consequently, if the solver script is available, the total time required to develop the DSL framework can be halved. However, in many cases, the non-linear and linear models of the process system are already available as an implementation in one of the process modelling tools; this implementation can potentially be used as the solver script. This gives DSL methodology an advantage over other software development methodologies.

Figure 7.6-2 also demonstrates that implementing the GUI part of the DSL product is the least time-consuming aspect of the whole implementation process, and yet the GUI elements are relatively similar to standard flow-sheet symbols (i.e., process domain symbols)
7.6.4. Effort comparison with CSTR case study

Table 7.6-4 compares the effort required to design and implement each element of the CSTR and FCE case studies. In fact, it has been illustrated that the different process model size (FCE with 26 equations and CSTR with four equations) does not have any major effect on the effort required to develop the DSL products. This proves that this methodology is efficient, regardless of the complexity of the process model. This is mainly due to the framework structure, which isolates the mathematical part from the rest; therefore, the model size has very little effect on the total effort.

<table>
<thead>
<tr>
<th>Case study</th>
<th>Solver script (H)</th>
<th>Semantic model (H)</th>
<th>GUI (H)</th>
<th>Code generator (H)</th>
<th>Total (H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCE</td>
<td>16</td>
<td>7</td>
<td>5</td>
<td>4.5</td>
<td>32.5</td>
</tr>
<tr>
<td>CSTR</td>
<td>14.5</td>
<td>5.5</td>
<td>6</td>
<td>4</td>
<td>30</td>
</tr>
</tbody>
</table>

7.7 Conclusion

This chapter explored applying DSL framework in developing software for controllability analysis of an FCE case study. The development methodology proposed was based on the four-element approach of the DSL framework. The procedure of designing and implementing the DSL framework was explained, and the RGA index calculation for the FCE case study illustrated.

This chapter demonstrated that despite the process model’s complexity and size, the amount of effort required to develop the DSL framework is roughly the same—so the methodology proposed in Chapter 4 is general and can be applied to any process system.

This chapter also illustrated that implementing solver script is the most time-consuming element of developing the DSL framework, and in cases where this script is available, using the DSL framework is even more justifiable.
References


Chapter 8

Conclusion

8.1 Summary of research contribution

This thesis aimed to address the challenge of developing software tools for the process engineering domain—specifically, the controllability assessment procedure.

In order to provide a systematic solution, the controllability analysis of a process system design was reviewed in detail. It was bolded out that early stage controllability analysis is essential to avoid iterations between design phases and, as a consequence, reduce the effort for this task. The importance of the software tools for controllability assessments was also highlighted.

Within process engineering literature, several software tools exist for modelling, simulation, optimisation and numerical calculation. Subsequently, a survey of these software tools was conducted, and the challenges facing software projects were presented. It was concluded that miscommunication between domain experts and software developers is the main cause of these challenges. Motivated by the efficiency, modifiability, scalability and usability of DSL tools, a software development procedure was proposed to address the communication challenge in the process. DSL methodology was used as a guideline to improve the domain expert interactions in the software development process.
Applying the proposed methodology, two software tools were implemented for controllability problems, where the modifiability, reusability and usability of result tools and the efficiency of the software development process were compared using conventional development methods. The results indicated 400%, 500%, and 1700% improvement for modifiability, efficiency, and usability, respectively.

Considering these results, the researchers believe that major goal of the study has been achieved.

The study’s key contributions are as follows:

1. Chapter 4 presented a DSL model for modelling the CSTR case study, and Chapter 5 discussed the general requirements for controllability problems based on the model presented in Chapter 4. Based on those requirements, three software tools were developed using three software development methodologies. How each requirement influenced the software tools’ architecture was then illustrated; how the principles of DSL methodology could be applied in the field of process engineering and controllability analysis in order to develop an appropriate software was also discussed. These tools were then compared in terms of the effort required to develop them and their performance, efficiency and user-friendliness. It was concluded that tool support and icons and symbols familiar to domain experts increase the efficiency of the resulting software tools.

2. In Chapter 6, modifiability as a software quality measure was selected to evaluate the software tools developed and explained in previous chapters. Two modifications—one major and one minor—were defined to assess these three software tools. The minor modification was the ability to consider input uncertainty for controllability measurements. Implementing this modification required almost the same effort for the GPL and DSL tools, but three times more for the MATLAB tool, while the performance of all three tools was similarly affected. Next, the superstructure selection was added as the major modification. This illustrated an improved, but on a minor scale, result for
the GPL tool compared with the DSL tool. However, the MATLAB tool required up to four times more effort to modify.

3. Chapter 7 tested the scalability of the DSL tool for controllability problems. Based on the DSL methodology, another software tool was developed for controllability analysis of an FCE. It was illustrated that, using the DSL methodology, the only fundamental difference for developing a software tool for different controllability case studies is the mathematical model used to analyse the case study. The effect of the model size on the effort required to develop the software tool was also explored.

8.2 Future work

As the idea of using DSL methodology for software development has recently gained attention, various elements should be carefully considered in order to make use of new guidelines in the process engineering domain. The researchers propose the following possible directions for further work and to highlight some open questions:

1. *Against other software quality measures.* Over the course of this study, with respect to the software quality assessment, four basic measures were applied: efficiency, usability, modifiability and scalability. In order to gain a more accurate evaluation of the proposed methodology, applying other software quality measures, such as portability, extendibility and overall performance of a software tool, is recommended. Moreover, during this study, one individual developed all the software tools. Thus, a survey-style research on a group of software developers could lead to a better understanding of the outcomes of the proposed methodology.

2. *Evaluate other controllability measures.* As stated in Chapter 3, RGA is only one of the important controllability measures. It is also important to evaluate the controllability using other linear and nonlinear controllability indices, such as condition number and singular value.

3. *Integrate other aspects of process engineering.* This study concentrated on the controllability analysis of a process system design. However, the
principles used in this research could be applied to either integrate other process engineering measures into current tools or develop new software tools for those measures. For example, economical assessment of a process system design might be a suitable candidate for this process, since no software tool is dedicated to this topic. Safety assessments are another potential focus for this purpose.

4. Study other types of DSL. MetaEdit+ was used in this research as the language workbench: a type of DSL tool for developing DSL tools. Although this tool offers features that improve productivity and maintainability, it is commercial, which limits its flexibility. As was pointed out in Chapter 2, various open-licence and open-source language workbenches are available. The benefit of using an open-source language workbench is the possibility of modifying the tool to generate code and integrate the solver tool directly and anonymously with the software product, thus increasing usability. Apart from language workbenches, which generate a new programming language, an internal DSL could also be used for process engineering programming tasks due to its massive tool support.
Appendix 1

This appendix shows the CSTR model using the OPPRR modelling tool:

<!-- First part of the model describes the objects in the model-->
<!-- This section of the model describes the stream objects. The main features are the properties (e.g. temperature, flowrate, concentration) of the stream. The inlet, outlet and coolant streams are inherited from stream -->
<object id="Object_InletStream_sysadmin" type="Object_InletStream_sysadmin" typeName="ProductOutlet">
  <superType>
    <object id="Object_Stream_user" type="Object_Stream_user" typeName="Stream">
      <slot id="a2wy9b" name="Name" unique="false">
      </slot>
      <slot id="a71q6d" name="Temperature" unique="false">
        <property type="Property_Temperature_user" typeName="Temperature">
          <description>hjhh</description>
          <dataType>
            <object id="Object_Variable_user" type="Object_Variable_user" typeName="Variable">
              <slot id="a0wwxn" name="Name" unique="false">
              </slot>
              <slot id="a31qlz" name="VariableType" unique="false">
                <property type="Property_VariableType_user" typeName="VariableType">
                  <dataType>
                    <simpleType>String</simpleType>
                    <widget>Editable List</widget>
                    <listValues>
                      <string>Constant</string>
                      <string>Measured Variable</string>
                      <string>Disterbance Variable</string>
                      <string>Controlled Variable</string>
                    </listValues>
                  </property>
                  </slot>
                </object>
              </slot>
              <slot id="a21mxh" name="Value" unique="false">
                <property id="Property_Value_user" type="Property_Value_user" typeName="Value">
                  <dataType>
                    <simpleType>Number</simpleType>
                    </object>
                </property>
              </slot>
            </property>
          </property>
        </property>
      </object>
    </superType>
  </object>
</object>
<!—- This section of the model describes the CSTR. Properties (e.g. temperature, concentration) as well as the mass and energy balance equations are contained in its model -->
Appendix 1

<!—- This section of the model describes the Mixer. -->
<object type="Object_Mixer_sysadmin" typeName="Mixer">
  <description></description>
  <superType>
    <object href="#Object_Unit_sysadmin"></object>
  </superType>
  <identProp slotID="ahz3x9"></identProp>
</object>

<!—- This section of the model describes the Splitter. The ratio between the outlet streams is specified in the model -->
<object id="Object_Splliter_sysadmin" type="Object_Splliter_sysadmin" typeName="Splliter"/>
Appendix 1

<description></description>
<superType>
<object href="#Object_Unit_sysadmin"></object>
</superType>
<slot id="a12c01" name="Portion" unique="false">
<property type="Property_Portion_sysadmin" typeName="Portion">
<description></description>
<dataType>
<simpleType>Number</simpleType>
</dataType>
.defaultValue>
<int>1</int>
</defaultValue>
</property>
</slot>
<identProp slotID="ahz3x9"></identProp>
</object>

<!— Inlet stream described here as a child of stream which was modelled previously. -->
<object id="Object_ProductInlet_sysadmin" type="Object_ProductInlet_sysadmin"
typeName="ProductInlet">
<description></description>
<superType>
<object href="#Object_Stream_user"></object>
</superType>
<slot id="a041ot" name="Percentage" unique="false">
<property href="#Property_Percentage_user"></property>
</slot>
<identProp slotID="a2wy9b"></identProp>
</object>

<!— Coolant stream described here as a child of stream which was modelled previously. -->
<object id="Object_CoolInput_sysadmin" type="Object_CoolInput_sysadmin"
typeName="CoolInput">
<description></description>
<superType>
<object href="#Object_Stream_user"></object>
</superType>
<identProp slotID="a2wy9b"></identProp>
</object>

<!— Auto-Splitter described here as a child of Splitter which was modelled previously. It also describes the different states that the auto-splitter can have. -->
<object type="Object_Automatic_Splitter_sysadmin" typeName="Automatic Splitter">
<description></description>
<superType>
<!—- Uncertainty described as an object in this model. It holds the value of the possible uncertainty that the disturbance variable can have -->

<object id="Object_Uncertainty_Value_sysadmin"
  type="Object_Uncertainty_Value_sysadmin" typeName="Uncertanty">
  <description></description>
  <slot id="ajs6p6" name="Value" unique="false">
    <property href="#Property_Value_user"></property>
  </slot>
  <identProp slotID="ajs6p6"></identProp>
</object>

<!—- Second part of the model describes the relationships between different objects -->

<!—- Cool stream defined as a relationship between the CSTR and on stream. -->

<relationship id="Relationship_Cool_in_user_3532248224"
  type="Relationship_Cool_in_user_3532248224" typeName="CoolStream">
  <description></description>
</relationship>

Stream:
<relationship id="Relationship_Input_sysadmin_3535645933"
  type="Relationship_Input_sysadmin_3535645933" typeName="streams">
  <description></description>
</relationship>
<!—- Input uncertainty is defined as a relationship between a stream and an uncertainty object. -->

<relationship id="Relationship_InputUncertainty_sysadmin" type="Relationship_InputUncertainty_sysadmin" typeName="InputUncertainty">
  <description></description>
  <slot id="a27mlm" name="Value" unique="false">
    <property href="#Property_Value_user"></property>
  </slot>
  <identProp slotID="a27mlm"></identProp>
</relationship>

<!—- Third part of the model describes the roles of different object in their relationships -->

Product Source:
<role id="Role_Source_user" type="Role_Source_user" typeName="ProductSource">
  <description></description>
</role>

<!—- Destenation -->

<role id="Role_Destenation_user" type="Role_Destenation_user" typeName="Destenation">
  <description></description>
</role>

<!—- Cool Destination -->

<role id="Role_CoolDestenation_sysadmin" type="Role_CoolDestenation_sysadmin" typeName="CoolDestenation">
  <description></description>
</role>

<!—- Cool Source-->  

<role id="Role_CoolSource_sysadmin" type="Role_CoolSource_sysadmin" typeName="CoolSource">
  <description></description>
</role>

<!—- Splitter down source-->  

<role type="Role_SplliterDownSource_sysadmin" typeName="SplliterDownSource">
  <description></description>
</role>

<!—- Splitter top source-->  

<role type="Role_SplliterTopSource_sysadmin" typeName="SplliterTopSource">
  <description></description>
</role>
Appendix 1

<!-- Uncertainty source -->

<role id="Role_UncertaintySource_sysadmin" type="Role_UncertaintySource_sysadmin" typeName="UncertaintySource">
  <description></description>
</role>

<!-- Uncertainty destination -->

<role id="Role_UncertaintyDestination_sysadmin" type="Role_UncertaintyDestination_sysadmin" typeName="UncertaintyDestination">
  <description></description>
</role>

<!-- Last part of the model illustrated the binding between different objects -->

(binding)
  <relationship href="#Relationship_Input_sysadmin"></relationship>
  <connection>
    <cardinality start="1" stop="1"></cardinality>
    <role href="#Role_Source_user"></role>
    <object href="#Object_ProductInlet_sysadmin"></object>
    <object href="#Object_Unit_sysadmin"></object>
    <port href="#Port_SplitterOutDown_sysadmin"></port>
    <port href="#Port_SplitterOutUp_sysadmin"></port>
    <port href="#Port_Stream_Output_user"></port>
    <port href="#Port_Stream_Output_user"></port>
  </connection>
  <connection>
    <cardinality start="1" stop="1"></cardinality>
    <role href="#Role_Destenation_user"></role>
    <object href="#Object_InletStream_sysadmin"></object>
    <object href="#Object_Unit_sysadmin"></object>
    <port href="#Port_MixerInDown_sysadmin"></port>
    <port href="#Port_MixerInUp_sysadmin"></port>
    <port href="#Port_Stream_Input_user"></port>
    <port href="#Port_CSTR_Input_Port_user"></port>
  </connection>
</binding>

(binding)
  <relationship href="#Relationship_InputUncertainty_sysadmin"></relationship>
  <connection>
    <cardinality start="1" stop="1"></cardinality>
    <role href="#Role_UncertaintySource_sysadmin"></role>
    <object href="#Object_Uncertanity_Value_sysadmin"></object>
  </connection>
</binding>
<cardinality start="1" stop="1"></cardinality>
<role href="#Role_UncertaintyDestination_sysadmin"></role>
<object href="#Object_ProductInlet_sysadmin"></object>
</connection>
</binding>
Appendix 2

This appendix shows the MATLAB script used to evaluate the parallel configuration of the CSTRs:

```matlab
function res=CalculateRGA(fid)

%%defining the CSTR variables
syms CoolIn1_F CoolIn2_F;
syms CSTR1_T CSTR1_Mc CSTR1_C CSTR1_Ti CSTR1_Ci;
DH=-5;
K0=2.7e8;
Qin_1=0.25;
Qin_2=0.25;
Cpc=1;
RoCp=1;
V=2.5;
E_R=6000;
Ua=0.35;
Tci=250;
%----------------------------------------------------------------------------------
%%defining CSTRs’ model equations
CSTR1_dC_dt = -K0*exp(-E_R/CSTR1_T)*CSTR1_C+ Qin_1*(CSTR1_Ci - CSTR1_C)/ V;
CSTR1_dT_dt = -DH*K0*exp(-E_R/CSTR1_T)*CSTR1_C+Qin_1*(CSTR1_Ti - CSTR1_T)/(Cpc*RoCp*V)-Ua/(Cpc*RoCp*V)*( CSTR1_T -Ua*CSTR1_T/(CSTR1_Mc*RoCp+Ua)- CSTR1_Mc*RoCp*Tci/(CSTR1_Mc*RoCp+Ua));
syms CSTR2_T CSTR2_Mc CSTR2_C CSTR2_Ti CSTR2_Ci;
CSTR2_dC_dt = -K0*exp(-E_R/CSTR2_T)*CSTR2_C+ Qin_2*(CSTR2_Ci - CSTR2_C)/ V;
CSTR2_dT_dt = -DH*K0*exp(-E_R/CSTR2_T)*CSTR2_C+Qin_2*(CSTR2_Ti - CSTR2_T)/(Cpc*RoCp*V)-Ua/(Cpc*RoCp*V)*( CSTR2_T -Ua*CSTR2_T/(CSTR2_Mc*RoCp+Ua)- CSTR2_Mc*RoCp*Tci/(CSTR2_Mc*RoCp+Ua));
rows_cnt=2;
columns_cnt=6;
inlet_T_pos=1;inlet_C_pos=2;
Inlet1_out_C=zeros(rows_cnt,columns_cnt);
Inlet1_out_C(1,inlet_T_pos)=1;
Inlet1_out_C(2,inlet_C_pos)=1;
%----------------------------------------------------------------------------------
%%setting up the flowsheet topology
Q=[];
CSTR1_T_pos=3;
CSTR1_C_pos=4;
Q=[Q,CSTR1_T;CSTR1_C];
CSTR2_T_pos=5;
CSTR2_C_pos=6;
Q=[Q,CSTR2_T;CSTR2_C];
CSTR1_out_C=zeros(rows_cnt,columns_cnt);
CSTR1_out_C(1,CSTR1_T_pos)=1;
CSTR1_out_C(2,CSTR1_C_pos)=1;
CSTR2_out_C=zeros(rows_cnt,columns_cnt);
CSTR2_out_C(1,CSTR2_T_pos)=1;
CSTR2_out_C(2,CSTR2_C_pos)=1;
Splliter1_outDown_C=zeros(rows_cnt,columns_cnt);
Splliter1_outUp_C=zeros(rows_cnt,columns_cnt);
Splliter1_outDown_C=Inlet1_out_C;
Splliter1_outUp_C=Inlet1_out_C;
```

176
Appendix 2

Mixer2_out_C=zeros(rows_cnt,columns_cnt);
Mixer2_out_C=(Qin_1*CSTR2_out_C+Qin_2*CSTR1_out_C)/(Qin_1+Qin_2);
Outlet1_C=Mixer2_out_C;
C=Outlet1_C(1:end,3:end);
CSTR1_C_In=Splliter1_outUp_C;
CSTR1_C_In=CSTR1_C_In(1:end,3:end);

% substituting the steady-state values in the equations
CSTR1_dC_dt=subs(CSTR1_dC_dt,{CSTR1_Ti,CSTR1_Ci},{CSTR1_C_In(1,1:end)*Q,CSTR1_C_In(2,1:end)*Q});
CSTR2_C_In=Splliter1_outDown_C;
CSTR2_C_In=CSTR2_C_In(1:end,3:end);
CSTR2_dC_dt=subs(CSTR2_dC_dt,{CSTR2_Ti,CSTR2_Ci},{CSTR2_C_In(1,1:end)*Q,CSTR2_C_In(2,1:end)*Q});

% linearizing the model
CSTR1_C_T_CSTR1=diff(CSTR1_dC_dt,CSTR1_T);
CSTR1_C_C_CSTR1=diff(CSTR1_dC_dt,CSTR1_C);
CSTR1_T_T_CSTR1=diff(CSTR1_dT_dt,CSTR1_T);
CSTR1_T_C_CSTR1=diff(CSTR1_dT_dt,CSTR1_C);
CSTR1_T_Mc_CSTR1=diff(CSTR1_dT_dt,CSTR1_Mc);
CSTR1_C_Mc_CSTR1=diff(CSTR1_dC_dt,CSTR1_Mc);
CSTR2_C_T_CSTR1=diff(CSTR2_dC_dt,CSTR1_T);
CSTR2_C_C_CSTR1=diff(CSTR2_dC_dt,CSTR1_C);
CSTR2_T_T_CSTR1=diff(CSTR2_dT_dt,CSTR1_T);
CSTR2_T_C_CSTR1=diff(CSTR2_dT_dt,CSTR1_C);
CSTR2_T_Mc_CSTR1=diff(CSTR2_dT_dt,CSTR1_Mc);
CSTR2_C_Mc_CSTR1=diff(CSTR2_dC_dt,CSTR1_Mc);

% calculating the state-space model
A=zeros(4,4);
B=zeros(4,2);
A(1,1)=subs(CSTR1_T_T_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
A(1,2)=subs(CSTR1_T_C_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
A(2,1)=subs(CSTR1_T_Mc_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
A(2,2)=subs(CSTR1_C_T_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
A(3,1)=subs(CSTR2_T_T_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
A(3,2)=subs(CSTR2_T_C_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
A(3,3)=subs(CSTR2_T_T_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{320.0080,0.7439,4.0e-1});
A(3,4)=subs(CSTR2_T_C_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{320.0080,0.7439,4.0e-1});
A(4,1)=subs(CSTR2_C_T_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
A(4,2)=subs(CSTR2_C_C_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
A(4,3)=subs(CSTR2_C_T_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{320.0080,0.7439,4.0e-1});
A(4,4)=subs(CSTR2_C_C_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{320.0080,0.7439,4.0e-1});

B(1,1)=subs(CSTR1_T_Mc_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
B(1,2)=subs(CSTR1_C_Mc_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
B(2,1)=subs(CSTR2_T_Mc_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
B(2,2)=subs(CSTR2_C_Mc_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
B(3,1)=subs(CSTR2_T_Mc_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{320.0080,0.7439,4.0e-1});
B(3,2)=subs(CSTR2_C_Mc_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{320.0080,0.7439,4.0e-1});
B(4,1)=subs(CSTR2_T_Mc_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
B(4,2)=subs(CSTR2_C_Mc_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{324.8090,0.5706,3.0e-1});
B(4,3)=subs(CSTR2_T_Mc_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{320.0080,0.7439,4.0e-1});
B(4,4)=subs(CSTR2_C_Mc_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{320.0080,0.7439,4.0e-1});
\[
\begin{align*}
&\text{B}(3,2) = \text{subs}(\text{CSTR2}\_\text{Mc}\_\text{CSTR2}, \{\text{CSTR2}\_\text{T}, \text{CSTR2}\_\text{C}, \text{CSTR2}\_\text{Mc}\}, \{320.0080, 0.7439, 4.0e-1\}) \\
&\text{B}(4,2) = \text{subs}(\text{CSTR2}\_\text{Mc}\_\text{CSTR2}, \{\text{CSTR2}\_\text{T}, \text{CSTR2}\_\text{C}, \text{CSTR2}\_\text{Mc}\}, \{320.0080, 0.7439, 4.0e-1\}) \\
\end{align*}
\]

%% saving the results in an output file
fprintf(fid, 'matrix A is:
');
fprintf(fid, '%2.2f  %2.2f  %2.2f  %2.2f  
');
fprintf(fid, 'matrix C is:
');
fprintf(fid, '%2.2f  %2.2f  %2.2f  %2.2f  
');
fprintf(fid, 'matrix B is:
');
fprintf(fid, '%2.2f  %2.2f  
');
D = zeros(2,2);
fprintf(fid, '%2.2f  %2.2f  
');
fprintf(fid, '****************************************************
');

%----------------------------------------------------------------------------------
%% setting up the transfer function
sys = ss(A, B, C, D);
[num, den] = tfdata(sys);
s = tf(sys);
message = evalc('s');
fprintf(fid, '****************************************************
');

num_el_size = size(num{1,1});
s0 = zeros(num_el_size(2), 1);
s0(num_el_size(2)) = 1;
% setting up the G matrix
G(1,1) = num{1,1} * s0 ./ den{1,1} * s0;
num_el_size = size(num{1,2});
s0 = zeros(num_el_size(2), 1);
s0(num_el_size(2)) = 1;
G(1,2) = num{1,2} * s0 ./ den{1,2} * s0;
num_el_size = size(num{2,1});
s0 = zeros(num_el_size(2), 1);
s0(num_el_size(2)) = 1;
G(2,1) = num{2,1} * s0 ./ den{2,1} * s0;
num_el_size = size(num{2,2});
s0 = zeros(num_el_size(2), 1);
s0(num_el_size(2)) = 1;
G(2,2) = num{2,2} * s0 ./ den{2,2} * s0;
%----------------------------------------------------------------------------------
%% Calculating Lambda matrix
L = G.*((inv(G)');
fprintf(fid, 'matrix L is:
');
fprintf(fid, '%2.2f  %2.2f  
');
fprintf(fid, '****************************************************
');

178
Appendix 3

This appendix shows one sample of the calculated steady-state, transfer function and \( \Lambda \) matrix of the four configurations illustrated in Figure 5.5-1:

**Configuration A:**
Header: Mc1=0.30, Mc2=0.40

matrix A is:
\[
\begin{bmatrix}
0.49 & 58.71 & 0.00 & 0.00 \\
-0.14 & -11.89 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.18 & 4.71 \\
0.00 & 0.00 & -0.06 & -0.99
\end{bmatrix}
\]

matrix C is:
\[
\begin{bmatrix}
0.75 & 0.00 & 0.25 & 0.00 \\
0.00 & 0.75 & 0.00 & 0.25
\end{bmatrix}
\]

matrix B is:
\[
\begin{bmatrix}
-12.06 & 0.00 \\
0.00 & 0.00 \\
0.00 & -5.06 \\
0.00 & 0.00
\end{bmatrix}
\]

matrix D is:
\[
\begin{bmatrix}
0.00 & 0.00 \\
0.00 & 0.00
\end{bmatrix}
\]

Transfer function from input 1 to output...
\[-9.045 s - 107.6 \]
\[\begin{align*}
\#1: \ & \frac{-9.045 s}{s^2 + 11.4 s + 2.446} \\
\#2: \ & \frac{-9.045 s}{s^2 + 11.4 s + 2.446}
\end{align*}\]

Transfer function from input 2 to output...
\[-1.265 s - 1.255 \]
\[\begin{align*}
\#1: \ & \frac{-1.265 s}{s^2 + 0.8171 s + 0.1087} \\
\#2: \ & \frac{-1.265 s}{s^2 + 0.8171 s + 0.1087}
\end{align*}\]

**Configuration B:**
Header: Mc1=0.30, Mc2=0.40

matrix A is:
\[
\begin{bmatrix}
0.49 & 58.71 & 0.00 & 0.00 \\
-0.14 & -11.89 & 0.00 & 0.00
\end{bmatrix}
\]
Appendix 3

0.15  0.00  -0.21  9.83
0.00  0.15  -0.00  -2.12

matrix C is:
0.00  0.00  2.00  0.00
0.00  0.00  0.00  2.00

matrix B is:
-12.06  0.00
0.00  0.00
0.00  -6.12
0.00  0.00

matrix D is:
0.00  0.00
0.00  0.00

Transfer function from input 1 to output...
#1:  ----------------------------
     s^4 + 13.73 s^3 + 29.47 s^2 + 11.1 s + 1.159
     0.5208 s + 0.1986
#2:  ----------------------------
     s^4 + 13.73 s^3 + 29.47 s^2 + 11.1 s + 1.159

Transfer function from input 2 to output...
-12.23 s - 25.87
#1:  ---------------------
     s^2 + 2.33 s + 0.4738
     0.02518
#2:  ---------------------
     s^2 + 2.33 s + 0.4738

Configuration C:
Header: Mc1=0.30, Mc2=0.40

matrix A is:
0.64  87.58  0.00  0.00
-0.18  -17.72  0.00  0.00
0.20  0.00  -0.26  19.88
0.00  0.20  -0.00  -4.18

matrix C is:
0.00  0.00  1.00  0.00
0.00  0.00  0.00  1.00

matrix B is:
-13.05  0.00
0.00  0.00
0.00  -7.21
0.00  0.00

matrix D is:
0.00  0.00
0.00  0.00

Transfer function from input 1 to output...
Appendix 3

\[-2.61 \, s^2 - 57.14 \, s - 183.7\]

\[\begin{align*}
\text{#1:} & \quad \frac{s^4 + 21.52 \, s^3 + 81.47 \, s^2 + 39.56 \, s + 5.16}{0.4773 \, s + 0.2317} \\
\text{#2:} & \quad \frac{s^4 + 21.52 \, s^3 + 81.47 \, s^2 + 39.56 \, s + 5.16}{0.4773 \, s + 0.2317}
\end{align*}\]

Transfer function from input 2 to output...
\[-7.205 \, s - 30.09\]

\[\begin{align*}
\text{#1:} & \quad \frac{s^2 + 4.44 \, s + 1.145}{0.01679} \\
\text{#2:} & \quad \frac{s^2 + 4.44 \, s + 1.145}{0.01679}
\end{align*}\]

******************************************************************************

matrix G is:
\[-35.61 \quad -26.29 \]
0.04 0.01

matrix L is:
-0.79 1.79
1.79 -0.79

Configuration D:
Header: Mc1=0.30, Mc2=0.40

matrix A is:
0.64 87.58 0.00 0.00
-0.18 -17.72 0.00 0.00
0.08 0.00 -0.14 13.59
0.00 0.08 -0.00 -2.79

matrix C is:
0.25 0.00 0.75 0.00
0.00 0.25 0.00 0.75

matrix B is:
-13.05 0.00
0.00 0.00
0.00 -6.61
0.00 0.00

matrix D is:
0.00 0.00
0.00 0.00

******************************************************************************

Transfer function from input 1 to output...
\[-3.26 \, s^3 - 68.05 \, s^2 - 185.9 \, s - 58.66\]

\[\begin{align*}
\text{#1:} & \quad \frac{s^4 + 20.01 \, s^3 + 55.04 \, s^2 + 20.36 \, s + 1.882}{0.5886 \, s^2 + 1.861 \, s + 0.288} \\
\text{#2:} & \quad \frac{s^4 + 20.01 \, s^3 + 55.04 \, s^2 + 20.36 \, s + 1.882}{0.5886 \, s^2 + 1.861 \, s + 0.288}
\end{align*}\]

Transfer function from input 2 to output...
\[-4.956 \, s - 13.85\]

\[\begin{align*}
\text{#1:} & \quad \frac{s^2 + 2.935 \, s + 0.4175}{0.008993} \\
\text{#2:} & \quad \frac{s^2 + 2.935 \, s + 0.4175}{0.008993}
\end{align*}\]

******************************************************************************
Appendix 3

matrix $G$ is:
\[
\begin{pmatrix}
-31.17 & -33.17 \\
0.15 & 0.02
\end{pmatrix}
\]

matrix $L$ is:
\[
\begin{pmatrix}
-0.15 & 1.15 \\
1.15 & -0.15
\end{pmatrix}
\]

Data for all configurations (for all chapters) is available at the following link:

https://www.dropbox.com/sh/roc1rnkdep0v0cw/AAD0-sDI10UCHZU4SFxvRmJga?dl=0
This appendix shows the controllability evaluation of the CSTR case study with various flowrate and different configurations:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Q1</th>
<th>Q2</th>
<th>Mc1</th>
<th>Mc2</th>
<th>T1</th>
<th>C1</th>
<th>T2</th>
<th>C2</th>
<th>ζ</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.375</td>
<td>0.125</td>
<td>0.350</td>
<td>0.350</td>
<td>351.304</td>
<td>0.286</td>
<td>310.673</td>
<td>0.864</td>
<td>1.364</td>
<td>0.133</td>
</tr>
<tr>
<td></td>
<td>0.375</td>
<td>0.125</td>
<td>0.370</td>
<td>0.330</td>
<td>350.373</td>
<td>0.300</td>
<td>311.857</td>
<td>0.806</td>
<td>1.433</td>
<td>0.188</td>
</tr>
<tr>
<td></td>
<td>0.375</td>
<td>0.125</td>
<td>0.330</td>
<td>0.370</td>
<td>352.307</td>
<td>0.273</td>
<td>309.587</td>
<td>0.922</td>
<td>1.310</td>
<td>0.096</td>
</tr>
<tr>
<td></td>
<td>0.375</td>
<td>0.125</td>
<td>0.300</td>
<td>0.400</td>
<td>353.965</td>
<td>0.252</td>
<td>308.115</td>
<td>1.007</td>
<td>1.246</td>
<td>0.061</td>
</tr>
<tr>
<td>b</td>
<td>0.375</td>
<td>0.375</td>
<td>0.350</td>
<td>0.350</td>
<td>351.304</td>
<td>0.286</td>
<td>319.983</td>
<td>0.021</td>
<td>1.728</td>
<td>0.529</td>
</tr>
<tr>
<td></td>
<td>0.375</td>
<td>0.375</td>
<td>0.370</td>
<td>0.330</td>
<td>350.373</td>
<td>0.300</td>
<td>320.045</td>
<td>0.021</td>
<td>1.728</td>
<td>0.531</td>
</tr>
<tr>
<td></td>
<td>0.375</td>
<td>0.375</td>
<td>0.400</td>
<td>0.300</td>
<td>349.093</td>
<td>0.319</td>
<td>320.299</td>
<td>0.022</td>
<td>1.731</td>
<td>0.535</td>
</tr>
<tr>
<td></td>
<td>0.375</td>
<td>0.375</td>
<td>0.330</td>
<td>0.370</td>
<td>352.307</td>
<td>0.273</td>
<td>320.066</td>
<td>0.020</td>
<td>1.727</td>
<td>0.529</td>
</tr>
<tr>
<td></td>
<td>0.375</td>
<td>0.375</td>
<td>0.300</td>
<td>0.400</td>
<td>353.965</td>
<td>0.252</td>
<td>320.201</td>
<td>0.018</td>
<td>1.728</td>
<td>0.529</td>
</tr>
<tr>
<td>c</td>
<td>0.500</td>
<td>0.500</td>
<td>0.350</td>
<td>0.350</td>
<td>360.181</td>
<td>0.251</td>
<td>332.501</td>
<td>0.012</td>
<td>1.794</td>
<td>0.630</td>
</tr>
<tr>
<td></td>
<td>0.500</td>
<td>0.500</td>
<td>0.370</td>
<td>0.330</td>
<td>359.358</td>
<td>0.261</td>
<td>331.336</td>
<td>0.013</td>
<td>1.798</td>
<td>0.637</td>
</tr>
<tr>
<td></td>
<td>0.500</td>
<td>0.500</td>
<td>0.400</td>
<td>0.300</td>
<td>358.223</td>
<td>0.275</td>
<td>329.747</td>
<td>0.015</td>
<td>1.805</td>
<td>0.649</td>
</tr>
<tr>
<td></td>
<td>0.500</td>
<td>0.500</td>
<td>0.330</td>
<td>0.370</td>
<td>361.065</td>
<td>0.241</td>
<td>333.762</td>
<td>0.011</td>
<td>1.790</td>
<td>0.624</td>
</tr>
<tr>
<td>d</td>
<td>0.500</td>
<td>0.500</td>
<td>0.300</td>
<td>0.400</td>
<td>362.519</td>
<td>0.226</td>
<td>332.714</td>
<td>0.011</td>
<td>1.794</td>
<td>0.630</td>
</tr>
<tr>
<td></td>
<td>0.500</td>
<td>0.375</td>
<td>0.350</td>
<td>0.350</td>
<td>360.181</td>
<td>0.251</td>
<td>325.941</td>
<td>0.013</td>
<td>1.152</td>
<td>0.023</td>
</tr>
<tr>
<td></td>
<td>0.500</td>
<td>0.375</td>
<td>0.370</td>
<td>0.330</td>
<td>359.358</td>
<td>0.261</td>
<td>326.124</td>
<td>0.013</td>
<td>1.151</td>
<td>0.023</td>
</tr>
<tr>
<td></td>
<td>0.500</td>
<td>0.375</td>
<td>0.400</td>
<td>0.300</td>
<td>358.223</td>
<td>0.275</td>
<td>326.557</td>
<td>0.014</td>
<td>1.148</td>
<td>0.022</td>
</tr>
<tr>
<td></td>
<td>0.500</td>
<td>0.375</td>
<td>0.330</td>
<td>0.370</td>
<td>361.065</td>
<td>0.241</td>
<td>325.841</td>
<td>0.013</td>
<td>1.153</td>
<td>0.023</td>
</tr>
<tr>
<td></td>
<td>0.500</td>
<td>0.375</td>
<td>0.300</td>
<td>0.400</td>
<td>362.519</td>
<td>0.226</td>
<td>325.845</td>
<td>0.012</td>
<td>1.152</td>
<td>0.023</td>
</tr>
</tbody>
</table>
Appendix 5

This appendix shows the MATLAB script used to evaluate the parallel configuration of CSTRs with ±20% disturbance in feed temperature:

```matlab
function res=CalculateRGA(fid)
%%defining the CSTR variables
syms CoolIn1_F CoolIn2_F ;
syms CSTR1_T CSTR1_Mc CSTR1_C CSTR1_Ti CSTR1_Ci;
DH=-5;
K0=2.7e8;
Qin_1=0.25;
Qin_2=0.25;
Cpc=1;
RoCp=1;
V=2.5;
E_R=6000;
Ua=0.35;
Tci=250;
%----------------------------------------------------------------------------------
%%defining CSTRs' model equations
CSTR1_dC_dt = -K0*exp(-E_R/CSTR1_T)*CSTR1_C+ Qin_1*(CSTR1_Ci - CSTR1_C)/ V;
CSTR1_dT_dt = -DH*K0*exp(-E_R/CSTR1_T)*CSTR1_C+Qin_1*(CSTR1_Ti - CSTR1_T)/(Cpc*RoCp*V)-Ua/(Cpc*RoCp*V)*( CSTR1_T -Ua*CSTR1_T/(CSTR1_Mc*RoCp+Ua)-
CSTR1_Mc*RoCp*Tci/(CSTR1_Mc*RoCp+Ua));
syms CSTR2_T CSTR2_Mc CSTR2_C CSTR2_Ti CSTR2_Ci;
CSTR2_dC_dt = -K0*exp(-E_R/CSTR2_T)*CSTR2_C+Qin_2*(CSTR2_Ci - CSTR2_C)/ V;
CSTR2_dT_dt = -DH*K0*exp(-E_R/CSTR2_T)*CSTR2_C+Qin_2*(CSTR2_Ti - CSTR2_T)/(Cpc*RoCp*V)-Ua/(Cpc*RoCp*V)*( CSTR2_T -Ua*CSTR2_T/(CSTR2_Mc*RoCp+Ua)-
CSTR2_Mc*RoCp*Tci/(CSTR2_Mc*RoCp+Ua));
rows_cnt=2;
columns_cnt=6;
inlet_T_pos=1;inlet_C_pos=2;
Inlet1_out_C=zeros(rows_cnt,columns_cnt);
Inlet1_out_C(1,inlet_T_pos)=1;
Inlet1_out_C(2,inlet_C_pos)=1;
%----------------------------------------------------------------------------------
%creating uncertainty
for uncertainty =-0.02:0.02:0.02
Inlet_tempreture = 300
Inlet_tempreture = Inlet_tempreture + Inlet_tempreture* uncertainty
%%setting up the flowsheet topology
Q=[];
CSTR1_T_pos=3;
CSTR1_C_pos=4;
Q=[Q;CSTR1_T;CSTR1_C];
CSTR2_T_pos=5;
CSTR2_C_pos=6;
Q=[Q;CSTR2_T;CSTR2_C];
CSTR1_out_C=zeros(rows_cnt,columns_cnt);
CSTR1_out_C(1,CSTR1_T_pos)=1;
CSTR1_out_C(2,CSTR1_C_pos)=1;
CSTR2_out_C=zeros(rows_cnt,columns_cnt);
CSTR2_out_C(1,CSTR2_T_pos)=1;
CSTR2_out_C(2,CSTR2_C_pos)=1;
Splliter1_outDown_C=zeros(rows_cnt,columns_cnt);
Splliter1_outUp_C=zeros(rows_cnt,columns_cnt);
Splliter1_outDown_C=Inlet1_out_C;
Splliter1_outUp_C=Inlet1_out_C;
```

184
Appendix 5

\[
\text{Mixer2\_out\_C} = \text{zeros(rows\_cnt,columns\_cnt)};
\text{Mixer2\_out\_C} = (\text{Qin\_1} \cdot \text{CSTR2\_out\_C} + \text{Qin\_2} \cdot \text{CSTR1\_out\_C}) / (\text{Qin\_1} + \text{Qin\_2});
\text{Outlet1\_C} = \text{Mixer2\_out\_C};
\text{CSTR1\_C\_In} = \text{Splliter1\_out\_Up\_C};
\text{CSTR1\_C\_In} = \text{CSTR1\_C\_In}(1:end,3:end);
\]

% substituting the steady-state values in the equations
\[
\text{CSTR1\_dC\_dt} = \text{subs(CSTR1\_dC\_dt,\{CSTR1\_Ti,CSTR1\_Ci\},\{\text{CSTR1\_C\_In}(1,1:end)\cdot Q,\text{CSTR1\_C\_In}(2,1:end)\cdot Q\};
\text{CSTR2\_dC\_dt} = \text{subs(CSTR2\_dC\_dt,\{CSTR2\_Ti,CSTR2\_Ci\},\{\text{CSTR2\_C\_In}(1,1:end)\cdot Q,\text{CSTR2\_C\_In}(2,1:end)\cdot Q\});
\]

% linearizing the model
\[
\text{CSTR1\_C\_T\_CSTR1} = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_T}\});
\text{CSTR1\_C\_C\_CSTR1} = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_C}\});
\text{CSTR1\_T\_T\_CSTR1} = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_T}\});
\text{CSTR1\_C\_T\_CSTR1} = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_C}\});
\text{CSTR1\_C\_M}\_C\_CSTR1 = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_M}\_C\});
\text{CSTR1\_T\_C\_CSTR1} = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_T}\));
\text{CSTR1\_C\_M}\_C\_CSTR1 = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_M}\_C\});
\text{CSTR1\_C\_T\_CSTR2} = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_T}\));
\text{CSTR1\_C\_C\_CSTR2} = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_C}\));
\text{CSTR1\_T\_T\_CSTR2} = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_T}\));
\text{CSTR1\_C\_T\_CSTR2} = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_C}\));
\text{CSTR1\_C\_M}\_C\_CSTR2 = \text{diff(CSTR1\_dC\_dt,\{\text{CSTR1\_M}\_C\});
\text{CSTR2\_C\_T\_CSTR1} = \text{diff(CSTR2\_dC\_dt,\{\text{CSTR1\_T}\));
\text{CSTR2\_C\_C\_CSTR1} = \text{diff(CSTR2\_dC\_dt,\{\text{CSTR1\_C}\));
\text{CSTR2\_T\_T\_CSTR1} = \text{diff(CSTR2\_dC\_dt,\{\text{CSTR1\_T}\));
\text{CSTR2\_C\_T\_CSTR1} = \text{diff(CSTR2\_dC\_dt,\{\text{CSTR1\_C}\));
\text{CSTR2\_C\_M}\_C\_CSTR1 = \text{diff(CSTR2\_dC\_dt,\{\text{CSTR1\_M}\_C\});
\text{CSTR2\_T\_T\_CSTR2} = \text{diff(CSTR2\_dC\_dt,\{\text{CSTR1\_T}\));
\text{CSTR2\_C\_T\_CSTR2} = \text{diff(CSTR2\_dC\_dt,\{\text{CSTR1\_C}\));
\text{CSTR2\_C\_M}\_C\_CSTR2 = \text{diff(CSTR2\_dC\_dt,\{\text{CSTR1\_M}\_C\});
\text{CSTR2\_C\_C\_CSTR2} = \text{diff(CSTR2\_dC\_dt,\{\text{CSTR1\_C}\));
\]

% calculating the state-space model
\[
\text{A} = \text{zeros}(4,4);
\text{B} = \text{zeros}(4,2);
\]

% calculating the new steady-state values considering uncertainty
\[
\text{CSTR1\_T\_ss, CSTR1\_C\_ss, CSTR1\_M}\_C\_ss = \text{calculate\_steady\_state(CSTR1\_dC\_dt, CSTR1\_dT\_dt, \{\text{CSTR1\_T, CSTR1\_M}\_C, CSTR1\_C, CSTR1\_T\_i, CSTR1\_C\_i\});
\text{CSTR2\_T\_ss, CSTR2\_C\_ss, CSTR2\_M}\_C\_ss = \text{calculate\_steady\_state(CSTR2\_dC\_dt, CSTR2\_dT\_dt, \{\text{CSTR2\_T, CSTR2\_M}\_C, CSTR2\_C, CSTR2\_T\_i, CSTR2\_C\_i\});
\]

\[
\text{A}(1,1) = \text{subs(CSTR1\_T\_T\_CSTR1, \{\text{CSTR1\_T, CSTR1\_C, CSTR1\_M}\_C\});
\text{A}(1,2) = \text{subs(CSTR1\_C\_T\_CSTR1, \{\text{CSTR1\_T, CSTR1\_C, CSTR1\_M}\_C\});
\text{A}(2,1) = \text{subs(CSTR1\_T\_T\_CSTR1, \{\text{CSTR1\_T, CSTR1\_C, CSTR1\_M}\_C\});
\text{A}(2,2) = \text{subs(CSTR1\_C\_T\_CSTR1, \{\text{CSTR1\_T, CSTR1\_C, CSTR1\_M}\_C\});
\text{B}(1,1) = \text{subs(CSTR1\_M}\_C\_CSTR1, \{\text{CSTR1\_T, CSTR1\_C, CSTR1\_M}\_C\});
\text{B}(1,2) = \text{subs(CSTR1\_C\_M}\_C\_CSTR1, \{\text{CSTR1\_T, CSTR1\_C, CSTR1\_M}\_C\});
\]
A(1,3)=subs(CSTR1_T_T_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
A(1,4)=subs(CSTR1_T_C_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
A(2,3)=subs(CSTR1_C_T_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
A(2,4)=subs(CSTR1_C_C_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
B(1,2)=subs(CSTR1_T_Mc_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
B(2,2)=subs(CSTR1_C_Mc_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
A(3,1)=subs(CSTR2_T_T_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{CSTR1_T_ss,CSTR1_C_ss,CSTR1_Mc_ss});
A(3,2)=subs(CSTR2_T_C_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{CSTR1_T_ss,CSTR1_C_ss,CSTR1_Mc_ss});
A(4,1)=subs(CSTR2_C_T_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{CSTR1_T_ss,CSTR1_C_ss,CSTR1_Mc_ss});
A(4,2)=subs(CSTR2_C_C_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{CSTR1_T_ss,CSTR1_C_ss,CSTR1_Mc_ss});
B(3,1)=subs(CSTR2_T_Mc_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{CSTR1_T_ss,CSTR1_C_ss,CSTR1_Mc_ss});
B(4,1)=subs(CSTR2_C_Mc_CSTR1,{CSTR1_T,CSTR1_C,CSTR1_Mc},{CSTR1_T_ss,CSTR1_C_ss,CSTR1_Mc_ss});
A(3,3)=subs(CSTR2_T_T_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
A(3,4)=subs(CSTR2_T_C_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
A(4,3)=subs(CSTR2_C_T_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
A(4,4)=subs(CSTR2_C_C_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
B(3,2)=subs(CSTR2_T_Mc_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss});
B(4,2)=subs(CSTR2_C_Mc_CSTR2,{CSTR2_T,CSTR2_C,CSTR2_Mc},{CSTR2_T_ss,CSTR2_C_ss,CSTR2_Mc_ss}); % saving the results in an output file
fprintf(fid,'matrix A is:\r\n');
fprintf(fid,'%2.2f  %2.2f  %2.2f  %2.2f  \r\n',A');
fprintf(fid,'
matrix C is :
');
fprintf(fid,'%2.2f  %2.2f  %2.2f  %2.2f  
',C');
fprintf(fid,'
matrix B is :
');
fprintf(fid,'%2.2f  %2.2f  
',B');
D=zeros(2,2);
fprintf(fid,'%2.2f  %2.2f  
',D');
fprintf(fid,'************************************************
');
%----------------------------------------------------------------------------------
%%setting up the transfer function
sys=ss(A,B,C,D);
[num den]=tfdata(sys);
s=tf(sys);
message=evalc('s');
fprintf(fid,'
');
fprintf(fid,'****************************************************
');
num_el_size=size(num{1,1});
s0=zeros(num_el_size(2),1);
s0(num_el_size(2))=1;
num_el_size=size(num{1,2});
s0=zeros(num_el_size(2),1);
s0(num_el_size(2))=1;
num_el_size=size(num{2,1});
s0=zeros(num_el_size(2),1);
s0(num_el_size(2))=1;
num_el_size=size(num{2,2});
s0=zeros(num_el_size(2),1);
s0(num_el_size(2))=1;
G(1,1)=num{1,1}*s0./den{1,1}*s0;
G(1,2)=num{1,2}*s0./den{1,2}*s0;
G(2,1)=num{2,1}*s0./den{2,1}*s0;
G(2,2)=num{2,2}*s0./den{2,2}*s0;

186
s0=zeros(num_el_size(2),1);
s0(num_el_size(2))=1;
G(2,2)=num{2,2}*s0./den{2,2}*s0;

% Calculating Lambda matrix
L=G.*(inv(G)');
fprintf(fid,'matrix G is:
');
fprintf(fid,'%2.2f  %2.2f 
',G');
fprintf(fid,'matrix L is:
');
fprintf(fid,'%2.2f  %2.2f 
',L');
fprintf(fid,'|
');
This appendix shows the controllability evaluation of the CSTR case study with ±5, ±12 and ±20 per cent uncertainties:

<table>
<thead>
<tr>
<th>AutoSplitter 1</th>
<th>AutoSplitter 2</th>
<th>Uncertainty</th>
<th>λ</th>
<th>RGA Index</th>
<th>Uncertainty</th>
<th>λ</th>
<th>RGA Index</th>
<th>Drift from 0 Uncertainty</th>
<th>Maximum Drift Percentage(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllDown</td>
<td>AllDown</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+5%</td>
<td>NaN</td>
<td>-5%</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>Half</td>
<td>AllDown</td>
<td>0%</td>
<td>-0.490</td>
<td>0.240</td>
<td>+5%</td>
<td>-0.470</td>
<td>0.221</td>
<td>+5%</td>
<td>-0.510</td>
</tr>
<tr>
<td>Equal</td>
<td>AllDown</td>
<td>0%</td>
<td>-2.740</td>
<td>7.508</td>
<td>+5%</td>
<td>-2.690</td>
<td>7.236</td>
<td>+5%</td>
<td>-2.790</td>
</tr>
<tr>
<td>Twice</td>
<td>AllDown</td>
<td>0%</td>
<td>2.580</td>
<td>2.496</td>
<td>+5%</td>
<td>2.500</td>
<td>2.250</td>
<td>+5%</td>
<td>2.610</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllDown</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+5%</td>
<td>NaN</td>
<td>NaN</td>
<td>-5%</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Half</td>
<td>0%</td>
<td>1.150</td>
<td>0.023</td>
<td>+5%</td>
<td>1.140</td>
<td>0.020</td>
<td>-5%</td>
<td>1.170</td>
</tr>
<tr>
<td>Half</td>
<td>Half</td>
<td>0%</td>
<td>-1.840</td>
<td>3.386</td>
<td>+5%</td>
<td>-1.760</td>
<td>3.098</td>
<td>-5%</td>
<td>-1.990</td>
</tr>
<tr>
<td>Equal</td>
<td>Half</td>
<td>0%</td>
<td>-5.170</td>
<td>26.729</td>
<td>+5%</td>
<td>-5.020</td>
<td>25.200</td>
<td>+5%</td>
<td>-5.370</td>
</tr>
<tr>
<td>Twice</td>
<td>Half</td>
<td>0%</td>
<td>3.720</td>
<td>7.398</td>
<td>+5%</td>
<td>3.590</td>
<td>6.708</td>
<td>+5%</td>
<td>3.810</td>
</tr>
<tr>
<td>AllUp</td>
<td>Half</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+5%</td>
<td>NaN</td>
<td>NaN</td>
<td>-5%</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Equal</td>
<td>0%</td>
<td>1.130</td>
<td>0.017</td>
<td>+5%</td>
<td>1.110</td>
<td>0.012</td>
<td>-5%</td>
<td>1.140</td>
</tr>
<tr>
<td>Half</td>
<td>Equal</td>
<td>0%</td>
<td>-3.670</td>
<td>13.469</td>
<td>+5%</td>
<td>-3.440</td>
<td>11.834</td>
<td>-5%</td>
<td>-3.990</td>
</tr>
<tr>
<td>Equal</td>
<td>Equal</td>
<td>0%</td>
<td>-8.149</td>
<td>66.260</td>
<td>+5%</td>
<td>-7.780</td>
<td>60.528</td>
<td>-5%</td>
<td>-8.650</td>
</tr>
<tr>
<td>Twice</td>
<td>Equal</td>
<td>0%</td>
<td>4.770</td>
<td>14.213</td>
<td>+5%</td>
<td>4.610</td>
<td>13.032</td>
<td>-5%</td>
<td>4.940</td>
</tr>
<tr>
<td>AllUp</td>
<td>Equal</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+5%</td>
<td>NaN</td>
<td>NaN</td>
<td>-5%</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Twice</td>
<td>0%</td>
<td>1.140</td>
<td>0.020</td>
<td>+5%</td>
<td>1.130</td>
<td>0.017</td>
<td>-5%</td>
<td>1.160</td>
</tr>
<tr>
<td>Half</td>
<td>Twice</td>
<td>0%</td>
<td>-8.989</td>
<td>80.640</td>
<td>+5%</td>
<td>-8.670</td>
<td>65.125</td>
<td>-5%</td>
<td>-10.540</td>
</tr>
<tr>
<td>Equal</td>
<td>Twice</td>
<td>0%</td>
<td>-16.540</td>
<td>273.572</td>
<td>+5%</td>
<td>-15.050</td>
<td>226.503</td>
<td>-5%</td>
<td>-18.820</td>
</tr>
<tr>
<td>Twice</td>
<td>Twice</td>
<td>0%</td>
<td>6.550</td>
<td>30.803</td>
<td>+5%</td>
<td>6.410</td>
<td>29.268</td>
<td>-5%</td>
<td>6.700</td>
</tr>
<tr>
<td>AllUp</td>
<td>Twice</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+5%</td>
<td>NaN</td>
<td>NaN</td>
<td>-5%</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>AllUp</td>
<td>0%</td>
<td>1.790</td>
<td>0.624</td>
<td>+5%</td>
<td>1.810</td>
<td>0.656</td>
<td>-5%</td>
<td>1.740</td>
</tr>
<tr>
<td>Half</td>
<td>AllUp</td>
<td>0%</td>
<td>20.460</td>
<td>379.470</td>
<td>+5%</td>
<td>24.090</td>
<td>533.148</td>
<td>-5%</td>
<td>17.250</td>
</tr>
<tr>
<td>Equal</td>
<td>AllUp</td>
<td>0%</td>
<td>22.910</td>
<td>480.048</td>
<td>+5%</td>
<td>26.750</td>
<td>663.063</td>
<td>-5%</td>
<td>19.460</td>
</tr>
<tr>
<td>Twice</td>
<td>AllUp</td>
<td>0%</td>
<td>17.620</td>
<td>276.224</td>
<td>+5%</td>
<td>20.250</td>
<td>372.104</td>
<td>-5%</td>
<td>15.180</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllUp</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+5%</td>
<td>NaN</td>
<td>NaN</td>
<td>-5%</td>
<td>NaN</td>
</tr>
<tr>
<td>AutoSplitter 1</td>
<td>AutoSplitter 2</td>
<td>Uncertainty</td>
<td>$\lambda$</td>
<td>RGA Index</td>
<td>Uncertainty</td>
<td>$\lambda$</td>
<td>RGA Index</td>
<td>Uncertainty</td>
<td>$\lambda$</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------</td>
<td>-------------</td>
<td>----------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>AllDown</td>
<td>AllDown</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>Half</td>
<td>AllDown</td>
<td>0%</td>
<td>1.620</td>
<td>0.384</td>
<td>+12%</td>
<td>1.550</td>
<td>0.303</td>
<td>-12%</td>
<td>1.710</td>
</tr>
<tr>
<td>Equal</td>
<td>AllDown</td>
<td>0%</td>
<td>7.600</td>
<td>43.560</td>
<td>+12%</td>
<td>7.350</td>
<td>40.323</td>
<td>-12%</td>
<td>7.940</td>
</tr>
<tr>
<td>Twice</td>
<td>AllDown</td>
<td>0%</td>
<td>2.020</td>
<td>1.040</td>
<td>+12%</td>
<td>1.920</td>
<td>0.846</td>
<td>-12%</td>
<td>2.190</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllDown</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+12%</td>
<td>0.000</td>
<td>NaN</td>
<td>-12%</td>
<td>0.000</td>
</tr>
<tr>
<td>AllDown</td>
<td>Half</td>
<td>0%</td>
<td>1.160</td>
<td>0.026</td>
<td>+12%</td>
<td>1.130</td>
<td>0.017</td>
<td>-12%</td>
<td>1.200</td>
</tr>
<tr>
<td>Half</td>
<td>Half</td>
<td>0%</td>
<td>3.290</td>
<td>5.244</td>
<td>+12%</td>
<td>3.020</td>
<td>4.080</td>
<td>-12%</td>
<td>3.780</td>
</tr>
<tr>
<td>Equal</td>
<td>Half</td>
<td>0%</td>
<td>14.200</td>
<td>174.240</td>
<td>+12%</td>
<td>12.850</td>
<td>140.423</td>
<td>-12%</td>
<td>17.060</td>
</tr>
<tr>
<td>Twice</td>
<td>Half</td>
<td>0%</td>
<td>2.690</td>
<td>2.856</td>
<td>+12%</td>
<td>2.530</td>
<td>2.341</td>
<td>-12%</td>
<td>2.930</td>
</tr>
<tr>
<td>AllUp</td>
<td>Half</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+12%</td>
<td>NaN</td>
<td>NaN</td>
<td>-12%</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Equal</td>
<td>0%</td>
<td>1.140</td>
<td>0.020</td>
<td>+12%</td>
<td>1.110</td>
<td>0.012</td>
<td>-12%</td>
<td>1.180</td>
</tr>
<tr>
<td>Half</td>
<td>Equal</td>
<td>0%</td>
<td>5.750</td>
<td>22.563</td>
<td>+12%</td>
<td>5.000</td>
<td>16.000</td>
<td>-12%</td>
<td>7.400</td>
</tr>
<tr>
<td>Equal</td>
<td>Equal</td>
<td>0%</td>
<td>25.360</td>
<td>593.410</td>
<td>+12%</td>
<td>20.500</td>
<td>380.250</td>
<td>-12%</td>
<td>42.270</td>
</tr>
<tr>
<td>Twice</td>
<td>Equal</td>
<td>0%</td>
<td>3.310</td>
<td>5.336</td>
<td>+12%</td>
<td>3.120</td>
<td>4.494</td>
<td>-12%</td>
<td>3.580</td>
</tr>
<tr>
<td>AllUp</td>
<td>Equal</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+12%</td>
<td>NaN</td>
<td>NaN</td>
<td>-12%</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Twice</td>
<td>0%</td>
<td>1.150</td>
<td>0.023</td>
<td>+12%</td>
<td>1.120</td>
<td>0.014</td>
<td>-12%</td>
<td>1.200</td>
</tr>
<tr>
<td>Half</td>
<td>Twice</td>
<td>0%</td>
<td>14.360</td>
<td>178.490</td>
<td>+12%</td>
<td>10.680</td>
<td>93.702</td>
<td>-12%</td>
<td>31.590</td>
</tr>
<tr>
<td>Equal</td>
<td>Twice</td>
<td>0%</td>
<td>122.380</td>
<td>14733.104</td>
<td>+12%</td>
<td>100.250</td>
<td>2425.563</td>
<td>-12%</td>
<td>84.070</td>
</tr>
<tr>
<td>Twice</td>
<td>Twice</td>
<td>0%</td>
<td>4.390</td>
<td>11.492</td>
<td>+12%</td>
<td>4.200</td>
<td>10.240</td>
<td>-12%</td>
<td>4.600</td>
</tr>
<tr>
<td>AllUp</td>
<td>Twice</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+12%</td>
<td>NaN</td>
<td>NaN</td>
<td>-12%</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>AllUp</td>
<td>0%</td>
<td>1.790</td>
<td>0.624</td>
<td>+12%</td>
<td>1.810</td>
<td>0.656</td>
<td>-12%</td>
<td>1.740</td>
</tr>
<tr>
<td>Half</td>
<td>AllUp</td>
<td>0%</td>
<td>17.830</td>
<td>283.249</td>
<td>+12%</td>
<td>16.070</td>
<td>628.505</td>
<td>-12%</td>
<td>11.490</td>
</tr>
<tr>
<td>Equal</td>
<td>AllUp</td>
<td>0%</td>
<td>19.400</td>
<td>338.560</td>
<td>+12%</td>
<td>27.870</td>
<td>721.997</td>
<td>-12%</td>
<td>12.760</td>
</tr>
<tr>
<td>Twice</td>
<td>AllUp</td>
<td>0%</td>
<td>14.380</td>
<td>179.024</td>
<td>+12%</td>
<td>20.000</td>
<td>361.000</td>
<td>-12%</td>
<td>9.870</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllUp</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+12%</td>
<td>NaN</td>
<td>NaN</td>
<td>-12%</td>
<td>0.000</td>
</tr>
<tr>
<td>AutoSplitter 1</td>
<td>AutoSplitter 2</td>
<td>Uncertainty</td>
<td>$\lambda$</td>
<td>RGA Index</td>
<td>Uncertainty</td>
<td>$\lambda$</td>
<td>RGA Index</td>
<td>Uncertainty</td>
<td>$\lambda$</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>-------------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>AllDown</td>
<td>AllDown</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+20%</td>
<td>NaN</td>
<td>-20%</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>Half</td>
<td>AllDown</td>
<td>0%</td>
<td>1.490</td>
<td>0.240</td>
<td>+20%</td>
<td>1.410</td>
<td>-20%</td>
<td>1.650</td>
<td>0.423</td>
</tr>
<tr>
<td>Equal</td>
<td>AllDown</td>
<td>0%</td>
<td>3.740</td>
<td>7.508</td>
<td>+20%</td>
<td>3.570</td>
<td>-20%</td>
<td>4.060</td>
<td>9.364</td>
</tr>
<tr>
<td>Twice</td>
<td>AllDown</td>
<td>0%</td>
<td>2.580</td>
<td>2.496</td>
<td>+20%</td>
<td>2.310</td>
<td>-20%</td>
<td>3.220</td>
<td>4.928</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllDown</td>
<td>0%</td>
<td>0.000</td>
<td>NaN</td>
<td>+20%</td>
<td>0.000</td>
<td>-20%</td>
<td>0.000</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Half</td>
<td>0%</td>
<td>1.150</td>
<td>0.023</td>
<td>+20%</td>
<td>1.100</td>
<td>-20%</td>
<td>1.230</td>
<td>0.053</td>
</tr>
<tr>
<td>Half</td>
<td>Half</td>
<td>0%</td>
<td>2.840</td>
<td>3.386</td>
<td>+20%</td>
<td>2.560</td>
<td>-20%</td>
<td>3.620</td>
<td>6.864</td>
</tr>
<tr>
<td>Equal</td>
<td>Half</td>
<td>0%</td>
<td>6.170</td>
<td>26.729</td>
<td>+20%</td>
<td>5.670</td>
<td>-20%</td>
<td>7.550</td>
<td>42.903</td>
</tr>
<tr>
<td>AllUp</td>
<td>Half</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+20%</td>
<td>NaN</td>
<td>-20%</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Equal</td>
<td>0%</td>
<td>1.130</td>
<td>0.017</td>
<td>+20%</td>
<td>1.080</td>
<td>-20%</td>
<td>1.210</td>
<td>0.044</td>
</tr>
<tr>
<td>Half</td>
<td>Equal</td>
<td>0%</td>
<td>4.670</td>
<td>13.469</td>
<td>+20%</td>
<td>3.980</td>
<td>-20%</td>
<td>7.210</td>
<td>38.564</td>
</tr>
<tr>
<td>Twice</td>
<td>Equal</td>
<td>0%</td>
<td>4.770</td>
<td>14.213</td>
<td>+20%</td>
<td>4.210</td>
<td>-20%</td>
<td>5.740</td>
<td>22.468</td>
</tr>
<tr>
<td>AllUp</td>
<td>Equal</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+20%</td>
<td>NaN</td>
<td>-20%</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>Twice</td>
<td>0%</td>
<td>1.140</td>
<td>0.020</td>
<td>+20%</td>
<td>1.090</td>
<td>-20%</td>
<td>1.230</td>
<td>0.053</td>
</tr>
<tr>
<td>Half</td>
<td>Twice</td>
<td>0%</td>
<td>9.980</td>
<td>80.640</td>
<td>+20%</td>
<td>7.460</td>
<td>-20%</td>
<td>38.730</td>
<td>1423.553</td>
</tr>
<tr>
<td>Equal</td>
<td>Twice</td>
<td>0%</td>
<td>17.540</td>
<td>273.572</td>
<td>+20%</td>
<td>13.510</td>
<td>-20%</td>
<td>68.580</td>
<td>4567.056</td>
</tr>
<tr>
<td>Twice</td>
<td>Twice</td>
<td>0%</td>
<td>6.550</td>
<td>30.803</td>
<td>+20%</td>
<td>5.960</td>
<td>-20%</td>
<td>7.020</td>
<td>36.240</td>
</tr>
<tr>
<td>AllUp</td>
<td>Twice</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+20%</td>
<td>NaN</td>
<td>-20%</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>AllDown</td>
<td>AllUp</td>
<td>0%</td>
<td>1.790</td>
<td>0.624</td>
<td>+20%</td>
<td>1.840</td>
<td>-20%</td>
<td>1.720</td>
<td>0.518</td>
</tr>
<tr>
<td>Half</td>
<td>AllUp</td>
<td>0%</td>
<td>20.480</td>
<td>379.470</td>
<td>+20%</td>
<td>38.620</td>
<td>-20%</td>
<td>9.140</td>
<td>66.260</td>
</tr>
<tr>
<td>Equal</td>
<td>AllUp</td>
<td>0%</td>
<td>22.910</td>
<td>480.048</td>
<td>+20%</td>
<td>41.970</td>
<td>-20%</td>
<td>10.600</td>
<td>92.160</td>
</tr>
<tr>
<td>Twice</td>
<td>AllUp</td>
<td>0%</td>
<td>17.620</td>
<td>276.224</td>
<td>+20%</td>
<td>30.670</td>
<td>-20%</td>
<td>8.850</td>
<td>61.623</td>
</tr>
<tr>
<td>AllUp</td>
<td>AllUp</td>
<td>0%</td>
<td>NaN</td>
<td>NaN</td>
<td>+20%</td>
<td>NaN</td>
<td>-20%</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Appendix 7

This appendix shows the solver code developed for Chapter 7:

function [ sys,G,Landa ] = Calc_Landa_Manual(x2_,p2_,f3,f1,x1,t1,t200)
%% Defininf the variabile and constants
syms x2 p2 p100 f200;

ua1=(f1+f3)*0.16;
ua2=6.84;
landa_=.38.5;
cp=0.07;
c=4;
m=20;

%%-------------------------------------------------------------------------
%% Defining the differential equations.
eq1=(1/m)*(f1*x1-x2*f1+x2*ua1*(0.1538*p100+90-0.5616*p2-0.3126*x2-48.43)/landa_-
x2*f1*cp*(0.5616*p2+0.3126*x2+48.43-t1)/landa_);
eq2=(1/c)*(ua1*(0.1538*p100+90-0.5616*p2-0.3126*x2-48.43)/landa_-
f1*cp*(0.5616*p2+0.3126*x2+48.43-t1)/landa_ -ua2*(0.507*p2+55-
t200)/(landa_*(1+ua2/(2*cp*f200))));

%%-------------------------------------------------------------------------
%% Calculating the steady-state-values of the manipulated variables.
 fid=fopen('calc_ss.m','w');
 fprintf(fid,'function F=calc_ss(x)
');
 fprintf(fid,'p100=x(1);
');
 fprintf(fid,'f200=x(2);
');
 s1=char(subs(eq1,{x2,p2},{x2_,p2_}));
 s2=char(subs(eq2,{x2,p2},{x2_,p2_}));
 fprintf(fid,'F(1)=%s;
',s1);
 fprintf(fid,'F(2)=%s;
',s2);
 fprintf(fid,'end');
 fclose(fid);

res=fsolve(@(calc_ss,[194, 208]);

p100 =res(1);
 f200 =res(2);

%%-------------------------------------------------------------------------
%% Linearizing the Model
 d_eq1_dx2=diff(eq1,x2);
 d_eq1_dp2=diff(eq1,p2);
 d_eq1_dp100=diff(eq1,p100);
 d_eq1_df200=diff(eq1,f200);
 d_eq2_dx2=diff(eq2,x2);
Appendix 7

d_eq2_dp2=diff(eq2,p2);
d_eq2_dp100=diff(eq2,p100);
d_eq2_df200=diff(eq2,f200);

%%-------------------------------------------------------------------------
%% Calculating the A, B, C and D Matrixes
a=zeros(2,2);
b=zeros(2,2);
c=[1 0;0 1];
d=[0 0;0 0];

a(1,1)=subs(d_eq1_dx2,{x2,p2,p100,f200},{x2_,p2_,p100_,f200_});
a(1,2)=subs(d_eq1_dp2,{x2,p2,p100,f200},{x2_,p2_,p100_,f200_});
a(2,1)=subs(d_eq2_dx2,{x2,p2,p100,f200},{x2_,p2_,p100_,f200_});
a(2,2)=subs(d_eq2_dp2,{x2,p2,p100,f200},{x2_,p2_,p100_,f200_});

b(1,1)=subs(d_eq1_dp100,{x2,p2,p100,f200},{x2_,p2_,p100_,f200_});
b(1,2)=subs(d_eq1_df200,{x2,p2,p100,f200},{x2_,p2_,p100_,f200_});
b(2,1)=subs(d_eq2_dp100,{x2,p2,p100,f200},{x2_,p2_,p100_,f200_});
b(2,2)=subs(d_eq2_df200,{x2,p2,p100,f200},{x2_,p2_,p100_,f200_});

%%-------------------------------------------------------------------------
%% defining the system and calculating the trasnfer function
sys=ss(a,b,c,d);

[num den]=tfdata(sys);
num_el_size=size(num{1,1});
s0=zeros(num_el_size(2),1);
s0(1)=1;
G(1,1)=num{1,1}*s0./den{1,1}*s0;
num_el_size=size(num{1,2});
s0=zeros(num_el_size(2),1);
s0(1)=1;
G(1,2)=num{1,2}*s0./den{1,2}*s0;
num_el_size=size(num{2,1});
s0=zeros(num_el_size(2),1);
s0(1)=1;
G(2,1)=num{2,1}*s0./den{2,1}*s0;
num_el_size=size(num{2,2});
s0=zeros(num_el_size(2),1);
s0(1)=1;
G(2,2)=num{2,2}*s0./den{2,2}*s0;

%%-------------------------------------------------------------------------
%% Calculating the RGA index.
Landa=G.*(inv(G)');
End
Appendix 8

This appendix shows the FCE model using the OPPRR modelling tool—please note that some of the objects are reused from the CSTR model:

<!-- First part of the model describes the objects in the model. -->
<!-- This section of the model describes the Splitter. The ratio between the outlet streams is specified in the model -->
<object type="Object_Splliter_sysadmin" typeName="Splliter">
<description></description>
<superType>
<object href="#Object_Unit_sysadmin"></object>
</superType>
<slot id="ai2c0l" name="Portion" unique="false">
<property type="Property_Portion_sysadmin" typeName="Portion">
<description></description>
<dataType>
<simpleType>Number</simpleType>
</dataType>
<defaultValue>
<int>1</int>
</defaultValue>
</property>
</slot>
<identProp slotID="ahz3x9"></identProp>
</object>

<!-- This section of the model describes the Mixer. -->
<object type="Object_Mixer__sysadmin" typeName="Mixer">
<description></description>
<superType>
<object href="#Object_Unit_sysadmin"></object>
</superType>
<identProp slotID="ahz3x9"></identProp>
</object>

<!-- This section of the model describes the Hot steam stream. Additional properties to normal stream is the pressure. -->
<object id="Object_HotInput_sysadmin" type="Object_HotInput_sysadmin" typeName="HotInput">
<description></description>
<superType>
<!—- Pump unit operation described in this section. Main property of a pump in this model is its output flowrate. -->
<object type="Object_Pump1_sysadmin" typeName="Pump">
  <description></description>
  <superType>
    <object href="#Object_Unit_sysadmin"></object>
  </superType>
  <slot id="a0z27r" name="PumpOutFlowrate" unique="false">
    <property type="Property_PumpOutFlowrate_sysadmin" typeName="PumpOutFlowrate">
      <description></description>
      <dataType>
        <object href="#Object_Variable_user"></object>
      </dataType>
    </property>
  </slot>
</object>

<!—- Condenser unit operation described in this section. Main property of a condenser in this model is its Ua. -->
<object id="Object_Condencer2_sysadmin" type="Object_Condencer2_sysadmin" typeName="Condenser">
  <description></description>
  <superType>
    <object href="#Object_Unit_sysadmin"></object>
  </superType>
  <slot id="a0z29z" name="Ua2" unique="false">
    <property type="Property_Ua_sysadmin" typeName="Ua">
      <description></description>
      <dataType>
        <object href="#Object_Variable_user"></object>
      </dataType>
    </property>
  </slot>
</object>
Appendix 8

Separator:
<object type="Object_Separator1_sysadmin" typeName="Separator">
  <description></description>
  <superType>
    <object href="#Object_Unit_sysadmin"></object>
  </superType>
  <slot id="a0z2t2" name="roA" unique="false">
    <property type="Property_roA_sysadmin" typeName="roA">
      <description></description>
      <dataType>
        <object href="#Object_Variable_user"></object>
      </dataType>
    </property>
  </slot>
  <identProp slotID="ahz3x9"></identProp>
</object>

<!-- Evaporator unit operation described in this section. Properties of the Evaporator are: the amount of mass in it (M), C constant (C) and the pressure inside it (P) -->
<object id="Object_Evaporator1_sysadmin" type="Object_Evaporator1_sysadmin" typeName="Evaporator">
  <description></description>
  <superType>
    <object id="Object_Unit_sysadmin" type="Object_Unit_sysadmin" typeName="Unit">
      <description></description>
      <slot id="ahz3x9" name="Name" unique="false">
        <property href="#Property_Name_user"></property>
      </slot>
      <identProp slotID="ahz3x9"></identProp>
    </object>
  </superType>
  <slot id="a2z2x5" name="C" unique="false">
    <property type="Property_C_sysadmin" typeName="C">
      <description></description>
      <dataType>
        <object href="#Object_Variable_user"></object>
      </dataType>
    </property>
  </slot>
  <identProp slotID="ahz3x9"></identProp>
</object>

<object id="Object_Evaporator1_sysadmin" type="Object_Evaporator1_sysadmin" typeName="Evaporator">
  <description></description>
  <superType>
    <object id="Object_Unit_sysadmin" type="Object_Unit_sysadmin" typeName="Unit">
      <description></description>
      <slot id="ahz3x9" name="Name" unique="false">
        <property href="#Property_Name_user"></property>
      </slot>
      <identProp slotID="ahz3x9"></identProp>
    </object>
  </superType>
  <slot id="a2z2x5" name="C" unique="false">
    <property type="Property_C_sysadmin" typeName="C">
      <description></description>
      <dataType>
        <object href="#Object_Variable_user"></object>
      </dataType>
    </property>
  </slot>
  <identProp slotID="ahz3x9"></identProp>
</object>

<object id="Object_Evaporator1_sysadmin" type="Object_Evaporator1_sysadmin" typeName="Evaporator">
  <description></description>
  <superType>
    <object id="Object_Unit_sysadmin" type="Object_Unit_sysadmin" typeName="Unit">
      <description></description>
      <slot id="ahz3x9" name="Name" unique="false">
        <property href="#Property_Name_user"></property>
      </slot>
      <identProp slotID="ahz3x9"></identProp>
    </object>
  </superType>
  <slot id="a2z2x5" name="C" unique="false">
    <property type="Property_C_sysadmin" typeName="C">
      <description></description>
      <dataType>
        <object href="#Object_Variable_user"></object>
      </dataType>
    </property>
  </slot>
  <identProp slotID="ahz3x9"></identProp>
</object>
<!—— condenser coolant output described here ——>
<object id="Object_ProductInlet_sysadmin" type="Object_ProductInlet_sysadmin" typeName="CoolOutput">
  <description></description>
  <superType>
    <object href="#Object_Stream_user"></object>
  </superType>
  <identProp slotID="a2wy9b"></identProp>
</object>

<!—— condenser coolant input described here ——>
<object id="Object_CoolInput_sysadmin" type="Object_CoolInput_sysadmin" typeName="CoolInput">
  <description></description>
  <superType>
    <object id="Object_Stream_user" type="Object_Stream_user" typeName="Stream">
      <description></description>
      <slot id="a2wy9b" name="Name" unique="false">
        <property href="#Property_Name_user"></property>
      </slot>
      <slot id="a71q6d" name="Tempreature" unique="false">
        <property type="Property_Tempreture_user" typeName="Tempreature">
          <description>hjhh</description>
          <dataType>
            <object id="Object_Variable_user" type="Object_Variable_user" typeName="Variable">
              <description></description>
              <slot id="a0wwxn" name="Name" unique="false">
                <property href="#Property_Name_user"></property>
              </slot>
              <slot id="a3lqlr" name="VariableType" unique="false">
                <property type="Property_VariableType_user" typeName="VariableType">
                  <description></description>
                  <dataType>
                    <simpleType>String</simpleType>
                  </dataType>
                  <widgetEditable List</widget>
                  <listValues>
                    <string>Constant</string>
                  </listValues>
                </property>
              </slot>
            </object>
          </dataType>
        </property>
      </description>
    </object>
  </superType>
  <identProp slotID="a2wy9b"></identProp>
</object>
<!—- Second part of the model describes the relationships between different objects -->

<!—- Cool stream defined as a relationship between the Condenser and the coolant stream. -->

<relationship id="Relationship_Cool_in_user_3532248224" type="Relationship_Cool_in_user_3532248224" typeName="CoolStream">
  <description></description>
</relationship>

<relationship id="Relationship_Input_sysadmin_3535645933" type="Relationship_Input_sysadmin_3535645933" typeName="streams">
  <description></description>
</relationship>

<!—- Hot stream relationship described here. This relationship is used to connect evaporator to the inlet hot stream-->

<relationship id="Relationship_Hot_Stream_sysadmin" type="Relationship_Hot_Stream_sysadmin" typeName="HotStream">
  <description></description>
</relationship>

<!—- Third part of the model describes the roles of different object in their..."/>
relationships -->

<!-- Cool Destination -->

<role id="Role_CoolDestination_sysadmin" type="Role_CoolDestination_sysadmin" typeName="CoolDestination">
  <description></description>
</role>

<!-- Cool Source-->

<role id="Role_CoolSource_sysadmin" type="Role_CoolSource_sysadmin" typeName="CoolSource">
  <description></description>
</role>

<!-- Hot destination-->

<role id="Role_HotDestination_sysadmin" type="Role_HotDestination_sysadmin" typeName="HotDestination">
  <description></description>
  <superType>
    <role href="#Role_Destenation_user"></role>
  </superType>
</role>

<!-- Hot source-->

<role id="Role_HotSource_sysadmin" type="Role_HotSource_sysadmin" typeName="HotSource">
  <description></description>
</role>

<!-- Last part of the model illustrated the binding between different objects-->

<binding>
  <relationship href="#Relationship_Hot_Stream_sysadmin"></relationship>
  <connection>
    <cardinality start="1" stop="1"></cardinality>
  </connection>
</binding>
Appendix 9

This appendix shows one sample of the calculated steady-state, transfer function and \( \Lambda \) matrix of the case study explored in Chapter 7:

\[
U = \begin{bmatrix} \text{Hot1_pressure} & \text{Coolin_F} \end{bmatrix}
\]
\[
Y = \begin{bmatrix} \text{Ev1_pressure} & \text{Outlet_C} \end{bmatrix}
\]

Disterbance vars
Coolin_T 25.00, Inlet_T 36.00, Inlet_F 9.30, Inlet_C 4.25,
\( x_2 = 6.84 \), \( m = 20.00 \), \( c = 4.00 \), \( cp = 0.07 \), \( \text{landa}_2 = 38.50 \), \( ual = 9.49 \), \( f200 = 208.00 \), \( p100 = 194.70 \)

Manipulated variable steady-state is
\( p100 = 190.26 \), \( f200 = 174.62 \)

matrix \( A \) is
\[
\begin{bmatrix}
-0.1849 & -0.1820 & -0.0546 & -0.0206
\end{bmatrix}
\]

matrix \( B \) is
\[
\begin{bmatrix}
0.0474 & 0.0000 \\
0.0095 & -0.0024
\end{bmatrix}
\]

matrix \( C \) is
\[
\begin{bmatrix}
1.0000 & 0.0000 \\
0.0000 & 1.0000
\end{bmatrix}
\]

matrix \( D \) is
\[
\begin{bmatrix}
0.0000 & 0.0000 \\
0.0000 & 0.0000
\end{bmatrix}
\]

Transfer function from input 1 to output...

\#1:
\[
\frac{0.04738 s - 0.0007491}{s^2 + 0.2055 s - 0.006124}
\]

\#2:
\[
\frac{0.009476 s - 0.0008337}{s^2 + 0.2055 s - 0.006124}
\]

Transfer function from input 2 to output...

\#1:
\[
\frac{0.0004396}{s^2 + 0.2055 s - 0.006124}
\]

\#2:
\[
\frac{-0.002416 s - 0.0004466}{s^2 + 0.2055 s - 0.006124}
\]

matrix \( G \) is
\[
\begin{bmatrix}
0.1223 & -0.0718 \\
0.1361 & 0.0729
\end{bmatrix}
\]

Landa is
<table>
<thead>
<tr>
<th>0.4772</th>
<th>0.5228</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5228</td>
<td>0.4772</td>
</tr>
</tbody>
</table>

RGA is a:
0.27332