Modular Model Specification on the Documentation Level
Concepts and Application in a Web-Based Modeling Environment

vorgelegt von
Dipl.-Ing. Stefan Kuntsche
aus Neustrelitz

von der Fakultät III - Prozesswissenschaften
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing.-

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr.-Ing. Matthias Kraume
Berichter: Prof. Dr.-Ing. Günter Wonzy
Berichter: Prof. Dr.-Ing. Flavio Manenti

Tag der wissenschaftlichen Aussprache: 5. April 2013

Berlin 2013
D83
Preface

The work presented here was completed during my time as a research staff member in the research group of Process Dynamics and Operation of the Technische Universität Berlin. The project was supported by the UNICAT Cluster of Excellence.

First, I would like to thank Prof. Dr.-Ing. Günter Wozny for supervising my work and for his many suggestions and inspiring discussions. He is responsible for creating an academic and industry-oriented environment where students can learn and achieve. I would also like to thank Prof. Dr.-Ing. Flavio Manenti for his cooperation, interest in my work and for serving as the second reviewer for this thesis.

Furthermore, I offer my thanks and gratitude to all my colleagues in the research group of Process Dynamics and Operation for the multiple inspiring discussions, shared ideas and productive criticism; their influence has contributed to the development of this work.

I would also like to thank Robert Kraus and Alejandro Merchan Resprepo for their pleasant and committed cooperation, and for continuing the MOSAIC project. Also, I would like to thank Tilman Barz and Karim Alloula for sharing ideas that helped me improve the concept. For realizing important extensions to the software, I want to thank Moritz Büchel, Jonas Bucher, and Tomasz Porst. Furthermore, I would like to express my gratitude to all MOSAIC users. They have helped demonstrate and improve the functioning of the software.

The support of my family should also be noted here. France and Alan, thank you very much for your patience, understanding, and encouragement.
Abstract

In process systems engineering, the formulation of models as equation systems is a common technique. Such models are used in research but are also present as integral parts of simulation tools. Several challenges can be seen in the corresponding modeling process. First, the computer models depend on their language of implementation, which renders them useless when technology advances and which hinders their application in environments that use other languages. Second, the documentation of computer models is still not satisfactory. Third, Internet technology should be used in a more effective way for the exchange and reuse of the computer models.

The definition of computer models in a meta language in combination with code generation for their implementation is a common concept in model-driven software engineering. In order to maximize usability, a meta language for equation-system-based computer models must be declarative, focus on the essential information, and also be intuitive. This can be achieved by defining the models on the level of documentation with the help of mathematical symbolic language as it is used in the literature. Modularity makes models and model elements reusable and thereby constitutes the basis for exchange and cooperation. The reusability of model elements is related to their independence from each other, from which follows that both the degree of modularity and the way model elements are combined play an important role. The applicability of the models in the form of program code is a direct measure of the benefit of a model-driven modeling approach. Therefore, a code generation feature should comprise many languages and be easily extensible.

In this work, the above subjects are addressed. A modeling concept is presented that works on the level of documentation, uses a complex mathematical symbolic notation, and is independent from storage formats. The class of considered mathematical notations includes modular variable names and is therefore more comparable to presentations in the literature than notations currently used in computer algebra systems. The aspect of modularity of the proposed modeling concept is focused on a high level of reusability and a fine-grained subdivision of the entire simulation problem. The described concepts are applied in a web-based modeling environment that permits the creation and the calculation of models and can be used as an exchange and cooperation platform. The modeling environment includes a versatile code generation functionality with a growing number of implemented language outputs, and a graphical interface for user-defined specification of the generated code.

Application examples of the modeling environment are provided, which also demonstrate the applicability of the modeling concept for scanned articles. The modeling environment includes a documentation generator, which is used to describe the examples presented.

This research was conducted for the UniCat cluster of excellence, which is an interdisciplinary project concerned with the oxidative coupling of methane. The activities in
UniCat include modeling at different levels of detail performed by different universities and research centers. In this context, the modeling process needed to be improved with respect to collaborative generation and use of models. Accordingly, developed equations and partial models must be reusable, and sub models of different detail must be exchangeable according to the desired modeling purpose. Furthermore, the models must be accessible from different locations and available in different numerical platforms. The author acknowledges the support of his research by the German Research Foundation (DFG).
**Kurzfassung**


Anwendungsbeispiele der Modellierungsumgebung werden vorgestellt, welche auch die Anwendbarkeit des Modellierungskonzepts auf gescannte Artikel zeigen. Die Modellierungsumgebung verfügt über einen Dokumentationsgenerator, welcher verwendet wird, um die Beispiele zu beschreiben.

Contents

I. Concepts 1

1. Introduction 2
   1.1. Motivation ............................................. 2
   1.2. Outline .................................................. 2
   1.3. Use of models in process systems engineering .............. 2
   1.4. Representation of equation-based models in PSE .......... 3
       1.4.1. Flowsheet environments ............................ 4
       1.4.2. Expert systems .................................... 4
       1.4.3. Programming languages ............................. 5
       1.4.4. Classification of the work presented here .......... 7
   1.5. Considered problems and numerical software .............. 7
   1.6. Documentation ........................................... 8
   1.7. Model specification vs. documentation ..................... 9
       1.7.1. Model content vs. presentation .................. 9
       1.7.2. General assumptions and PSE model representation .. 11

2. Overview of the covered concepts 12
   2.1. Identification of research aims .......................... 12
       2.1.1. The main goals and proposed concepts ................ 12
       2.1.2. Resulting fields of interest ....................... 14
       2.1.3. Order of elaboration ............................... 14
   2.2. Orientation with respect to similar projects .............. 14
       2.2.1. Tools that work on a meta level .................. 14
       2.2.2. Tools that work on the documentation level ......... 15
       2.2.3. Tools related to computer algebra systems .......... 15
       2.2.4. Orientation with respect to the preceding project .... 15

3. Documentation-level modeling 17
   3.1. Motivation ............................................... 17
       3.1.1. The need for and the challenges of documentation .... 17
       3.1.2. A visualization of the modeling process ............ 18
   3.2. Principle aim and used techniques ........................ 19
       3.2.1. Considered characteristics of a documentation-level description 20
       3.2.2. The aspect of model-driven software engineering .... 20
   3.3. Platform-independent models in the documentation-level ... 21
       3.3.1. Involved models .................................... 22
### Contents

3.3.2. Involved transformations ........................................ 24  
3.3.3. The challenge of diversity ....................................... 24  
3.4. In-depth consideration of algebraic expressions ..................... 26  
  3.4.1. Symbolic notations .......................................... 27  
  3.4.2. Documentation-level formats .................................. 33  
  3.4.3. Parsing based on documentation-level formats .................. 35  
  3.4.4. Transfer between documentation-level formats ................. 39  
  3.4.5. Combined application ......................................... 42  
3.5. Discussion .......................................................... 44  
  3.5.1. Compatibility of models in the documentation level .......... 44  
  3.5.2. Effort of exchange ............................................. 44  
4. Modular modeling ..................................................... 46  
  4.1. Supported mathematical problems ................................. 46  
  4.2. Model information within mathematical expressions .............. 47  
  4.3. Equations and functions as building blocks ....................... 48  
  4.4. Domain-specific concepts ....................................... 49  
    4.4.1. Documentation specific point of view ...................... 49  
  4.5. Use of the documentation-level models .......................... 50  
  4.6. Conclusion ....................................................... 50  
5. Use of the Internet .................................................. 52  
  5.1. Application in MOSAIC ........................................... 52  
II. Application ......................................................... 53  
6. Symbolic mathematic expressions ....................................... 54  
  6.1. The symbolic notation used in MOSAIC ........................... 54  
    6.1.1. Introduction with the help of examples ................... 54  
    6.1.2. The data model ............................................. 55  
    6.1.3. The symbolic representation .................................. 59  
    6.1.4. Application of the methodology ............................. 60  
  6.2. Storage formats .................................................. 62  
  6.3. Specialized storage formats ..................................... 63  
    6.3.1. MosaicLatex .................................................. 63  
    6.3.2. MosaicMathML .................................................. 64  
7. Modular modeling approach ............................................ 68  
  7.1. Model elements .................................................. 68  
  7.2. Modular reuse concepts ......................................... 80  
    7.2.1. Connection Techniques ...................................... 80  
8. Generation of program code and documentation ............................ 83  
  8.1. Introduction .................................................... 83  
  8.2. General applied principles ..................................... 83  
    8.2.1. The involved models ....................................... 83
### Contents

8.2.2. The software structure and extensibility .......................... 89
8.3. Representation of problem-specific interfaces .......................... 90
8.4. User-defined language specification ................................. 94
  8.4.1. Variables and non-linear algebraic expressions ................. 94
  8.4.2. Definition of the complete code fragment ...................... 95
8.5. Generation of documentation ........................................ 97
  8.5.1. Requirements and complexity .................................. 97
  8.5.2. The implemented approach .................................... 97

### III. Examples

9. Examples 106
  9.1. Transfer between documentation-level formats .................... 106
  9.2. Use of terminologic notations .................................... 110
  9.3. Case study within the UniCat research project ................... 112
  9.4. Modular modeling and reuse ...................................... 115
    9.4.1. A basic flash .............................................. 115
    9.4.2. Reuse and augmentation .................................... 116
    9.4.3. Use of functions and specific interfaces .................. 117
    9.4.4. Techniques for combining equation systems ................ 117
    9.4.5. Overview of calculation results ............................ 118
    9.4.6. Object-oriented point of view ............................. 119
  9.5. A distillation column model .................................... 120

### IV. Conclusion

10. Conclusion 124
  10.1. Practical experience .......................................... 124
  10.2. Related publications .......................................... 125
  10.3. Outlook ...................................................... 125
    10.3.1. Current research projects ............................... 125
    10.3.2. Potential directions and ongoing development ............ 125

Bibliography 127

### V. Annex

11. Documentation-level modeling 139
  11.1. Additional proofs .............................................. 139
  11.2. EBNF of notation elements in mathematic expressions in MOSAIC . 141

12. Detailed description of connection strategies 143
Contents

13. Generated documentation for selected example cases 156
   13.1. Basic flash ................................................................. 156
   13.2. Flash raoult antoine a ................................................. 161
   13.3. Flash raoult antoine b ............................................... 164
   13.4. Flash raoult dippr .................................................... 170
   13.5. Flash raoult function DIPPR ......................................... 177
   13.6. Flash raoult function CAPE-OPEN .................................... 180
   13.7. Flash sequence enc ................................................... 183
   13.8. Flash ports .............................................................. 184
   13.9. Flash sequence ports .................................................. 185
   13.10. Flash including heat balance ...................................... 186
   13.11. Distillation column .................................................. 189
   13.12. Generated program code for selected example cases .......... 198
      13.12.1. Simple Flash ..................................................... 198
      13.12.2. Use of functions ............................................... 201
      13.12.3. Use of CAPE-OPEN interfaces ............................... 204
   13.13. Generated or imported MathML code ............................... 209
List of Tables

6.1. Symbolic notation for the representation of operators. 59
6.2. Additionally used symbols. 59
6.3. Symbolic notation of index instantiation directives. 60
6.4. Symbolic notation of numbers. 60
6.5. Supported MathML tags in MosaicMathML. 65

8.1. Implemented platform models. 89
8.2. Example definitions of fix elements. 95

9.1. Tools used for the creation of mathematical expressions. 107
9.2. Tools used for the transfer between documentation-level formats. 107
9.3. OCM membrane reactor reactions and parameters. 112
9.4. Calculation results of the presented flash models. 119
List of Figures

3.1. Model creation based on documentation. ........................................ 18
3.2. Manual application of changes. .................................................. 18
3.3. Automated application of changes based on documentation. .......... 19
3.4. Translations in MDA. ............................................................... 21
3.5. Model representations and desired workflow. ............................ 22
3.6. Creation of a model on the abstract level. ................................... 22
3.7. Complexity of the model representation on the documentation level. 23
3.8. Model representation on the implementation level. ...................... 24
3.9. Use of models for documentation and implementation. ................. 25
3.10. Use of mathematical models in a tool-specific format. ................. 25
3.11. Use of existing tools and documentation-level models. ................. 26
3.12. Transfer and use of documentation-level models. ....................... 26
3.13. Basic offset patterns with respect to one line. .......................... 34
3.14. Resulting additional offset patterns. ......................................... 34
3.15. Tree representations of an algebraic expression. ......................... 36
3.16. Tree representations of a signed algebraic expression. ................. 38
3.17. Nested lines according to the existing patterns. ......................... 40
3.18. Depth of nesting and contained patterns. .................................. 40
3.19. Use of an algebraic expression after transfer steps. .................... 43
3.20. Number of tools and transfer steps in different configurations. .... 45

6.1. Elements of a variable naming. ............................................... 55
6.2. Equations in the data model. .................................................. 55
6.3. Applied data model for algebraic expressions. ............................ 56
6.4. In-line operators and functions in the data model. ....................... 57
6.5. Data model of possible variable names. ..................................... 58
6.6. Data model of unsigned real numbers. ..................................... 58
6.7. MosaicMathML subscript environment. ...................................... 65
6.8. MosaicMathML and Variable Namings. ..................................... 66
6.9. MosaicMathML and brackets. .................................................. 67
6.10. MosaicMathML and enclosing operator environments. ................ 67

7.1. UML representation the Notation. ........................................... 69
7.2. UML representation of the Equation. ....................................... 69
7.3. UML representation of the Equation System. ............................ 70
7.4. UML representation of Variable and Variable Naming. ................. 71
7.5. UML representation of the composition of Equation Systems. .......... 71
7.6. UML representation of the Connector. ...................................... 72
7.7. UML representation of the Interface. ....................................... 73
7.8. UML representation of the Function in the general case. ............... 73
7.9. Function usage. ............................................. 75
7.10. Combined use of Function and Interface. .......................... 76
7.11. Equation Systems representing process units. ....................... 76
7.12. Equation System in the context of flow sheeting. .................... 77
7.13. UML representation of the Parameter List. .......................... 77
7.14. UML representation of the index maximum value specification. .... 78
7.15. UML representation of the Variable Specification List. .............. 78
7.16. UML representation of the Evaluation. ................................ 79
7.17. Reuse in the creation of Equation Systems. ........................ 80
7.18. Four cases of reuse. ........................................... 81

8.1. Transformation and involved models. .................................. 84
8.2. Documentation-level model. ........................................ 85
8.3. Implementation-level data model NLE. .................................. 85
8.4. Implementation-level data model DAE. .................................. 86
8.5. UML representation of the Content Tree. ............................... 87
8.6. Implementation-level data model and code fragments. ................. 88
8.7. Basic model elements involved in code generation. .................... 89
8.8. Support matrix over interfaces and platforms. ........................ 91
8.9. UML representation of the Specific Interface. ........................ 91
8.10. Specific interface example: Activity coefficient. ..................... 92
8.11. User’s perspective of Interface Specifications. ....................... 93
8.12. UML representation of a Block ...................................... 96
8.13. Software structure of documentation generation. ....................... 97
8.15. Language Specificator, heritage tree and realized interfaces. ....... 99
8.16. Definition of specific interfaces. ................................... 100
8.17. Definition of special-purpose language specificators. ............... 101
8.18. User-defined language specification: Block classes. .................. 102
8.19. Example of generated Matlab code. .................................. 103
8.20. Separation of a given Matlab code into blocks. ....................... 104

9.1. Possible translation steps with the implemented software. ............. 106
9.2. Sample expressions created in Microsoft Word 2007. .................. 107
9.3. Example expression created in MathType. ................................ 108
9.4. Example expressions created with Firemath. ................................ 109
9.5. Comparability of representation in the literature and in MOSAIC. ...... 111
9.6. Problem specification supported by nomenclature information. ....... 111
9.7. Results in MOSAIC, Matlab, and ACM. MOSAIC. ....................... 113
9.8. Connection structure of the equation system ‘basic_flash_steady’. .... 116
9.9. Connection structure of ‘flash_raoult_antoine_a’ and ‘flash_raoult’. .... 116
9.10. Connection structure of ‘flash_raoult_antoine_b’ and ‘flash_raoult_dippr’. 117
9.11. Con. structure of ‘flash_raoult_fun_dippr’ and ‘flash_raoult_fun_co’. 118
9.12. Sequence of flashes: Con. structure and alternatives. ............... 118
9.13. Connection structures of a flash sequence. .......................... 118
9.15. Instantiation of an equation system. .................................. 119
9.16. Connection structure of ‘flash_steady_h’. ......................... 120
9.17. Connection structure of ‘flash_stdy_non-id_co’. ............... 121
9.18. Connection structure of ‘stage_steady’ and ‘total_cond_steady’. .. 121
9.19. Connection structure of column model and sub models ........ 121
9.20. Simulation results using the column model. ......................... 122

12.1. Naming policy ‘integrate’, case a. ................................. 145
12.2. Naming policy ‘integrate’, case b. ................................. 146
12.3. Naming policy ‘integrate’, case c. ................................. 147
12.4. Naming policy ‘encapsulate’, case a. ............................. 148
12.5. Naming policy ‘encapsulate’, case b. ............................. 149
12.6. Naming policy ‘encapsulate’, case c. ............................. 149
12.7. Equation System using ports. ..................................... 152
12.8. Units using ports created by two different teams. ............. 154
12.9. Connection using ports and streams. ............................ 155
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS</td>
<td>Computer algebra systems</td>
<td>6</td>
</tr>
<tr>
<td>CN#</td>
<td>Characteristic of symbolic notation</td>
<td>28</td>
</tr>
<tr>
<td>DAE</td>
<td>System of differential and algebraic equations</td>
<td>8</td>
</tr>
<tr>
<td>DIPPR</td>
<td>Design Institute for Physical Properties</td>
<td>116</td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended Backus-Naur Form</td>
<td>140</td>
</tr>
<tr>
<td>IPDAE</td>
<td>System of partial differential and algebraic equations</td>
<td>7</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-driven architecture</td>
<td>20</td>
</tr>
<tr>
<td>NLE</td>
<td>System of non-linear algebraic equations</td>
<td>8</td>
</tr>
<tr>
<td>OCM</td>
<td>Oxidative coupling of methane</td>
<td>2</td>
</tr>
<tr>
<td>OCR</td>
<td>Optical character recognition</td>
<td>10</td>
</tr>
<tr>
<td>ODE</td>
<td>System of differential equations</td>
<td>46</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform-independent model</td>
<td>20</td>
</tr>
<tr>
<td>PSE</td>
<td>Process systems engineering</td>
<td>2</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform-specific model</td>
<td>20</td>
</tr>
<tr>
<td>RN#</td>
<td>Requirement for symbolic notations</td>
<td>30</td>
</tr>
<tr>
<td>RS#</td>
<td>Requirement for storage formats</td>
<td>33</td>
</tr>
<tr>
<td>UML</td>
<td>Unified modeling language</td>
<td>21</td>
</tr>
</tbody>
</table>
Part I.

Concepts
1. Introduction

1.1. Motivation

The cluster of excellence UniCat [Pre07] is concerned with the topic of oxidative coupling of methane (OCM) with the aim of developing a process that is feasible on an industrial level. In order to reach the zone of feasibility, researchers from different disciplines including chemistry, biology, and process systems engineering cover various aspects of the problem and cooperate in a multi-national group that is spread over several universities and research centers. In the research department of Dynamic and Operation of Technical Plants of the TU-Berlin, the focus was put on the design of a suitable reactor (cf. [JGAG+10]) and the design and evaluation of efficient process alternatives (cf. [SAGW11]).

The interdisciplinary and inter-cultural cooperation within UniCat represents a modern form of teamwork in a globalized world and therefore, the means of cooperation itself became part of the research. Since the development and exchange of mathematical models is an important part of the cooperation, a common modeling platform was desired that overcomes the boarders between individual specialization and numerical tools. Based on the experience of an existing project ([ZGUW04, Zer08]), the new modeling platform should be easily accessible and the models should be self explanatory, well documented and usable in different applications.

1.2. Outline

Part I of this document is concerned with the identification of research aims and the specification of concepts in response. Part II covers the application of the introduced concepts in a new modular modeling environment named MOSAIC. Part III contains examples that demonstrate the practical use of MOSAIC with respect to the introduced concepts. Conclusions are given in Part IV.

In the remainder of this chapter, the type of models considered in this work is specified, and various aspects of model creation and application are discussed. In this context, existing approaches for the improvement of the modeling process are described and the respective advantages and drawbacks are pointed out. The concrete research aims are identified in Chapter 2 and the resulting concepts are presented in Chapters 3, 4, and 5.

1.3. Use of models in process systems engineering

In process systems engineering (PSE), computer models play an important role. In the industry, simulation studies are performed to support market decisions, the design
of equipment, as well as safety and feasibility analysis. Computer models are used to formulate optimization problems in order to pursue economical or ecological goals under constraints of product demands, and safety and environmental regulations. Furthermore, the operation can be based on computer models for specific units through model based control techniques. The application of models in PSE has been described by many authors, e.g. [BD06, KM09].

Along with the different needs in PSE, different approaches to describing related problems have been developed. In most cases, modeling involves the formulation of mathematical relations that serve as mechanistic or empirical representations of the reality. Systems of equations are a common way of defining models of unit operations and processes. Other approaches are computational fluid dynamics ([BMP04]), which focuses on the detailed internal behavior, and black-box models such as neural networks (e.g. [TK94]), which are useful when the problem’s structure is unknown or likely to change. The calculation of physical properties is an important field that can be considered as a service in all process systems engineering models [GO01]. Besides the sound mechanistic insight and mathematical representation of the reality, the expertise in this field includes the algorithmic knowledge to obtain physically meaningful solutions. In general, mechanistic correlations are used that are supplied with parameters obtained through ongoing experiments or through molecular modeling calculations ([Pan00]).

The definition of models through systems of equations plays a major role and demand for the creation of custom models, which are generally of this type, has been observed by several authors, c.f. [FLM98, CI08], and therefore, this work focuses on such models.

In PSE, important and frequently used elements of equation systems are the conservation laws of mass, energy, and impulse, thermodynamic equilibrium correlations, equations of state and the approaches for transport phenomena. The application of equation systems varies from simple flash calculations up to models of complex apparatuses and entire processes.

1.4. Representation of equation-based models in PSE

In the beginning of the use of computer simulations in PSE, equations and solution algorithms were interlinked: Due to the lack of general numerical libraries, the pertinent solution algorithm was implemented together with each individual model. This was likewise done when using general methods like the Newton-Raphson root finding algorithm and when solving short-cut models that often come with a corresponding algorithm, cf. [HS81]. With the development of powerful numerical libraries, the efficiency of the modeling process was improved.

In the ongoing development of PSE modeling tools and languages, two aspects can be identified: First, the efficient representation and solution of pure mathematical models, second, the assistance in the creation, documentation, and maintenance of large models. For the first aspect, general programming languages, such as Fortran, are applied and declarative programming languages for PSE problems, such as gPROMS, have emerged. For the second aspect, different approaches have been proposed. The field can be subdivided into flowsheet modeling environments, specialized programming languages, and expert systems [Mar96, CI08].
1.4. Representation of equation-based models in PSE

1.4.1. Flowsheet environments

Flowsheet environments focus on the design of the process based on building blocks that are predefined in programming languages. In such tools, process alternatives can be created quickly. Flowsheet environments usually contain

- predefined units that can be connected via streams,
- routines and data bases for the calculation of thermodynamic behavior and physical properties,
- numerical methods including approaches for the effective treatment of flowsheet-type problems.

Examples for such software are AspenPlus [Asp12], ProSimPlus [Pro12], and ChemCad [Che07]. The extension and adaptation of the flowsheeting tools is often made possible with the help of user-defined models of units and thermodynamics, which are generally implemented in a programming language. Furthermore, interfaces are provided that permit the evocation of external program fragments for the representation of process units and thermodynamic properties, which either use propriety language constructs or standardized techniques such as CAPE-OPEN.

1.4.2. Expert systems

Expert systems offer support for the design of models. Such support can be offered for language-based models and for process design on the flowsheeting level and the field is diverse as far as the applied concepts and the provided functions are concerned [CI08]; the proposed tools differ in what kind of knowledge is considered, how this knowledge is represented, and how this knowledge is used to improve the execution of the modeling task or the quality of the resulting models.

One important goal was the enhancement of the efficiency of process flowsheet synthesis. Different techniques were used, such as collecting design alternatives and tracking design decisions in a suitable environment [BA95], or the implementation of heuristics and shortcut methods [KLD88], and many others. At the same time, the representation and intelligent use of PSE concepts in general became a central goal. A language proposed for that purpose is MODEL.LA [SHL90]. Due to the common goal of capturing PSE concepts, process-based knowledge environments contain a data model of PSE problems (at least implicitly). Since the data model applied plays a major role in knowledge based techniques, pure data models have been proposed that are reusable and extensible [BM03]. One example of such data models is VeDa [Mar96], which was successfully applied to DIVA [KHMG90] and ModKit [BLM01]. The research corresponding to data models led to the development and application of ontologies [BAY02, MWM09]. Furthermore, a formal representation and consideration of model assumptions was proposed [HC01]. Major techniques in process-based knowledge environments are:

- Providing tool selection and integration with the help of frameworks [KBG+05, MC07]. In general, the conceptual layer is separated from the actual process models and the necessary tool integration is performed in different ways, [CFS+96, BPBS00, BSM00],
• Generation of model equations and subsequent generation of program code, [DSS+97, WL08, BLM01],

• Improving the documentation in order to represent design decisions, assumptions, and the complete model development during its life cycle, eg. [DSS+97, BLM01].

The advanced approaches described above have many aspects that different workgroups have focused on individually. This kind of modularization has proved to be beneficial: COGents, a web based modeling framework, was proposed [YBF+08], which makes use of the reusable ontology [MWM09], the PSE software interface standard CAPE-OPEN [BPBS00] and web service technology in order to permit the intelligent combined use of existing PSE software published in the world wide web.

1.4.3. Programming languages

Programming languages offer the most flexible way to implement models. Their focus is the detailed representation of physical or phenomenological behavior. Programming languages can be used for the implementation of models for thermodynamic correlations and the calculation of physical properties, as well as the modeling of apparatuses and the corresponding phenomena on different levels of detail. The explicit representation of equation systems is still widely practiced. Depending on the intended purpose, different languages and language based tools are applied. The existing languages can be classified in different ways (cf. [FS03, Hen07]), some of which are presented in the following:

Degree of abstraction and scope Computer languages are human-machine interfaces and as such represent a trade-off between the extent of the power over the machine and the efficiency of problem formulation. The more general a programming language is, the more power is gained over the machine; the more problem-specific a language is, the higher its usability for a certain purpose. The relevant general programming languages are imperative, i.e. a concrete sequence of commands is given in order to solve the problem step-by-step. When using such a language, the engineer must reformulate the problem (often considerably) and provide solving algorithms or apply solving routings from existing numerical libraries. In contrast, most of the relevant problem-specific languages are declarative, i.e. the information is described without exact specifications for how the problem is to be solved. When using a problem-specific language, the necessary concepts can be represented directly and therefore, the reformulation undertaken by the engineer is limited. A bottleneck is encountered, however, when the modeling tasks includes concepts that are not considered by the language used. Two major subclasses of problem-specific languages relevant to PSE are equation-based approaches such as gPROMS [OP96] and phenomenological approaches like MODEL.LA [SHL90]. The paradigm of object orientation is used in both general programming languages (e.g. Java, C++) and problem-specific languages (e.g. Modelica [FE98]).

Imperative programming languages such as Fortran, C/C++, and Visual Basic are used in the development of model packages and flowsheet environments. Matlab [Mat12] is an imperative language that is specialized for the efficient formulation of mathematical problems. In Matlab, matrices can be handled efficiently and extensive libraries
for various engineering disciplines are provided. Python [Pyt12] is an object-oriented language with an intuitive syntax and high extensibility.

**Programming languages for general engineering** allow the implementation of models on a higher level in comparison to general programming languages. Such languages differ in their scope. POLYMATH [SCB85] is an equation-oriented environment for educational purposes that aims to facilitate the creation of comparatively small models. It has an intuitive syntax and many useful features that facilitate the analysis of instructive problems. Modelica is ‘intended to become a *de facto* standard’ modeling language [FE98] that is applicable in different domains of engineering, permits hierarchical structuring, and combines several modern concepts such as object orientation, equation-based and model-driven specification.

**Programming languages dedicated to PSE problems** have language elements that represent concepts of PSE. Thereby, languages such as SPEEDUP [Pan88], ASCEND [PEWW91], and gPROMS [OP96], make it possible to represent the topology of process models and classify the variables into physical quantities. Corresponding numerical environments perform consistency checks, problem simplification and Jacobian matrix generation. Additional packages such as MULTIFLASH [INF12] perform problem specific tasks such as physical properties calculation and initialization of physical models.

The language MODEL.LA [SHL90] is strictly focused on the description of the concepts which are only translated into mathematical correlations at compile time.

**Environments using a symbolic mathematic notation** have a long tradition. An overview of the first steps of symbolic languages is given in [Sam72]. Two main aims can be distinguished: First, the application of natural language and ‘natural’ mathematic symbolic notation as input for computers. Second, the automation of symbolic methods to solve mathematical problems, which resulted in the creation of computer algebra systems (CAS). Examples for the first group are the Automated Programmer [KKG92] and the tool MathCad [Mat97]. Prominent examples of the second group are Maple [AB94] and Mathematica [Soi95]. The environment MapleSim [Map12] permits the creation and reuse of hierarchical models and makes use of the computer algebra functionalities of Maple.

**Steps of abstraction** The development of programming languages can be subdivided into five generations, [Hen07]. The first generation represents a direct control of the computer, with the help of binary processor-specific codes (machine language). The second generation consists of various assembly languages which permit the use of textual commands instead of binary expressions. The third generation is represented by higher programming languages such as Fortran, C++ and Java, which offer an abstraction of the used memory, convenient program flow control, modularization, and object orientation. The fourth generation comprises script languages, such as Python, that permit the integration of different programs, and descriptive languages, such as XML, that permit the representation of data models and data structures. The fifth generation is represented
by declarative problem-specific languages, which do not describe the solution of the problem explicitly.

Every evolutionary step represents an abstraction with respect to the previous generation: The assembly languages were created to generate machine code; Fortran was created to generate assembly code; Matlab was developed as a wrapper for Fortran to facilitate the description of matrix-related problems. Modelica was created to describe the models independently from solution techniques and from the implementation code. A further abstraction was undertaken with the help of software tools, such as MathCad, which use a symbolic mathematic notation and can therefore be considered to be a layer above programming languages. In conclusion, starting from the use of computers in propriety languages, the gap between the computer representation of mathematical models and the documentation-level representation was decreased steadily.

As far as the use of symbolic notions is concerned, however, a generalization is missing: The employed notations differ between the software tools and between articles in the literature; furthermore, the proposed guidelines (such as [DIN97, Int09]) are not sufficient for an unambiguous representation.

1.4.4. Classification of the work presented here

The work presented here is concerned with the creation and documentation of programmed models based on equations and equation systems. Such programmed models can be used in a wide variety of contexts. They can be used independently in order to obtain results for a concrete engineering task, they can be implemented as customized models in modeling environments in order to reflect the specific behavior of novel equipment, and they can be integrated in software projects to create dedicated tools, such as flowsheeting software. Furthermore, this work discusses the specification of mathematical models on the level of documentation. This aspect improves the readability of the models and avoids deficiencies between documentation and implementation. As far as expert systems are concerned, technique of documentation-level modeling can be applied in all situation where mathematical specifications are used.

1.5. Considered problems and numerical software

Types of equation systems in PSE. If all existing approaches for modeling phenomena in equation systems are considered, the models are mixed systems of integral, partial differential and algebraic equations (IPDAE) [OP96]. equations are rarely used. Even the dynamic simulation is used comparatively seldom, which is due to the higher modeling effort [BD05]. Simple models are preferable with respect to prior knowledge, modeling effort and simulation stability [FLM98]. Thus, the differential algebraic equation systems are from a practical point of view of more relevance than the above mentioned integral, partial differential algebraic equation systems.

Scope of this work. Although integral terms and partial differential equations are important for the representation of certain phenomena, the most relevant applications in industry are covered by non-linear algebraic or differential algebraic equation systems
1.6. Documentation

(NLE/DAE) [BD06]. This work is focused on modeling that is based on the latter type of equation systems.

The properties of a plant are subject to discontinuities and many processes include complex discrete manipulations [PB93]. The representation of such aspects does not lie within the scope of this work.

**Existing numerical software.** There are many numerical algorithms for solving the problems relevant in this work. The basic principles for the solution of NLE and DAE systems, as well as important techniques for pretreatment and reformulation, are described by many authors, e.g. [Pan88, DSE92, Ram89], and [CH01]. Different numerical libraries have been developed that address a large variety of engineering related numerical techniques including the solution of DAE and NLE systems. Examples for commercial libraries are NAG [The12] and IMSL [Rog12], important non-commercial libraries are BzzMath [MDBFP09, BF10] and GSL [Gal09]. Computer algebra systems are also applied in PSE modeling tasks, most importantly where the techniques of symbolic computation, efficient reformulation and derivative generation are necessary. An overview of different computer algebra systems is given in [AHKA08]. The modeling environment MapleSim is a commercial example of this trend, the combined use of computer algebra and numerical algorithms is discussed in [ABLL09, All07].

The work presented here focuses on the generation of program code for the use of existing numerical libraries and pertinent PSE software tools. The implementation or improvement of numerical algorithms does not lie within the scope of this work.

1.6. Documentation

The documentation of computer models is an important aspect. It should contain information about assumptions, design decisions but also about decisions for changes [FLM98]. The two primary documentation techniques are the use of in-line comments in the programmed model and the provision of significant information in individual files. Both of these pieces of information must be revised and updated whenever the model changes during its life cycle. For the use of in-line comments, documentation generators exist that analyze the program code and create interactive documentation, usually using hypertext, that represents the structure of the code up to a certain level. Textual descriptions in the program code can be made part of the generated documentation. Examples of this approach are JavaDoc (Java), Doxygen [vH10] and ROBODoc [SKvWB10] (different languages), M2HTML [Fla10] (Matlab), and the Matlab Report Generator [Mat10]. For the management of documentation and specification in separate files, intelligent storage and update mechanisms have been proposed, e.g. [BHL01].

Obtaining the correct documentation of the current state of a model is an important feature. Therefore, many expert systems and modeling environments offer a documentation output of the applied assumptions and model equations. Examples are DIVA [Rau98], ModKit [BLM01], and ECOBAS [BHL01].

The work presented here discusses the generation of a model specification on the documentation level. That means that a detailed documentation of the model is created in formats such as LaTeX or MathML and the contained mathematical expressions and
definitions are used as specification for the numerical calculation of the model. Therefore, the documentation cannot be wrong or outdated in this approach, since it is identical with the model specification.

1.7. Model specification vs. documentation

A technique for avoiding redundancies and outdated documentation is to maintain the model description and the implementation in the same physical document. As described above, the in-line documentation of program code is one example of this technique. Apart from that, this technique is present in mathematical tools, such as MapleSim, MathCad, ScientificWorkplace, and Mathematica, which work with a combination of linear commands and expressions in mathematic symbolic notation. In comparison to in-line documentation of program code, such approaches minimize the visual and conceptual offset between the specification and the presentation in the literature. The above approaches use proprietary data formats to store the models and generally offer the generation of program code and documentation in common formats (e.g. LaTeX or HTML); consequently, the files and the storage formats of model specification and model documentation are different.

A more consequent application of this approach is the specification of the model directly in a documentation-level file, e.g. a LaTeX or Microsoft Word document. Corresponding projects are presented by [Koh08] and [ABLL10].

The work presented here tries to go farther than the above approaches. The focus is not on the use of one concrete documentation-level language or numerical interface for the specification of models. The main argument is that if an unambiguous visual symbolic notation of mathematical content is used, then the translation of the described models into program code is independent from the used documentation-level format, and consequently, different suitable formats exist and can be used simultaneously. The discussion includes the observation that many widely used symbolic notation conventions lead to ambiguities, which could be avoided by simple measures. An example for an unambiguous and yet powerful symbolic notation is presented and tested with different creation and translation tools.

1.7.1. Model content vs. presentation

The representation of mathematical expressions is generally divided into content and presentation related approaches. The mathematical expressions of model descriptions in the literature are displayed in symbolic mathematical notation and the used data formats are focused on the presentation rather than the content. The representations of a mathematical model including annotations that contain descriptions can be considered as artifacts which are the primary means of scientific interaction; furthermore it was observed that each research domain develops a characteristic form of mathematical symbolic notation (cf. [MK08]). The presented model description should be unambiguous and complete at least to a human reader. However, such documentation-level model descriptions are not used to represent the content of the mathematical expressions, e.g. as an input for numerical software. One reason for this situation is that there is no
applied standardization for symbolic mathematic expressions: The proposed rules, such as [Int09], are considered to be loose recommendations that are deliberately open to personal adaption. Furthermore, on closer observation, it can be found that such norms are aimed at uniformity of presentation but not capable of achieving unambiguity with respect to the mathematical content. Accordingly the unambiguity of mathematic expressions on the level of documentation is subject to unwritten rules that vary between research domains.

**Recognition of symbolic notation.** The evaluation of mathematical notation is a wide and active research field. Starting from scanned documents, the first task to be solved is optical character recognition (OCR). Introductions to the topic and its basic principles are given in [BG97, CY00]. In this context, the parsing of two-dimensional notations is important. Parsers work on the basis of an expected structure of the input. The definition of such structures is given formally, e.g. in grammars. The different proposed algorithms and grammars for the parsing of symbolic mathematic notations are reviewed in [CY00] ¹. Notably, OCR approaches are not only directed to the recognition of the symbols and their spatial arrangement but often additionally attempt to resolve the meaning of the mathematical expression. This step of content determination, however, suffers from the weaknesses of mathematical notations described above (i.e. ambiguity, variety of style, field of application). Moreover, in such approaches many written and unwritten assumptions about the input notations are made which do not hold for expressions used in process systems engineering, as will be described later.

**Content and presentation in practice.** There is a need for unambiguous representation of mathematical content, not only for exchange between researchers but also between software tools. In order to guarantee unambiguous representations, several storage formats have been developed that focus on the mathematical content, e.g. Content MathML [ABC+10] and OpenMath [BCC+04]. Thereby, the plain content can be evaluated and manipulated in software tools, whereas the presentation of the content in different documentation-level formats is then a separate task, c.f. [KB06, SW06]. An advantage of this approach is that the content remains intact even if the resulting documentation-level representation is ambiguous.

In the work presented here the focus lies on the use of an unambiguous symbolic notation. The drawback of the above approaches is that the correctly specified mathematical content can be presented visually in an ambiguous way, which leads to differences between specification and documentation. If, however, the symbolic notation used for the visual presentation is unambiguous, then it can be used for the representation of the mathematical content and pertinent symbolic markup formats such as LaTeX or Presentation MathML can be used for the storage. The potential of the resulting technique is discussed in this work.

¹For grammars of string-based programming languages refer to [Mar97]
1.7.2. General assumptions and PSE model representation

Symbolic mathematic notations are the main subject of several tools and approaches described above: Computer algebra systems, tools for optical character recognition, and approaches for symbolic representation of mathematical content. Such tools and their related publications apparently share a common unwritten assumption: A mathematical identifier (i.e. variable or parameter name) has one character. Accordingly, examples such as

\[ a + \frac{b}{c} \quad \text{or} \quad x^2 = 2x - 1 \]

(taken from [CY00]) are common and the discussion of unambiguously displaying and recognizing mathematical content focuses on different ways of representing mathematical operations, e.g. by using different character sets for operators. In contrast, in the domain of PSE, the identifiers of model equations often contain more symbols, e.g.

\[ HU^L_j \]

which is usually further explained by a terminological notation (or ‘nomenclature’), which helps to interpret the variable name above as the ‘holdup’ of the ‘liquid’ phase of stage \( j \). Other purely explanatory notations are possible, e.g. \( \hat{a} \). It should be noted that such combined use of symbols is often used to make the expressions easier to understand. More importantly, the consideration of such modular variable names has a strong effect on the ambiguity of symbolic notations. Therefore, the discussion of symbolic notations should encompass such practices. In some cases, especially in computer algebra systems, variable names with several characters and subscripts are considered. However, the situation of more complex variable names including superscripts is never considered. Furthermore, the terminological notation is not considered in such tools and therefore modelers use long variable names to render the expressions understandable. As a result, the visual offset between published and implemented mathematical expressions remains quite large.

The work presented here discusses the description and the use of complex symbolic notations. While the scope of covered mathematical operations and problems is intentionally kept small, the focus is put on the unambiguous use of composed variable names in combination with mathematical operators. Furthermore the use of terminological notations, as it is common practice for models published in the literature, is considered as a vital part of the model documentation.
2. Overview of the covered concepts

2.1. Identification of research aims

2.1.1. The main goals and proposed concepts

The MOSAIC modeling environment is a tool to implement customized models. There are four basic aims that are addressed in this work:

1. the encouragement and support of cooperative work,
2. the improvement of documentation during the modeling process,
3. the minimization of programming errors,
4. the minimization of programming effort.

In the following, several concepts and the corresponding ways of addressing the goals will be discussed.

Documentation-level modeling  The first concept that will be investigated in this work is the definition of models directly on the level of documentation. The application of such a concept supports goals 1, 2 and 3: If the mathematical symbolic notation can be used for the formal description of equation systems, then programming can be completely avoided. Thereby, a risk of error through manually transferring the mathematical problem description into a modeling language is avoided as well. In addition, the model definition on the documentation level can serve as model documentation, which would avoid differences between model definition and documentation. Therefore, such an approach would improve documentation during modeling. Finally, such an approach facilitates cooperative work, since the universal mathematical language is used instead of specific modeling languages. It should be noted that although it is reasonable for an engineer to be familiar with several modeling languages, expecting all cooperating engineers to be equally familiar with the same modeling language is less reasonable, especially when ad-hoc cooperation between different universities or industrial work groups is considered.

Modularity  The reuse of programmed models or model parts is a common strategy. Reuse avoids reimplementation and thereby reduces both the effort of programming and the overall amount of implementation errors. Since reuse leads to a reduction in the amount of model implementations, and thereby to a concentration of the resources for correction, improvement, and maintenance, the quality of the models can be increased.
2.1. Identification of research aims

Modularity is a key property for reusability. In programming, procedural abstraction and object orientation are used to divide computational models into small reusable modules. Generally, modularity leads to small model elements with a low complexity and therefore reduces the potential of implementation errors. If the created modules are fashioned in a way that they can be reused by others, then modularity is also a cornerstone of cooperative work. Reuse through modularity is not a paradigm that is restricted to programming: It can be applied to any kind of model description. Furthermore, it can be shown that the procedure of defining models can be subdivided and the corresponding information can be specified in modules, which leads to an additional degree of flexibility. In summary, it can be said that the concept of modularization addresses the goals 1, 3, and 4.

Model-driven development and code generation

The implementation of a model is the formulation of all desired aspects of the model in a machine-readable form. Usually programming languages are used for that purpose, which has several drawbacks. First, models implemented in programming languages cannot be exchanged arbitrarily, unless the cooperating engineers use the same language. Second, as programming languages evolve, implemented models carry the risk of becoming obsolete. During the live cycle of a model, its mathematical content may be constantly adapted to changes in demands and the state of knowledge. For the sake of persistence, the knowledge accumulated in this way should be stored in an implementation-independent manner. The storage format or language used for this purpose should be able to represent all necessary model information without considering implementation-related aspects. The model documentation usually has the function of storing such model information including assumptions, design decisions, and changes during the life cycle. However, due to its separated and supplementary character, the documentation is often incomplete or not present at all.

The field of model-driven development addresses the separation of model description and model implementation. The model description is stored in a platform-independent model that can be translated by software tools into the necessary fragments of program code, i.e. the platform-specific model. Software-driven code generation also minimizes programming errors and effort and, since it also comprises the reuse of existing platform-independent models in different programming languages, it supports cooperative work. Thereby, goals 1, 3, and 4 are addressed.

The question of the choice of the implementation-independent language, the meta language used, is not trivial, since such a language and the corresponding translation tools also evolve, which results in the same persistence problems that were stated above for the use of programming languages. This problem can be solved by using mathematical symbolic notation for the platform-independent model description and storing the corresponding expressions in documentation-level formats. If the persistence of the mathematical expressions can be ensured, the ongoing development of translating software can focus on adaption to new documentation-level formats. Apart from that, defining models on the level of documentation allows to join model definition and documentation, which addresses goal number 2.

The use of the Internet

An enhancement of the positive effects of modularity can be achieved by improving the accessibility of reusable model parts, which can be effected by
establishing a central storage of the data. The access to such databases can be effected via network technologies. Among other things, such an approach solves the problem of cooperating from locally separated workplaces in a convenient way, since it avoids the distribution of actual copies of the model descriptions and thereby the diminishes risk of inconsistencies.

This point addresses goal number 1, but enhances the reuse effects stated in the modularity paragraph and therefore, goals 3 and 4 are also addressed.

### 2.1.2. Resulting fields of interest

Summarizing, the four general aims stated in the beginning of this section lead to four main areas of interest:

1. to work on the documentation level using a two-dimensional symbolic notation for mathematical expressions and additional fields for meta information.

2. to implement a modeling systematic that is highly modular through dividing the models into small reusable building blocks.

3. to extensively use automated code generation based on platform-independent models described in XML/MathML.

4. to implement the software as an Internet application so that the software runs in a web browser and the data is centrally stored on a modeling server.

### 2.1.3. Order of elaboration

Sections 3 and 4 present a foundation of the concepts of documentation-level modeling and modularity. The application of these concepts is described in sections 6 and 7. The concept of code generation can be described in short as a mapping from the meta-level models to program code; the application of this concept is described in section 8. The advantages of web based tools and some notes on its application within this work are presented in section 5.

### 2.2. Orientation with respect to similar projects

The stated goals and the addressed fields of interest are not new and many approaches and tools have been proposed to improve distinct aspects of modeling. In the following a selection of similar tools is discussed.

### 2.2.1. Tools that work on a meta level

Knowledge-based tools that permit the generation of the set of model equations and the transfer to program code are related to this work, since they have the same goals and make the generated models usable in many environments. Examples of this class of tools are ModKit [BLM01] and the Mobatec Modeller [WL08]. Such tools focus on the conceptual aid in generating the mathematical content of the models, whereas
the work presented here discusses the definition of models by the user on the level of
documentation.

2.2.2. Tools that work on the documentation level

A software for the use of two-dimensional symbolic notation as input for code generation
is proposed in [KKG92], which “has been designed to minimize the linguistic difference
between the solution specification, expressed in a notation and syntax ‘natural’ to the
application area, and the executable program module”. The software is able to generate
code in Fortran, C, and Ada for a large set of mathematical operations but the used
symbolic notation does not consider modular variable names such as $D_{dry}^{left}$.

The definition of models in documentation-level documents is proposed in [ABLL10],
where model equations are defined in Microsoft Word documents. Specific XML struc-
tures are employed to connect to numerical solvers; the equations are not presented in
a two-dimensional symbolic notation.

2.2.3. Tools related to computer algebra systems

Modeling tools that are related to computer algebra systems offer a presentation of the
model in documents that contain documentation and the mathematical specification in
a two-dimensional symbolic notation. Examples for this class of tools are MapleSim
and MathCad. Such tools do not consider modular variable names. Furthermore, the
specification of mathematical expressions is generally performed in the respective envi-
ronment using a specific editor or a combination of special keys, menu items and the
mouse. Thereby, the content and the presentation are entered at the same time. The
approach proposed in this work also includes the definition of mathematical expressions
in independent storage formats and thereby makes higher demands on the used symbolic
notation.

2.2.4. Orientation with respect to the preceding project

The project preceding this work, cf. [ZGUW04, Zer08], was a web-based software working
with MathML in the Content Markup. It used code generation into Java and Fortran for
numerical algorithms which were an integral part of the tool. Physical concepts could be
represented with the help of a fixed terminological notation, e.g. by using $T$ for variables
representing the temperature. The algorithmical and numerical aspects of this tool were
advanced: The structure of the equation system could be adapted and specific physical
contents could be considered. The software did not have a graphical user interface and
was used as an engine for XML files that were created by the user on the text level;
the software NetBeans was recommended as a programming aid for the XML/MathML
expressions.

At the beginning of the work for this project, the problems arising from the offset be-
tween published models and their implemented versions were discussed in our research
group. The theoretically unambiguous connection between models presented in the lit-
erature and the corresponding program code seemed to be crucial: If the documentation
(or published version) of a programmed model was correct (i.e. if the programmed model
was completely and correctly represented), then it should be possible to generate the programmed model out of the documentation. Visions of translating scans of papers and of generating model documentation for direct use in papers were part of this discussion. One conclusion of such thoughts is to move the model definition into the documentation level and to take documentation-level characteristics, such as modular variable names and nomenclatures, into account.

Based on the experience with the preceding project and in combination with the above considerations, several new directions were taken for the work presented here. First, the representation of the mathematical content was moved from the content level to the documentation level by using the Presentation Markup of MathML in combination with an expressive symbolic notation including modular variable names. Thereby, the gap between the literature and the computerized specification could be diminished – also in comparison to existing approaches with the same aim.\(^1\) Second, LaTeX was established as the input language, since the model definition in textual MathML is error-prone, much less readable than conventionally programmed expressions, and, consequently, unlikely to be accepted by the user. Third, the focus of code generation was moved away from internal numerical methods. A more general and easily extensible architecture was implemented so that program code for existing numerical libraries and specific PSE tools could be generated. Fourth, the modularity structure of the models was generalized; a move towards the literature, and thereby the documentation level, was done by representing the nomenclature as a reusable, modular model element. Finally, a graphical user interface was created to offer a user-friendly and correct creation and application of the model elements.

\(^1\)In contrast, the use of content-oriented formats, such as the previously used Content Markup of MathML, is today well covered.
3. Documentation-level modeling

3.1. Motivation

3.1.1. The need for and the challenges of documentation

In 1978, a survey about the recent developments in model documentation at that time contains the statement: “All general surveys of computer models and all assessments of specific models that have come to our attention conclude that the documents that are supposed to describe and explain these models are either nonexistent or are lacking in detail and do not serve the models well[...],” [Gas78]. Almost twenty years later, nearly the same statement can be found: “In far too many cases, model documentation leave a lot to be desired. Even worse, there are a number of models for which no documentation exists at all, so that published research results based on these models are not verifiable.” [BK97]. In the same period, the authors of a field study about modeling practices ([FLM98]) stress the general acceptance of the importance of documentation; about the behavior during the modeling process they state that: “In some cases there is virtually no documentation provided at the end of a project, whereas in other cases a significant amount of time is used for documentation.” Furthermore, the authors of this study found that “documentation often deteriorates over time because of a lack of updating.” Ten years later, in 2009, the authors of [KM09] write that “[...] we still have no adequate modeling and simulation methods and tools [...] to document, maintain and reuse models across the lifecycle of the plant in an efficient and economical way.”

The above citations demonstrate that there is a common understanding concerning the importance of model documentation in terms of maintenance and ongoing development during the life cycle, but also in terms of reuse. The problems mentioned with regard to documentation include that it is time-consuming and likely to be incomplete, missing, or outdated. Furthermore, the need for good tools and systematics to support the user in the documentation process is expressed.

In this work, the basic idea that is used to respond to these demands is to join the model definition and the model description on the level of documentation. As opposed to existing approaches, such as JavaDoc or Doxygen, in which the documentation is extracted from implementation-level files, the approach proposed in this work is to use documentation-level files to store the complete information and to extract the model definition and translate it into the desired programming language. Thereby, the documentation exists before the implementation and a snapshot of all documentation information can be generated at several stages of development, which supports a solution to the above-mentioned life-cycle documentation problem.
3.1. Motivation

The task of creating models includes many steps and considerations. Many workflows and systematics for the creation of models exist (e.g. [FLM98]), which differ according to the aspect considered. The general workflow presented in Figure 3.1 represents the creation of computer models based on mathematical and structural content that is present on the level of documentation. The level of documentation is at first used in a broad sense, so that it includes the assembling of models on paper based on personal deductions, as well as on mathematical equations and correlations found in the literature. The necessary mathematical equations and correlations are assembled in a single document, which is then programmed in the language of the desired application environment and compiled in order to obtain an executable program. Notably, the documentation takes place before the implementation in this approach, which implies that the engineer describes the model on a conceptual level first and thereby separates the PSE modeling from the implementation. If the equations have been laid down correctly, which includes numerically sensible formulation, the documentation can serve as a pattern for the program code. Certainly, Figure 3.1 is incomplete, since it does not include the iterative process of model validation and adaption. The necessary extension of the workflow is shown in Figure 3.2. The executable corresponding to the model is started and results are obtained, which are used for validation. Usually the model needs to be adapted, either to better represent the reality or to include changes in the modeled plant. This leads to changes in both the documentation and the program code. According to a strict pursuit of the workflow, any changes should be laid down first in the documentation file and then transferred to the code (arrow A in Figure 3.2). However, such changes are often implemented and documented asynchronously (arrow B in Figure 3.2), which easily leads to inconsistent or outdated documentation. The situation could be improved by

**Figure 3.1.:** Model creation on the basis of equations given in a documentation file.

**Figure 3.2.:** Manual application of changes.

### 3.1.2. A visualization of the modeling process

The task of creating models includes many steps and considerations. Many workflows and systematics for the creation of models exist (e.g. [FLM98]), which differ according to the aspect considered. The general workflow presented in Figure 3.1 represents the creation of computer models based on mathematical and structural content that is present on the level of documentation. The level of documentation is at first used in a broad sense, so that it includes the assembling of models on paper based on personal deductions, as well as on mathematical equations and correlations found in the literature. The necessary mathematical equations and correlations are assembled in a single document, which is then programmed in the language of the desired application environment and compiled in order to obtain an executable program. Notably, the documentation takes place before the implementation in this approach, which implies that the engineer describes the model on a conceptual level first and thereby separates the PSE modeling from the implementation. If the equations have been laid down correctly, which includes numerically sensible formulation, the documentation can serve as a pattern for the program code. Certainly, Figure 3.1 is incomplete, since it does not include the iterative process of model validation and adaption. The necessary extension of the workflow is shown in Figure 3.2. The executable corresponding to the model is started and results are obtained, which are used for validation. Usually the model needs to be adapted, either to better represent the reality or to include changes in the modeled plant. This leads to changes in both the documentation and the program code. According to a strict pursuit of the workflow, any changes should be laid down first in the documentation file and then transferred to the code (arrow A in Figure 3.2). However, such changes are often implemented and documented asynchronously (arrow B in Figure 3.2), which easily leads to inconsistent or outdated documentation. The situation could be improved by
avoiding independent changes in documentation and implementation. Moreover, if the documentation file is formal enough with regard to the contained mathematical expressions and structural information, then the implementation process can be automated, as shown in Figure 3.3. However, the simplicity of the basic approach displayed in this figure hides considerable challenges when it comes to a concrete application. A review of models laid down in the literature and in technical documentation reveals that there are many different styles, conventions, unwritten rules, etc., cf. e.g. [Mar71, CY00, SW06].

The questions of defining a general and suitable form for describing models in documentation files and for determining whether such descriptions are ‘formal enough’ can not be answered in a general way, apparently. A further question is: In which cases can we do without programming and when are the structural modeling capacities of higher programming languages indispensable?

Despite the above-mentioned challenges, the approach motivated in Figure 3.3 has great potential, especially if a separation between the model description file and the modeling software can be achieved. In this case, the model description can sustain the ongoing development of numerical tools and the task of continuous support is reduced to the translation of the model description. Thereby, the model is kept available for machines and modelers throughout its life cycle. Furthermore, the above-mentioned separation of description and implementation can serve as a background for techniques adapted from the field of model-driven software engineering, which leads to a considerable gain in productivity. In order to make use of such advantages, some of the above questions will be answered, and even if a complete solution to the problem of documentation-level model definitions cannot be given within the scope of this work, a significant step will be taken. To this end, a systematics for the definition of documentation-level mathematical notations will be proposed. To demonstrate the functioning of the systematics, it will be applied to create a symbolic notation for use in a new modeling environment.

3.2. Principle aim and used techniques

The several disadvantages of storing implemented models and the corresponding documentation in separate files were described above in section 3.1. Accordingly, this chapter is dedicated to the question of whether or not the definition of mathematical models on
the documentation level is at all possible and if so, whether or not such a way of defining models also has advantages.

In order to derive relevant characteristics, the notion of the term ‘documentation-level modeling’ needs to be defined more precisely: The aim of this technique is to use files in documentation-level formats that are intended to contain informal information, and to give sections of such documents a formal character so that they can be used as unambiguous statements that define a numerical model, which in turn can be used for code generation. The simplest way to do so would be to define the models in a programming language, e.g. Modelica, and then place the model description within a document in a documentation-level format, e.g. Microsoft’s DOC binary format. Thereby, the file containing the mathematical model description can be easily supplied with formatted and structured descriptions including images, rendered tables etc. A specialized parser can be used to filter out the formal model specification and precompile this information into a Modelica model, which can be solved with an applicable software.

3.2.1. Considered characteristics of a documentation-level description

The argument in this section attempts to expand upon the above example: In addition to storage in a documentation-level format, the formal model specification must share also characteristics with the usual documentation-level content and presentation. This work focuses on three possible characteristics:

1. The mathematical content should be presented in two-dimensional symbolic notation, i.e. \( a^b = b \cdot \frac{c}{d} \) should be used instead of \( aL := b \cdot (c/d) \).

2. A nomenclature, or ‘terminological notation’, should be present that describes all symbols used for the variable names, i.e. the meaning of \( a, b, c, d \) and \( L \) must be stated for the expression above.

3. Textual descriptions of assumptions, design decisions etc. should be required for each equation, set of equations, problem specification, etc.

3.2.2. Documentation-level modeling as a form of model-driven software engineering

In the following, documents are considered that are stored in a given binary format or markup language and contain the description of a mathematical model. For a class of numerical problems, it will be shown that the plain mathematical description within such documents is sufficient to specify a numerical problem. The fact that such documentation-level descriptions contain the mathematical model without implementation-related information can be beneficial. Documentation-level models share this characteristic with classical platform-independent models (PIMs) in model-driven architecture (MDA), cf. [MM03]. Platform-independent models are used as a basis for the automated code generation of platform-specific models (PSMs), which, in the best-case scenario, is the complete program code, cf. Figure 3.4. This approach has many advantages. First, the platform-independent models can be translated into the program
code of different programming languages that is shaped to meet platform specific needs. Thereby, the models can be used in different applications. Second, repeated implementation of the models in different platforms or programming languages is avoided. Third, instead of debugging and correcting individual program code representations (PSMs) of the same model, the platform-independent models can be corrected, which leads to an improvement of all platform-specific models. Fourth, the usability of the corrected and validated PIMs is independent from the technological development of programming languages and applications.

MDA is proposed and continually developed by the Object Management Group (OMG) [Obj97a] and widely accepted by the industry. The approach uses the unified modeling language (UML), which is a graphical modeling language. The UML standard is specified and published by the OMG [Obj97b]. For the exchange of UML models, the XML derivate XMI has been developed. It is important to note that the platform-specific models within the established model-driven software engineering approach use therefore a fixed notation and data format. In contrast, the documentation-level models considered in this work cannot be assumed to have a fixed notation, since no standardization can be expected soon. Furthermore, the documentation-level modeling approach should be open to various existing storage formats, which leads to an increase in complexity. The feasibility of this aspect, as well as the restrictions resulting from it, will be examined here.

3.3. Platform-independent models in the documentation-level

In order to serve as platform-independent models, the mathematical problems stored on the documentation level must be processable by a software and therefore must adhere to formal specifications. In practice, however, several different ways of defining a given mathematical problem are used and many data formats exist for the documentation level. This situation will now be considered in more detail.

Figure 3.4.: Translation of platform-independent models to platform-specific models (source: [RFW+04]).
3.3.1. Involved models

Whenever an engineer produces a representation of reality, he creates a model, which is the assembly of all assumptions, considered facts, applied sub models (such as mathematical correlations) etc. Such a model, which will be called ‘abstract model’ in the following, is not bound to any document or implementation: It may exist in the mind of the engineer and be communicated orally, through written model descriptions or in the form of implementations. The full set of formal statements and informal descriptions used to document and exchange the model in a non-implemented form will be referred to as ‘model documentation’, whereas any program code representing the model will be called ‘model implementation’. Since several data formats and different valid notations exist on the documentation level, different model documentations may exist for one abstract model. Similarly, model implementations in different programming languages are possible, so many model implementations may correspond to the same abstract model, cf. Figure 3.5 left. The foremost interest of this work lies in the connection between the documentation of a model, more exactly one issue of the model documentation, and the various model implementations, cf. Figure 3.5 right. The therefore necessary consideration of a potential formality of the documentation of models leads to an analysis of the connection between different model documentations of the same abstract model.

To provide a more detailed view, the three types of models will be described in the form of levels in the example of a given model identified by “Distillation Column 1023”. On the abstract level, the focus is put on the choice of equations, constraints etc. Representation in a document or program code is not yet considered, cf. Figure 3.6.

On the documentation level, the situation becomes more complex: If the model documentation is supposed to be processable by a software, e.g. a parser, the information
3.3. Platform-independent models in the documentation-level

![Diagram of Documentation Level (DL)]

Figure 3.7.: Complexity of the model representation on the documentation level.

to be obtained by such software must be presented in a formal design. For the following discussion it is useful to divide the description of such a formal design into the ‘data model’ and the ‘symbolic notation’. The term data model is widely used in the field of database engineering, but is used here in the broader notion found in [AU95], where data models are considered as ‘the abstractions used to describe problems’, and where data models are explained further by ‘In fact, if you examine any [...] familiar piece of software [...] a pattern emerges. Each program that is designed to be used by others has its own data model, within which the user must work.’ In the context of model documentation, the data model is understood as the concepts accounted for in the description of mathematical models, e.g. a ‘problem’, which consists of an ‘equation system’, which is built of ‘equations’, etc. Generally, several data models are possible that may be very different from each other, even if the scope is restricted to models described by non-linear algebraic equation systems. The differences may be found in the way the algebraic expressions and their variables are represented, or in the way several equations are assigned to an equation system. In order to create a model according to a given data model, a syntactical or graphical convention must be provided that can be used to represent the contained concepts. In the following, such conventions will be called ‘symbolic notation’. Several symbolic notations can stem from each data model. A further degree of complexity is added by the fact that the files containing model representations can have different storage formats, such as markup languages or binary formats. The complete situation is illustrated in Figure 3.7.

On the implementation level, the abstract model is represented by fragments of program code. Since additional requirements are stated for the documentation-level model definitions, cf. section 3.2.1, a given data model within the documentation level will contain more information than is necessary for the formulation of the numerical problem. Accordingly, the data models used on the implementation level may contain less information and have a different structure when compared to documentation-level data models. Based on each implementation-level data model, many model implementations can be created. Platform models are used to transform the model into program code, cf. Figure 3.8.
3.3. Platform-independent models in the documentation-level

![Diagram showing model representation on the implementation level.]

Figure 3.8.: Model representation on the implementation level.

### 3.3.2. Involved transformations

As was described in section 3.1, the desired workflow in this work is to lay down the abstract models in the documentation level and to use the model documentation to automatically generate model implementations, as shown in Figure 3.5 right. In the previous section, several models were introduced on the documentation and the implementation level. The in and output of transformations, however, are the actual documents on both levels. In such transfer steps, documentation-level documents contain the platform independent models, whereas the implementation-level models contain the platform specific models, cf. Figure 3.9. In the field of model-driven software engineering, the usual notation is the symbolic language UML, for which the standardized XML-based storage format XMI exists. In this case, all platform-independent models use the same data model, symbolic notation, and storage format\(^1\). For the models covered in this work, no suitable standard is available that also meets the requirements formulated in section 3.2.1. The proposition of such a standard is not the intention of this work. However, the potentials and hindrances of documentation-level modeling without such standards will be discussed, in order to derive a useful methodology that will serve as a foundation for the modeling concepts used in this work.

In this context, several transformations must be considered. In addition to the transfer from documentation-level models taking the role of PIMs, there are two relevant transformations in the documentation level: the transfer between different symbolic notations and between different storage formats, cf. Figure 3.7 and 3.9. Next, both of these transformations will be discussed with respect to comprehensiveness and effort.

### 3.3.3. The challenge of diversity

There are many tools that allow the use of mathematical symbolic notation and are therefore interesting for this subject (e.g. MapleSim, Mathematica, MathCad, Scientific WorkPlace); often code generation functionalities are included. In such tools the formulas are added either interactively through a graphical user interface or with the

\(^1\)Even if an alternative storage format is used, an export feature to the standard XMI is usually offered.
help of a linear input language and the content is stored in the tool-specific format\(^2\), cf. Figure 3.10. One drawback of this situation is that the models are not independent from the numerical tool. Another drawback is that linear input language is interpreted differently by the tools, cf. [SR07]. On the contrary, the situation can be improved by focusing on the visible symbolic notation and allowing the use of documentation-level formats such as LaTeX as input. The challenge of such an approach arises from the diversity of symbolic notations and documentation-level storage formats.

As far as the storage of the models is concerned a restriction to one ‘standard’ storage format is not desired; likewise, the software tools cannot be assumed to support all existing storage formats. Moreover, the use of the models should be independent from the tool, which results in a situation of diversity and apparent incompatibility, cf. Figure 3.11. One solution to the problem is the transfer of the documentation-level models into the applicable storage format before the import or use in the numerical tool, cf. Figure 3.12. Such a transfer step is done by applicable software that can be part of the numerical tool or supplied by a third party. An important question is the effort of transfer and the complexity of the corresponding software. If the transfer between storage formats must consider the semantic of the expressions, then rather complex specialized software must be developed which will result in high costs in addition to the used numerical tool.

\(^2\)It should be noted that the internal representation in this case may be based on the semantic rather than on the symbolic notation.
3.4. In-depth consideration of algebraic expressions

This section does not discuss the complete problem statement, but focuses on the symbolic notation of non-linear algebraic expressions. Algebraic expressions can be used for the representation of equations and functions, and are therefore of central importance. The representation of the content of algebraic expressions using programming languages or propriety formats, such as the MathML derivative used in MathCad, is standard. On the contrary, the use of tool-independent symbolic notations with typical documentation-level characteristics for the definition, exchange and calculation of mathematical models is not common. Yet, as described above, such model definitions would have many advantages, and therefore the situation will be considered next in more detail. There are two important aspects: First, the desired independence from documentation-level formats necessitates a specification of the characteristics of suitable formats. Second, the absence of an accepted convention for a mathematical symbolic notation necessitates the development of required characteristics of such notations. Especially with regard to the intended use of the stored algebraic expressions as platform-independent model for code generation, the goal of such considerations must be unambiguous visual and computational representation. Since it is often considered best practice to separate content and presentation of mathematic formulas, cf. [BCC+04, ABC+10]. In this work, however, such a separation is deliberately avoided. Therefore, the correctness of the presented approach must be justified. To this end, a methodology for the use of symbolic notations
3.4. In-depth consideration of algebraic expressions

in documentation-level storage formats will be proposed and justified by several proofs. For the applied techniques of proof refer to [Mar71] and [CLRS01].

The methodology presented next has the following goals:

1. Non-linear algebraic expressions must be displayed unambiguously with respect to their content.

2. Non-linear algebraic expressions must represent an unambiguous machine-readable statement that matches the displayed content.

3. The transfer of non-linear algebraic expressions between different storage formats must not impair the above demands.

3.4.1. Symbolic notations

Characteristics of symbolic notations

Existing standards. There are many conventions for the symbolic representation of algebraic expressions. Such conventions differ considerably according to the field of application and the cultural background. The conventions can be divided up into the representation of mathematic operations, variables, and numbers.

As for the mathematic operations, the German norm ‘DIN 1302’ [DIN97] contains a list of symbols and structures used for the representation of mathematic operations. According to that norm, the arithmetic operators are expressed by $+$, $-$, $\cdot$, and $/$. It is stated that the multiplication sign is usually omitted and that the fraction is a common alternative for the division sign $/$. Other conventions exist in this respect.

As for the representation of variables, the norm ‘DIN 1304’ in [DIN97] postulates rules for the symbols used for physical quantities. According to that norm, the conceptual structure of names is given by

$$\frac{G^4}{6^3}$$

(3.1)

where $G$ is defined as ‘base symbol’ (Grundzeichen) and consists of Latin and Greek upper-case and lower-case letters. In the norm, the positions 1 through 6 are labeled ‘side symbols’ (Nebenzeichen) and may consist of letters, ciphers or special characters. Furthermore, several conventions are given for the symbols actually used for the representation of quantities in certain domains, which is labeled terminological notation in this work. A terminological notation for the domain of process systems engineering is laid down in [VDI72], which however is no longer used. Establishing global rules is difficult due to the specific needs of each individual field of engineering. For example, the domain of process systems engineering covers both chemical and biological systems, each of which use different conventions for the naming of variables.

The representation of numbers is a complex subject of its own. Differences in the representation of real numbers are, e.g. the symbol used as floating point delimiter and the way of grouping numerals in order to highlight a number’s magnitude. Several additional conventions exist, such as the e-notation used in computer languages. For German publications, the representation of numbers is discussed in ‘DIN 1333’, [DIN97].
Rules for the notation of scientific content are also laid down in the norm ‘ISO 8000’, [Int09], of the International Organization for Standardization. Although the above-mentioned norms have served as guidelines within this work, it was found that they are not applied in current practice due to the variety of research domains, which each have a fast pace of development. To conclude, it must be stated that there is no generally applicable standard for the symbolic notation of mathematical expressions, and therefore, the consideration of a concrete set of definitions does not make sense. The presence of different valid notations and the emergence of new notations must be taken into account. Therefore, the following general considerations describe symbolic notations with characteristics and requirements instead of concrete symbols.

**Derived characteristics** The considerations above were used as a basis to propose a set of characteristics, which define a certain class of symbolic notations. The major point of this step is to provide a means to designate notations that use documentation-level techniques, such as displacements. The symbolic notations considered in this work are described by the following characteristics:

CN1 (Use of displacements) The displacement of symbols is part of the symbolic notation and follows the principle patterns

\[
\begin{align*}
\text{□}, & \text{ □, □, □} \\
\end{align*}
\] (3.2)

The patterns may be combined. The displacements are always used one degree at a time, i.e. the expression

\[
\text{□}[\text{first disp}][\text{second disp}] \\
\] (3.3)

is only allowed, if \text{first disp} contains at least one symbol or one operator according to CN4.

CN2 (Composed variable names) The variable names have the general structure (cf. [DIN97]).

\[
\begin{align*}
\text{□□□□□□} \\
\end{align*}
\] (3.4)

In the side symbols’ position, several independent symbols are allowed. All symbols used in a variable name will be called elements.

CN3 (Use of special symbols) The variable names may consist of Latin and Greek letters. The operators may be represented using special symbols, e.g.

\[
\text{\cdot}, \times, \div \\
\] (3.5)

The whole set of symbols supported in this way will be called principle symbols.
3.4. In-depth consideration of algebraic expressions

CN4 (Use of special structures for operators) The expressions contain operators that are a combination of special symbols and displaced lines, e.g.

\[ \Box \sqrt{\Box}, \Box \]

Such expressions will be called first group operators.

CN5 (Possible use of expressions for operations) Certain operators can be a combination of symbols and displacements, e.g.

\[ \sin(\Box), \sin^x(\Box), \log_{\Box}(\Box), (\Box) \]

Such expressions will be called second group operators.

CN6 (Operands) The operands of first-group and second-group operators are non-linear algebraic expressions that follow these characteristics.

In this work, first-group and second-group operators are called enclosing operators because they contain their operators within an expression (cf. Statement RN3, page 31). It should further be noted that brackets are considered to be second-group operators.

Definition of symbolic notations

The characters used to express the operations are part of the definition of the symbolic notation. According to the characteristics CN1 through CN6, operations and variables may consist of symbols and displacements. In the following, the ‘structure’ of variable names, numbers or operations refers to the combination of used characters and displacements. As far as non-linear algebraic expressions are concerned, a symbolic notation is supposed to be described by:

1. A specification of the supported data model:
   a) supported operations,
   b) covered set of numbers,
   c) elements of variable names.

2. A specification of the symbolic representation:
   a) the presentation of the operations,
   b) the presentation of numbers,
   c) the presentation of variable names.

The following example describes the effects of the data model and the symbolic representation.
Example  Consider the symbolic notations $S_X$ and $S_Y$. The data models of both notations support the arithmetic operations, which are performed on real numbers. The variable names of symbolic notation $S_X$ consist of one base-line element and several optional superscripts, whereas the variable names of symbolic notation $S_Y$ consist of one base-line element, one optional superscript, and one optional subscript. As far as the representation of the operations is concerned, both notations use the binary operators $+ - \times \div$, the numbers are presented in the decimal system using Arabic numerals, where '.' is the floating point delimiter and exponential representation is not allowed. In both notations the variable names consist of Latin letters. Subsequent superscripts in notation $S_X$ are separated by commas.

The notations $S_X$ and $S_Y$ have different data models, since the variable names are constructed in different ways. This fact leads to different scopes of expressions:

\begin{align*}
    a^{M,N} \times B - 2.4 \div c^M \\
    a_p^L - B_q + 4.03 \\
    a^L \times B + 3.5
\end{align*}

expression (3.8)

Expression 3.8 is only correct within notation $S_X$, since $a^{M,N}$ contains more than one superscript, which is not supported in notation $S_Y$ due to the specific data model. Expression 3.9 is only correct within notation $S_Y$, since $a_p^L$ contains a subscript, which is not supported in notation $S_X$. Expression 3.10 is correct in both symbolic notations.

Now consider the symbolic notation $S_Z$, which uses the same data model as notation $S_X$. However, a different symbolic representation is used: The symbols $+ - \ast /$ are used for the presentation of the operations and subsequent superscripts are separated by a white space. Since $S_X$ and $S_Z$ use the same data model, a transfer between the notations is possible (cf. Figure 3.7 in section 3.3.1), and the content of expression 3.8 can be displayed in notation $S_Z$:

\begin{align*}
    a^{M,N} \ast B - 2.4 / c^M
\end{align*}

expression (3.11)

Symbolic mathematical notations and unambiguity

In order to fulfill the goals defined at the beginning of section 3.4, the mathematical symbolic notations used must be able to represent non-linear algebraic expressions unambiguously. The desired characteristics for symbolic notations compose a frame of possible expressions. However, in order to be avoid ambiguity, symbolic notations must comply with a set of requirements.

Requirements for symbolic notations. The following set of requirements is proposed:

RN1 Distinguishability of numbers and variables:

a) Numbers must be distinguishable from variable names.

b) The value of numbers must be represented unambiguously.
c) Different variable names must be distinguishable from each other.

RN2 Distinguishability of operations: Each supported operation must be presented un-
ambiguously with respect to the other operations.

RN3 Distinguishability of arguments:
   a) Within each supported operation, the arguments must be distinguishable from
      each other as well as from their operators.
   b) The operands must be attributable unambiguously to the operations.

At first glance, the above requirements may seem to be too general to be helpful. In
this work, however, they are used as the foundation for a methodology in which they have
the purpose to serve as a bootstrap reference when designing symbolic notations. To this
end, it must be shown that they are sufficient criteria for the unambiguity of symbolic
notations. This will be achieved by formulating and proving the statement N1 below.
The advantage of this approach is that, when designing a symbolic notation, the proof
of unambiguity can be made by demonstrating that the requirements RN1 through RN3
are met.

Proof of unambiguous representation

Statement N1: A symbolic notation that has the characteristics CN1 through CN6 is
unambiguous, if the requirements for notations RN1 through RN3 are met.

Definitions: Let $S$ be a symbolic notation and let $E_k$ be the set of non-linear algebraic
expression that can be built when $k$ operators are defined in $S$. Let $\Omega$ be the set of
operators that are defined in $S$, and $\omega_k$ the $k$-th element of $\Omega$.

Assumptions: $S$ is based on the characteristics CN1 through CN6. The requirements
for notations RN1 through RN3 are met when $S$ is created.

Proof by complete induction. To show: Statement $P(n)$ is true for all $n \geq 0$, where
$P(n)$ is the following statement: All expressions contained in $E_n$ are unambiguous.

Basis Step: The set $E_0$ contains all expressions that can be built without operators.
Therefore, $E_0$ contains all numbers and all variable names that can be expressed by $S$
when no operators are defined in it. According to requirement 1, numbers and variable
names are discernible. Therefore, all expressions in $E_0$ are unambiguous and $P(0)$ is
ture.

Induction hypothesis: For $k \geq 0$ and for every $n$ with $0 \leq n \leq k$, all expressions
contained in $E_n$ are unambiguous.

Statement to be shown in induction step: All expressions contained in $E_{k+1}$ are
unambiguous.

Proof of induction step: $E_{k+1}$ contains all expressions of $E_k$ and all expressions
that are created by introducing operator $\omega_{k+1}$. Let $E_{k+1}^{(0)} \subseteq E_{k+1}$ be the set of expressions
that are created by using the elements of $E_k$ as operands of $\omega_{k+1}$. Since all elements of $E_k$ are unambiguous by the induction hypothesis, and since it is assumed that RN2 and RN3 are met, all elements of $E_{k+1}^{(0)}$ are unambiguous expressions. Let $E_{k+1}^{(1)} \subseteq E_{k+1}$ be the set of expressions that are created by using the elements of $E_{k+1}^{(0)}$ as operands. Since all elements of $E_{k+1}^{(0)}$ are unambiguous as shown above, and due to the assumed compliance with RN2 and RN3, all expressions contained in $E_{k+1}^{(1)}$ are unambiguous.

In order to include all possible expressions, it is necessary to consider the sets of expressions $E_{k+1}^{(m)} \subseteq E_{k+1}$, which are created correspondingly by using the elements of $E_{k+1}^{(m-1)}$ as operands, for all $m \geq 0$. The unambiguity for the cases $m = 0$ and $m = 1$ has been demonstrated above. In order to prove the unambiguity for all $m > 1$, it is necessary to show that all expressions contained in $E_{k+1}^{(p+1)}$ are unambiguous under the hypothesis $H$: for $p \geq 1$ and for every $m$ with $1 \leq m \leq p$, all expressions in $E_{k+1}^{(m)}$ are unambiguous. Since all elements of $E_{k+1}^{(p)}$ are unambiguous according to the hypothesis $H$, and since the compliance with RN2 and RN3 is assumed, all expressions contained in $E_{k+1}^{(p+1)}$ are unambiguous.

Finally, it is true that $E_{k+1} = E_k \cup \left( \bigcup_{m=0}^{\infty} E_{k+1}^m \right)$ and $E_k \cap \left( \bigcap_{m=0}^{\infty} E_{k+1}^m \right) = \emptyset$. Because $E_k$ and all $E_{k+1}^m$ respectively contain only unambiguous expressions, all expressions contained in $E_{k+1}$ are unambiguous. □

Resulting statements

In the following, statements are presented that can be derived either from the characteristics CN1 through CN6 or from the requirements RN1 through RN3.

Need for explicit representation of operators. Consider the set of notations $S_{\text{base}}$ with the characteristics CN1 through CN6 in which the pattern $\square$ is allowed for variable names and where the number of symbols in a base name is not fixed.

Statement N2: A notation $S_o \in S_{\text{base}}$ is ambiguous, if not all operations are expressed explicitly, i.e. through a symbol or a displacement.

Proof. Let $\omega$ be a given operation of $S_o$ that is not expressed explicitly and let $a$, $b$, $c$, and $d$ be valid symbols in $S_o$. Accordingly, the variable names $a$, $ab$, $bc$, $cd$, $d$ are in $S_o$. The application of $\omega$ to the variables $ab$ and $cd$ results into the expression $abcd$. However, the application of $\omega$ to the variables $a$, $bc$, and $d$ likewise results into the expression $abcd$. It is not possible to resolve the correct meaning of $abcd$ without prior knowledge about the existing variables. Therefore $S_o$ is ambiguous. □

Different structure of operators and variables Consider the set of notations $S_{\text{super}}$ with the characteristics CN1 through CN6 in which superscripts are allowed but not mandatory so that the patterns $\square$ and $\square^2$ (amongst others) are allowed for variable names.
Statement N3: A notation \( S_p \in S_{\text{super}} \) is ambiguous, if the operation of raising a variable \( a \) to the power of \( b \) is expressed by \( a^b \).

Proof. Let \( x \) and \( y \) be valid symbols in \( S_p \). Then \( x \), \( y \) and \( x^y \) are valid variable names. The raising of \( x \) to the power of \( y \) is expressed as \( x^y \), which cannot be distinguished from the variable name \( x^y \). Therefore, \( S_p \) is ambiguous. \( \square \)

3.4.2. Documentation-level formats

Storage of mathematical formulas

There are many documentation-level storage formats that support the representation of mathematical expressions. Some examples include MathML in the Presentation Markup, (La)TeX, and the markup within ‘troff’, which uses the tool ‘eqn’ for equations. Mathematical expressions can be created and displayed within word processors such as OpenOffice.org Math and Microsoft Word, as well as in specialized tools such as MathType.

Storage formats and unambiguity

For the sake of generality, the consideration of a concrete selection of existing storage formats is undesirable. Therefore, this discussion will be based on a set of requirements.

Proposed requirements.

RS1 The principle symbols that are expected according to CN3 must be supported in an unambiguous manner.

a) The principle symbols must be stored individually.

b) Mathematical expressions, as well as any symbols used in them, must not be stored as an image.

c) The presence of format information (such as typefaces, font styles and sizes) must not interfere with the recognition of the principle symbols.

RS2 The concept of displaced lines according to CN1 must be reflected so that one line can contain other lines by displacement.

a) For the displacement at least the following two patterns exist: The super-sub pattern and the over-under pattern, cf. Figure 3.13.

b) It must be possible to determine the beginning and the end of each displacement pattern.

RS3 Special operators according to CN4 must be supported either by an environment or by a fixed pattern. The set of operators supported in this way will be referred to as operator environments in the following.
RS4 The representation of all symbols, displacement patterns and operator environments according to CN1 through CN5 must be possible. A mapping from used representing symbols to the concepts supported in a given notation can be surjective.

Note on RS2: The basic offset patterns cf. Figure 3.13 include the patterns in which only one displaced line is used, as shown in Figure 3.14. For the sake of generality, only the basic offset patterns are considered here.

Note on RS4: The focus of this requirement is the support of all concepts. For certain symbols, patterns and operator environments several internal representations may be present that have no effect on the content. For example, if the symbolic notation supports Latin letters for variable names, then the symbols $a$, $a$, and $a$ are equivalent, even though different characters might be used for their representation.

Note on RS1(c): This requirement addresses the same problem as RS4 but is directed to format information that may be present in the expression. The characteristics CN1 through CN5 do not include format information as notational element and such information is not considered in the scope of work. Therefore, the primary aim of RS1(c) is the unambiguous recognition of the symbols.

![Figure 3.13. Basic offset patterns with respect to one line.](image)

![Figure 3.14. Resulting additional offset patterns.](image)
3.4. In-depth consideration of algebraic expressions

Use of the assumptions for documentation-level formats

As with the requirements for symbolic notations, the presented requirements for storage formats are intended to serve as a reference. More specifically, the requirements specify a class of documentation-level storage formats that can be used within the proposed methodology. The demonstrations presented in the following subsection show that the established Goals 2 (parsibility) and 3 (lossless transfer) can be attained using an unambiguous symbolic notation, if the requirements RS1 through RS4 are met by the involved storage formats. To test whether a storage format is applicable, it is sufficient to ensure its compliance with the requirements. In this sense, the requirements can be considered to be sufficient criteria for the applicability of storage formats.

3.4.3. Parsing of non-linear algebraic expressions in documentation-level formats

The representation and parsing of algebraic expressions in programming languages is well covered in computer science and many introductions to the topic can be found, e.g. [AU95]. The general topic of such introductions is the description, assessment and parsing of computer languages. A set of techniques, such as grammars and parse trees, has been established for this task. However, since such techniques are dedicated to languages that work on the level of character strings, they are not applicable to notations with the characteristics proposed above. It is not in the scope of this work to propose an extension of the existing techniques so that they can be applied to the class of notations described above. Instead, the established techniques will be applied to the concepts presented here and will be amended when necessary.

The existing technique of parsing in a recursive manner that results in parse trees is applied, but the pertinent symbols are the underlying concepts instead of concrete character strings. For example, the expression \( \sin(x) \) is considered here as an enclosing operator ‘\( \sin() \)’ and the variable named ‘\( x \)’ as the argument, instead of considering the terminals \( s, i, n, (, x, and ) \). A means to visualize the result of parsing is to use expression trees. An example of an expression tree as it is used in this work is given in Figure 3.15. The use of such trees within a software is twofold: First, if the values corresponding to all variables are known, the postorder tree walk returns the correct value of the expression. Second, if for a given programming language the translations are specified for all operators, numbers and variables, a program code representing the algebraic expression can be generated. Within the frame of this work, the second use is more important. However, the important point is shared by both uses: It must be ensured that the parsing of an expression, i.e. the creation of the corresponding tree, is possible in an unambiguous manner.

Unambiguous parsing

**Statement P1:** A non-linear algebraic expression that is given in an unambiguous symbolic notation according to Statement N1 can be parsed from a documentation-level storage format \( F \) into a correct expression tree, if \( F \) fulfills the requirements RS1 through RS4.
3.4. In-depth consideration of algebraic expressions

Figure 3.15.: A tree of sub-expressions (left) and the corresponding expression tree (right).

To show that Statement P1 is true, a parsing algorithm will be presented that works under the required assumptions. Then a proof of correctness of the algorithm will be given, which also constitutes a proof of Statement P1. It should be noted that the presented algorithm has a principle character, and both the level of detail and the algorithm’s structure have been chosen for comprehensibility. Better solutions to the problem may exist.

**Principle of the parsing algorithm.** An entire line representing a non-linear algebraic expression is considered. If the line contains arithmetic operators, then it can be considered as an alternating sequence of these operators and their operands.\(^3\) In this case, the operands are either variables, numbers, or enclosing operators, and the line can be split up at the position of the operators according to the respective precedence. If the line does not contain arithmetic operators, then the expression must be either of a variable, a number, or an enclosing operator. Variables and numbers represent leaves of the expression tree. Enclosing operators represent nodes of the expression tree, the operands of enclosing operators are lines, which are parsed using this description. A line represents a correct expression according to the notation \(S\), if and only if all its sub expressions are correct. A formal presentation of the above principles is given in Algorithm 1. The following operator precedence is used in the algorithm:

\[
\begin{align*}
+ & \quad 1 \\
- & \quad 2 \\
\times \text{ and } \div & \quad 3
\end{align*}
\]

For the sake of simplification, it is ensured in the algorithm that arithmetic operators always have a left and a right subtree. Therefore, a leaf with the value ‘0’ is introduced as the left operand if an expression contains a leading negative sign, cf. Figure 3.16.

**Assumptions for parsing:** A line containing a complete non-linear algebraic expression is the input of the algorithm. The non-linear algebraic expression is represented using

\(^3\)As described above, brackets are considered as enclosing operators.
Algorithm 1: Parse line

**Input:** a line \( \lambda \) to parse

**Output:** the corresponding expression tree \( T \)

1. if \( \lambda \) has length 0 then
2.  \> return new \( T \) with content 0 (and without children);
3. end
4. test whether \( \lambda \) contains at least one arithmetic operator outside of enclosing operators;
5. if such arithmetic operator was found then
6.  \> find the leftmost arithmetic operator \( \omega_o \) outside of enclosing operators with the lowest precedence;
7.  \> split \( \lambda \) at \( \omega_o \) into lines \( \lambda_{left} \) and \( \lambda_{right} \);
8.  \> generate \( T_{left} \) by applying this algorithm to \( \lambda_{left} \);
9.  \> generate \( T_{right} \) by applying this algorithm to \( \lambda_{right} \);
10.  \> return new \( T \) with content \( \omega_o \) and children \( T_{left} \) and \( T_{right} \);
11. else
12.  \> determine whether \( \lambda \) represents an enclosing operator, a number, or a variable;
13. if \( \lambda \) represents an enclosing operator then
14.  \> determine the actual enclosing operator \( \omega_e \);
15.  \> determine the \( N \) arguments of \( \omega_e \), lines \( \lambda_1..\lambda_N \);
16.  \> generate \( T_1..T_N \) by applying this algorithm to each \( \lambda_1..\lambda_N \);
17.  \> return new \( T \) with content \( \omega_e \) and children \( T_1..T_N \);
18. else if the line represents a number then
19.  \> parse the number \( \eta \);
20.  \> return new \( T \) with content \( \eta \) (and without children);
21. else if the line represents a variable then
22.  \> parse the variable \( \xi \);
23.  \> return new \( T \) with content \( \xi \) (and without children);
24. else
25.  \> return new \( T \) with content ‘error’ // reject expression
26. end
27. end
3.4. In-depth consideration of algebraic expressions

Figure 3.16.: Sub expressions and expression tree of a signed algebraic expression.

a notation that is based on the characteristics CN1 through CN6 and that fulfills the requirements RN1 through RN3. The line is stored in a documentation-level format fulfilling the requirements RS1 through RS4.

**Statement P2:** In Algorithm 1, the code lines that read the storage file and operate on the retrieved lines work correctly.

A proof of Statement P2 is presented in section 11.1. It shows the correctness of the pertinent lines of Algorithm 1 on the basis of the requirements RN1 through RN3 and RS1 through RS4.

**Statement P3:** Algorithm 1 creates a correct expression tree for a non-linear algebraic expression that is stored in a documentation-level format.

**Assumptions:** The statement is subject to the ‘assumptions of parsing’ stated above.

**Proof by complete induction.** To show: For a given number \( n \geq 0 \), a given line \( \lambda \) containing a non-linear algebraic expression \( E \) containing \( n \) operators is parsed correctly into a tree \( T \), which represents the value of \( E \).

**Basis Step.** \( n = 0 \), so \( \lambda \) does not contain any operations. This is the case if (a) \( \lambda \) is empty, (b) \( \lambda \) contains exactly one number, or (c) \( \lambda \) contains exactly one variable. In case (a), the value of the expression is 0, which is correctly reflected in lines 1 and 2. In case (b), the value is given by the contained number, which is realized in the algorithm in lines 12 and 18 through 20. Likewise in case (c), the value is represented by the variable, which is realized in lines 12 and 21 through 23. For cases (b) and (c) it is important that line 12 is reached. Since \( n = 0 \), i.e. \( E \) does not contain any operators, line 12 is reached after the test in line 5.

**Induction hypothesis.** For \( k \geq 0 \) and for every \( n \) with \( 0 \leq n \leq k \), a given line \( \lambda \) containing \( n \) operations is parsed correctly.

**Statement to be shown in induction step.** A given line containing \( k+1 \) operations is parsed correctly.

**Proof of induction step.** Consider an algebraic expression containing \( k + 1 \) operations. Operations are represented unambiguously according to the N2. The algorithm creates a node for one operation, which has the lowest possible precedence: If arithmetic operators are present, then the argument statement in code line 5 must deliver ‘true’, and the operator with the lowest precedence is treated in code lines 6 through 10. If no arithmetic operator is present, the line must contain an enclosing operator, since \( k + 1 > 0 \). In this case, the argument statement in code line 5 must deliver ‘false’, the
argument statement in code line 13 must deliver ‘true’, and the operator is treated in code lines 14 through 17. The lines that are created in code lines 7 and 15 represent the arguments of the operations. They contain $n$ with $n \leq k$ operations, which, according to the induction hypothesis, are parsed correctly by the algorithm (in code lines 8, 9, and 16 respectively). Since one node is created for the operator with the lowest precedence, and since the lines representing the operands are parsed correctly and stored in the subtrees of this operator node, the given line containing $k + 1$ operations is parsed correctly. □

**Proof of Statement P1** The statement is correct, since Algorithm 1 performs the parsing under the stipulated conditions, and since the correct functioning of this algorithm has been proven (Statements P2 and P3). □

It should be noted that the option of subdividing lines, e.g. by ‘<mrow> · · · </mrow>’ in Presentation MathML or ‘{···}’ in Tex, does not stay in conflict with the above demonstrations, since such subdividing does not cause a displacement and consequently, all divided parts can be associated with exactly one line. In practice, a parsing software can either ignore subdivisions or filter them out in a pre-parsing step.

### 3.4.4. Transfer of non-linear algebraic expressions between documentation-level formats

The correct transfer of a non-linear algebraic expression between storage formats should be independent from the symbolic notation used. Furthermore, the parsing, i.e. the retrieval of the expression’s content, is an independent problem and should not be part of the transfer. In order to profit from existing technology, it is desirable to use available software that is specialized for the exchange between storage formats. More specifically, the use of such software is desirable even if it does not consider the mathematical content: If the transfer can be performed without considering the expression’s content and at the same time without any loss of mathematical information, then the corresponding transfer software can be used for various symbolic notations, and the proposed methology can be extended easily to many storage formats.

In order to state whether a lossless transfer under the desired conditions can be performed, a parsing algorithm is considered that works only on the basis of the expression’s presentation. It is assumed that the storage format meets the requirements RS1 through RS4. Accordingly, the storage and the display of mathematic expressions is realized in the form of lines that may contain other lines either through displacements or within special operator environments. The nested lines for the expression

$$x^{n+n\sqrt{2t+1}} + e \cdot h^2$$

are shown in Figure 3.17.
3.4. In-depth consideration of algebraic expressions

Figure 3.17.: Nested lines according to the patterns a through d, introduced in section 3.4.2.

Terminology. In the following, ‘depth’ will be used with respect to nested lines to denote the maximum number of extraction steps that are necessary to obtain lines that consist of pattern a, cf. Figure 3.18. The term ‘structure of nesting’ will be used to denote the structure in which the lines of patterns a through d are nested, as well as the attribution of the contained symbols to the lines.

Lossless transfer

Statement T1: A given line containing a non-linear algebraic expression can be transferred from one documentation-level format to another without changing the structure of the expression, if both formats fulfill the requirements RS1 through RS4.

To show the correctness of Statement T1, one transfer algorithm will be presented and proven.
3.4. In-depth consideration of algebraic expressions

Principle of the transfer algorithm. Each line $\lambda$ may consist of undisplaced symbols $\sigma$ (of pattern $a$) and displaced or enclosed lines according to patterns $b$ through $d_i$. Subsequent patterns of the same type represent a line element $\mu$ and each line can be subdivided accordingly into its elements $\mu_l$ with $l < 0$, cf. Figure 3.18. The line elements are transferred separately according to their pattern. For pattern $a$, all symbols $\sigma$ are transferred into $\sigma'$ and appended to the corresponding line $\lambda'$ of the new storage format. For line elements with other patterns, a new line element $\mu'$ with the corresponding pattern is created. Then the lines $\lambda_i$ contained in $\mu$ are translated into $\lambda'_i$ using the procedure described in this paragraph. Finally, the $\lambda'_i$ are placed into $\mu'$, which is appended to the corresponding line $\lambda'$ of the new storage format. A formal representation of the above is contained in Algorithm 2.

**Algorithm 2:** Transfer line

| Input: a line $\lambda$ to translate |
| Output: the translation $\lambda'$ of the input line |
| 1 | create empty line $\lambda'$; |
| 2 while not end of line $\lambda$ do |
| 3 | read the next element $\mu$ of line $\lambda$; |
| 4 | determine the pattern $\pi$ of $\mu$; |
| 5 if $\pi$ is of type $a$ (undisplaced) then |
| 6 | read symbols $\sigma_i$ of $\mu$; |
| 7 | transfer $\sigma_i$ into $\sigma'_i$; |
| 8 | append $\sigma'_i$ as undisplaced symbols to $\lambda'$; |
| 9 else if $\pi$ is of type $b$, $c$, or $d$ then |
| 10 | extract the whole lines $\lambda_1$,..,$\lambda_N$ contained in $\pi$; |
| 11 | generate $\lambda'_1$,..,$\lambda'_N$ by transferring each line $\lambda_i \in \{\lambda_1,..,\lambda_N\}$ through applying this algorithm; |
| 12 | transfer the pattern $\pi$ into $\pi'$; |
| 13 | create $\mu'$ by assembling the $\lambda'_1$,..,$\lambda'_N$ in $\pi'$; |
| 14 | append $\mu'$ to $\lambda'$; |
| 15 else |
| 16 | error: assumptions not fulfilled; |
| 17 end |
| 18 end |
| 19 return $\lambda'$ |

Assumptions for transfer Both storage formats involved fulfill requirements RS1 through RS4.

Statement T2: In Algorithm 2, the code lines that read from and write to the storage files and the code lines that perform operations on obtained lines both work correctly.

A proof of Statement T2 is presented in section 11.1. It shows the correctness of the pertinent lines of Algorithm 2 on the basis of the requirements RS1 through RS4.
3.4. In-depth consideration of algebraic expressions

**Statement T3:** Algorithm 2 transfers a non-linear algebraic expression from one storage format to another without changing the structure of nesting.

**Assumption:** This statement is subject to the assumptions of transfer.

**Proof by complete induction.** To show: For a given number \( n \geq 0 \) and a given line \( \lambda \) containing nested lines at the depth \( n \), Algorithm 2 creates a copy in the goal storage format that has the same structure of nesting.

**Basis Step.** Lines that have an encapsulation depth of \( n = 0 \) consist of one single line element \( \mu \) which has the pattern \( a \). The contained symbols \( \sigma_i \) are translated in code line 7 and appended to the new line \( \lambda' \) without displacement in code line 8. Thereby, the translated symbols \( \sigma'_i \) are contained in pattern \( a \) in line \( \lambda' \). Since the displacement pattern within the new line is the same as in the original line, the structure of nesting does not change.

**Induction hypothesis.** For \( k \geq 0 \) and for every \( n \) with \( 0 \leq n \leq k \), a given line \( \lambda \) containing nested lines in the depth \( n \) is transfered in a way that does not change the structure of nesting in the goal format.

**Statement to be shown in induction step.** A given line containing \( k+1 \) operations is transfered without changing the structure of nesting.

**Proof of induction step.** Consider a line \( \lambda \) with a depth of nesting of \( k+1 \). If \( \lambda \) contains elements of pattern \( a \), then such elements are translated without changing the structure of nesting according to the basis step above. Since \( k+1 > 0 \), it follows that \( \lambda \) contains line elements \( \mu_l \) with \( l > 0 \) that have the patterns \( \pi_l \in \{b,c,d_j\} \) and contain the lines \( \lambda_{l,i} \) with \( i \geq 1 \). The line elements are determined in code line 4 and transfered in code lines 10 through 14. The contained lines \( \lambda_{l,i} \) are extracted in code line 10 and transfered in code line 11 using this algorithm. Since the nesting depth of \( \lambda \) is \( k+1 \), the \( \lambda_{l,i} \) have a nesting depth of \( n \leq k \) for all \( i \). Therefore, the structure of nesting of the generated lines \( \lambda'_{l,i} \) remains unchanged according to the induction hypothesis. The necessary creation of the respective new line element \( \mu'_l \) with the pattern \( \pi'_l \) and the placement of the lines \( \lambda'_{l,i} \) are performed in code lines 12 and 13. Finally, each \( \mu'_l \) is appended to \( \lambda' \) without displacement in code line 14. Since the \( \mu_l \) are obtained directly from \( \lambda \), i.e. without displacement, and since the \( \mu'_l \) are appended to \( \lambda' \) likewise without displacement, the appending operation does not change the structure of nesting. Consequently, all line elements \( \mu_l \) contained in \( \lambda \) are transfered without a change in the structure of nesting. Since the line elements of pattern \( a \) are also transfered correctly and since in an input line fulfilling the assumptions no other cases exist, the line \( \lambda \) is transfered without a change in the structure of nesting.

\[\Box\]

3.4.5. Combined application

After the considerations above, the target use of the described symbolic notations and documentation formats should be examined. The goal is the exchange of platform-independent models on the documentation level, cf. section 3.5.2. Figure 3.19 shows the processes involved: the creation, transfer, and use of algebraic expressions. In order to
3.4. In-depth consideration of algebraic expressions

![Diagram of algebraic expression creation and use](image)

Figure 3.19.: Use of an algebraic expression after various transfer steps between documentation-level formats.

test whether or not such expressions can be used as platform-independent models in an automated code generation process, it must be determined whether or not a parsing is still possible after a transfer between documentation-level formats.

**Parsing after transfer**

**Statement T4:** The parsability of a non-linear algebraic expression according to Statement P1 is not impaired by the transfer between documentation-level formats according to T1.

**Assumption:** The creation and the parsing of the expression are subject to the assumptions of parsing. The transfer of the expression is subject to the assumptions of transfer.

**Proof.** A line created under the above assumptions can be parsed correctly according to the proven Statement P1. As far as the stored symbols are concerned, the parsing is based on the placement of the symbols within the nested lines, the structure of nesting. Since the structure of nesting does not change according to the proven Statement T1, the expression is parsable after the transfer. □

**Consequences and applicability**

Symbolic mathematic notations were described by a set of characteristics (CN1-CN6). Usually notations with such characteristics lead to ambiguities in the representation of formulas. For non-linear algebraic notations, it was shown that ambiguity can be avoided through compliance with the requirements for notations RN1-RN3. Furthermore, requirements for documentation-level storage formats were presented (RS1-RS4). It was shown that parsing and transfer can be done unambiguously for non-linear algebraic expressions stored in formats that comply with the presented requirements.

It should be noted that the above considerations represent a separation of concerns: The use of a non-linear algebraic expression can be subdivided into its creation, the transfer between storage formats and its parsing for the generation of program code. According to Statement T4, it is possible to create the formula in any documentation-level storage format that fulfills the proposed requirements. Accordingly, off-the-shelf software can be used for the creation and editing of the mathematical formulas used. Likewise, existing transfer software can be used assuming that the structure of the expressions is not changed during the transfer. A parsing software that is restricted to
one documentation-level format can therefore be used for expressions that have been created in other formats, cf. Figure 3.19.

The requirements RS1 through RS4 are met by common markup-languages such as Presentation MathML or Tex. Whether or not propriety formats, such as Microsoft DOC, do meet these requirements cannot be answered within the scope of this work. However, the presented methodology has been successfully applied to the use of algebraic expressions created in Microsoft Word 2007.

3.5. Discussion

3.5.1. Compatibility of models in the documentation level

When compared to the generated program code, the documentation-level models carry additional information such as mandatory descriptions of certain model elements, keywords for registration in model repositories, composition of variable names, description of the symbols used in such names, etc. Therefore, data models of numerical problems on the documentation level describe more information than those on the implementation level. Accordingly, a mapping from the documentation to the implementation level is generally not injective, and therefore, an unambiguous inverse mapping cannot be obtained. The mapping between different model representations within the documentation level, i.e. between different symbolic notations and different storage formats, is bijective, if both representations use the same data model and if both involved symbolic notations unambiguously represent the elements of the underlying data model, cf. the example in section 3.4.1.

3.5.2. Effort of exchange

Software dedicated to parsing and code generation is unlikely to support every documentation-level format. The transfer of models with the help of external software is therefore inevitable, and it must be estimated how many tools $N_{tool}$ and transfer steps $N_{step}$ are need. It is assumed that there are $m$ different storage formats and that the tools only translate in one direction.

a) If a direct translation translation between all formats is desired, i.e. $N_{step} = 1$, then the number of tools is (see Figure 3.20 (1))

$$N_{tool} = 2 \cdot \sum_{i=1}^{m} (m - i) = m \cdot (m - 1)$$  \hspace{1cm} (3.13)

This not desired, since $N_{tool}$ is of the order $m^2$.

b) The number of tools can be reduced in a way that the resulting graph is strongly connected (see Figure 3.20 (2)). Then the effort is

$$N_{tools} = 2 \cdot (m - 1)$$  \hspace{1cm} (3.14)

$$N_{steps} \leq m - 1$$  \hspace{1cm} (3.15)

In this case, $N_{steps}$ is high in the worst case.
c) If it is assumed that $n$ of the $m$ storage formats are well established and tools exist for the translation to and between them (see Figure 3.20 (3)), the effort is

\[
N_{\text{tools}} = 2 \cdot (m - 1) \quad \text{(3.16)}
\]
\[
N_{\text{steps}} \leq n + 1 \quad \text{(3.17)}
\]

In this case $N_{\text{tools}}$ is the same as in b. However, $N_{\text{steps}}$ is small, if the number of established languages is small ($n \approx 3$).
4. Modular modeling

Modularization is an important means for structuring models: "Common to all approaches are a multi-level modularization of process models" [Mar96]. The modularization helps to cope with complexity; furthermore it provides a basis for reuse. The decomposition of a plant model into “building blocks” that represent clearly defined concepts and can be used in different models is an important technique. The smallest possible scale of such building blocks differs within the tools starting from a term of an equation representing a certain phenomenon and going up to equation systems or algorithms describing a balance volume representing a certain phase or an entire operating unit. A good model base of such elements can be used in generalized approaches such as COCO [vB12] and the COGents system [YBF+08]. The desired decomposition of a PSE simulation problem in reusable elements is motivated in the following.

4.1. Supported mathematical problems

The problems supported by MOSAIC at the time of writing can be subdivided into three classes according to the underlying equation system. First of all, non-linear algebraic equation systems (NLE’s) can be described, which are of the general form

\[ g(y,u) = 0 \]  

(4.1)

where the \( y \) and the \( u \) represent the “algebraic” variables and the “parameters” respectively [BSMM07, Pan88]. When addressing the numerical aspects of process systems engineering (PSE), the quantities are also referred to as “iteration” variables and “design” values [CH01]. In PSE, this class of problems corresponds to steady-state calculations.

Second, ordinary differential equation systems (ODE’s) can be formulated. The equations used to describe such systems are explicit first-order ordinary differential equations, which are of the form

\[ h \left( x; \frac{d}{dz} x, u, z \right) = 0 \]  

(4.2)

where the \( x(z) \) represent the “state” or “differential” variables, and the \( z \) stands for the “independent” variable.

Third, there are differential-algebraic equation systems (DAE’s), which are of the form

\[ f \left( x, \frac{d}{dz} x, y, u, z \right) = 0 \]  

(4.3)

\[ g \left( x, y, u, z \right) = 0 \]  

(4.4)

In PSE, the second and the third classes of problems are used to model transient behavior over a continuous quantity that often, but not exclusively, represents time. All quantities involved (\( x, y, u, z, \) and \( v \)) are considered to be real numbers.
4.2. Model information within mathematical expressions

Classification values of variables In order to achieve consistency in the model, a correct classification of the system’s variables is necessary. Obviously, for NLE’s there are only two possible classifications, ‘algebraic variable’ and ‘parameter’, whereas for variables of DAE’s the possible classifications are ‘state variable’, ‘parameter’, and ‘independent variable’. Depending on the equation system considered, there is a range of choice for the classification. However, related decisions are not arbitrary and represent an important characteristic of the simulation problem. The information represented by non-linear algebraic equations varies according to the classification of the variables contained, [BSMM07, CH01]. Therefore, different simulation problems can be formulated based on identical sets of non-linear algebraic equations, which suggests a separation of the variable classification from the definition of the equation system. In the case of differential equations, the classification of the state variables and the independent variable is defined in the differential operators of the mathematical expression; a user-defined classification of such variables would therefore be redundant.

In conclusion, it can be stated that, in order to enhance the reusability of equations and equation systems, the classification of variables should not be stored in the equation system. Since the classification of some variables is defined by their use within mathematical expressions, a certain part of the classification can be performed automatically. Within the range of choice left between algebraic variables and parameters, the user may formulate different problems based on a given set of equations. In this process, the user must observe the restrictions that are pertinent with respect to linear independence, in order to correctly specify the equation system.

Similar considerations, combined with the intention to enhance reusability, lead to the conclusion that the values of the variables should not be stored within the equation system.

Scalable formulation of models A common way to write equations with a general form is to furnish variable names with indices. In a PSE context, such indices are often used to relate variables to components, streams, phases, etc. Since such equations are frequently used, they should be supported by the modeling concept.

Equations using indices can look similar to

\[\frac{d}{db} a_l = c_{l-1} + d_{l+1} - c_l - d_l + e_l\] (4.5)

with

\[l = 1..NL \ (l, NL \in \mathbb{N})\] (4.6)

Equation (4.5) has a general form, since it stands for a coupled subset of equations with the same structure. The size of the represented subsystem is reflected in the expression (4.6), and the actual number of equations and variables introduced through (4.5) depends on the value given to \(NL\). Equations using indices in this way will be referred to as generic in this work.
For the case $N_l = 6$, equation (4.5) represents the following subsystem:

\[
\begin{align*}
\frac{da_{l=1}}{db} &= c_{l=0} + d_{l=2} - c_{l=1} - d_{l=1} + e_{l=1} \\
\frac{da_{l=2}}{db} &= c_{l=1} + d_{l=3} - c_{l=2} - d_{l=2} + e_{l=2} \\
\vdots
\frac{da_{l=6}}{db} &= c_{l=5} + d_{l=7} - c_{l=6} - d_{l=6} + e_{l=6}
\end{align*}
\] (4.7) (4.8) (4.9)

In this work, equations that are created based on generic equations, such as equations (4.7) through (4.9), will be referred to as instantiated.

It should be noted that different assignments of $N_l$ represent different applications of (4.5). Accordingly, in order to maintain the reusability of generic equations, the specification of values for index maximum value symbols should be separated from the equation and consequently from the equation system.

4.3. Equations and functions as building blocks

**Reuse of equations** Different models describing the same phenomenon or technical process may have equations in common. For example, a certain operating unit may be represented by several models that have different degrees of detail or are subject to different assumptions. Therefore, some of those models may use the same overall energy balance, another group of those models may use the same mass balance, and so forth. It is very likely that these groups will overlap. However, it can not be assumed that one group completely includes the other.

It seems to be appropriate to avoid multiple implementations of identical equations, since first, errors can be introduced in every instance during the model implementation and second, eventual changes in common equations necessitate updates in all instances. In order to avoid reimplementation of identical equations, inheritance techniques adopted from object-oriented modeling could be used. The resulting inheritance trees are well suited for situations in which the models are structured hierarchically, i.e. where the common use of certain equations is exclusive and therefore represents the characteristic of a sub class of models. However, in situations where the common use is non-exclusive, hierarchical structuring obtained from inheritance concepts is not effective for avoiding the reimplementation of equations. It should be noted that in most existing modeling approaches, the equations are a fixed part of some kind of model building block. However, if identical equations are shared by different models, then a modeling concept should be considered that treats the equation itself as an independent building block. Based on a modeling concept that supports the reuse of individual equations, it is possible to create both hierarchical and heterogeneous model structures with a very high degree of reusability of equations.

**Explicit functions** In some situations, it is common practice to calculate values of variables directly through a function call. In PSE, this technique can be used for the
4.4. Domain-specific concepts

Since the modeling environment presented in this work is dedicated to the domain of process systems engineering, additional concepts need to be reflected in the modeling concept.

Units, ports, and streams First, it should be possible to model processes in a modular way, so that elements representing individual units are set apart from each other. Consequently, the concept of a unit containing ports should be present as well as the concept of streams connecting the units. The pertinent conceptual models should be adaptable to the modelers needs and to different concepts: The definition of what quantities are contained in the ports should be provided by the modeler and not by the modeling environment.

External function calls A further important feature of a PSE modeling environment is the possibility to perform calls to thermodynamic libraries, in order to obtain physical properties or make use of external calculations of non-trivial sub problems, e.g. the evaluation of equations of state.

4.4.1. Documentation specific point of view

Meta information In this work it is assumed that model documentation must contain the model equations and provide textual explanations of the assumptions. Therefore, a new modeling concept should consider textual descriptions as an important model element instead of an optional feature. The underlying assumptions should be laid down for each equation, each equation system, and more generally, for each building block.

Symbolic notation Following the model presentation in the literature, the mathematical content should be presented in a human-readable form, preferably in two-dimensional symbolic expressions, as discussed in section 3.4.

Terminological notation The model presentation in the literature often contains a nomenclature section, which describes all symbols used in the mathematical expressions. The presentation of the terminological notation serves as a reference for the meaning of the variables used. In many cases, the naming of the variables is given a systematic character by reflecting specific concepts in baseline symbols, superscripts, or subscripts. Although there are certain conventions for the naming of quantities, deviating or new naming concepts may be used. Nomenclatures can be attributed to articles or entire books, in some cases individual nomenclatures are assigned to specific sections.
of books. This practice in the literature can be summarized as follows: Each nomenclature is assigned to a set of equations, and each equation references a nomenclature, which is shared by several equations or equation systems. Therefore, if equation systems are assembled in the documentation level, e.g. as a preparation for programming a new model, and if the literature sources use different nomenclatures, the variable names must be transformed, which generally results in a new nomenclature, even if the latter is unwritten. Specifying a nomenclature for such assembled equation systems adds an additional element of description to the model, and therefore makes them more comprehensible. A modeling concept that works on the documentation level should reflect the nomenclature. According to the multiple application of model elements described above, the nomenclature should be an individual, reusable model element.

4.5. Use of the documentation-level models

Language specification As described above 3.3 the documentation-level model represents a meta-level model that can be used as a basis for code generation, which is subject to an exact specification of the transformation. Such specifications contain rules for the translation of each abstract model element into an actual platform-specific program code fragment, and will in this work be referred to as language specification in this work. Since it is desirable to transform the meta-level models into different languages, the language specification should be a modular model element.

Engineering units For general meta-level models of the presented modeling concept, engineering units are not considered. This goes in line with many existing general-purpose numerical tools, such as BzzMath [BF10], GSL [Gal09], Matlab [Mat12], and GAMS [Ros12]. In certain cases, however, reflecting engineering units within the program code is indispensable. For example, if the code for an operating unit is intended to be used in a flow sheeting environment, the quantities contained in the ports need to have engineering units. In such cases, the interfaces existing in the platform, including the engineering units, must be reflected in the meta-level model so that the user can apply the models correctly. Applying engineering units in this way constitutes an adaption to specific cases. Therefore a punctual, well-directed support of engineering units should be implemented, through which the user can respond to individual platform-specific demands.

4.6. Conclusion

A complete documentation-level model created to perform a simulation task, must contain both the information necessary to describe the numerical problem, and a detailed documentation of assumptions, as well as additional descriptions to make the model more comprehensible. In order to make models more reusable, the complete model information should be divided up into sensibly small elements. The model elements obtained in this way should be exchangeable and applicable in different situations. The considerations presented in the above section can be summarized as follows:
4.6. Conclusion

- The supported types of equation systems are NLE’s, ODE’s, and DAE’s.
- The classification of identifiers into algebraic, state, independent, and fixed variables (parameters) should not be stored in the equation system.
- Accordingly, equations and equation systems should not contain interfaces, since they would represent a classification of variables and therefore restrict reuse.
- The formulation of generic equations should be supported. The maximum values corresponding to the generic indices should not be stored in the equation system.
- The equation should be present as an independent model element.
- It should be possible to define functions.
- The PSE concepts of units, ports, and streams should be reflected in the modeling concept.
- External function calls should be describable within the modeling concept.
- Text descriptions should be a part of all model elements.
- The terminological notation should be introduced as a model element to reflect the nomenclature of model presentations in the literature.
- Engineering units should be reflected in a platform-directed manner.
- The transformation directives for code generation, i.e. the language specification, should be present as a model element.

The application of the above points is presented in chapter 7.
5. Use of the Internet

As stated above, the centralized storage of model files enhances the reusability of models, and improves cooperation between different work groups. Apart from that, there are more advantages to internet-based software systems that work independently from local work stations, i.e. ‘in the cloud’. The term Cloud Computing was coined for the paradigm according to which a functionality is provided by the Internet and the provision of this functionality is partitioned into subtasks that are performed by different web services. This paradigm is advantageous with respect to accessibility, scalability and maintenance (cf. [BYV08, Lin09]): With strictly internet-based tools (i.e. tools where no software is installed on the local computer) all users automatically use the same version of the software. The maintenance of the software is managed centrally and necessary changes in the system are automatically applied to all users. Changes in local work stations etc. do not affect the usability of the software as long as they do not affect Internet access; accordingly there is no local maintenance. Furthermore, network resources for data storage and for computation can be scaled easily.

5.1. Application in MOSAIC

The application of this aspect is discussed here directly, since the small amount of content does not justify an entire chapter.

The MOSAIC modeling environment is programmed in Java and the graphical user interface is included in a Java applet. The tool is used in a web browser. The presence of the Java runtime environment on the local machine is assumed. The user interface of MOSAIC can be integrated in existing web applications (such as orangeProject/orangeScience by Conveloper [WGK09]); a stand-alone interface is provided at http://www.mosaic-modeling.de.

There are three maintenance aspects: The deployment of the software’s actual binaries, the storage of model and user data, and the numerical computation of the models. All three aspects are independent and can be operated on different servers.

It should be noted that even though MOSAIC permits calculation of the models on a modeling server, an important functionality is the use of the models through generated program code in locally-installed software such as Matlab, Aspen Custom Modeler, etc. Thereby, resources of the local machines are often used, and the expected computational burden on the modeling server is comparatively small.

1The engine of the modeling environment is separated from the user interface in order to make a change in technology or the use of other graphical user interfaces possible.
Part II.

Application
6. Symbolic mathematic expressions

6.1. The symbolic notation used in MOSAIC

6.1.1. Introduction with the help of examples

Before the symbolic notation supported by MOSAIC is described in detail, some of its features will be introduced with example expressions. The operations in the symbolic notation use special symbols and displacements as discussed in section 3.4.1. Each variable name may use several superscripted and subscripted symbols, the displacements used for variable names are restricted to the structure □□□. The expression

\[ a^L + b^{L,M} \cdot \frac{(b^L)^{P_a}}{a^r + 3} = 4 \]  

(6.1)

is valid in the notation used. The variable names may have indices that consist of a name and a value, which can either be given directly as in

\[ b_{i=1} + c_{j=5} = d_{i=2,j=6} \]  

(6.2)

or in a generic form as in

\[ \frac{d}{d \ b} a_l = c_{l-1} + d_{l+1} - c_l - d_l + e_l \]  

(6.3)

and in

\[ B_{i-N_i} + \alpha = \frac{A_{j=N_j-1}}{C_{k=N_k+1}} \]  

(6.4)

The elements of a variable naming are shown in Figure 6.1. A comprehensive explanation of the symbolic notation used is presented in the next sections.

It should be noted that the distinguishing property of indices is the name instead of the order. This is different from conventional symbolic notations and programming languages where the order of the indices is the distinguishing property. However, constantly using the name of the index together with its value has the advantage that misinterpretations are avoided. This advantage is strengthened, if the use of a terminologic notation is assumed so that the description of the index name can be accessed at any time. In other words, in MOSAIC the user reads a model where the name and description of an index is clearly specified. In contrast, in conventional languages the user must deduct the conceptual meaning of an index from its placement within other numbers – it should be kept in mind that this placement depends on the preferences of the modeling engineer.

In the following, the symbolic notation used is described in terms of the methodology presented in section 3.4.
6.1.2. The data model

**Equations** The data model of the mathematic expressions used in MOSAIC contains two classes of ‘equation expressions’: the ‘differential equation’ and the ‘algebraic equation’. The ‘algebraic expression’ is a building block of both classes. The differential equation is composed of an ‘equality operator’, one ‘differential expression’ on its left side and one algebraic expression on its right side. The algebraic equation is composed of an equality operator and two algebraic expressions, cf. Figure 6.2.

**Algebraic expressions** In order to avoid a succession of operators when using negative signs, the data model holds two classes of algebraic expressions: the ‘unsigned algebraic expression’ and the ‘signed algebraic expression’, which is an unsigned algebraic expression equipped with a negative sign at the beginning. There are exactly four classes that represent unsigned algebraic expressions: ‘Unsigned numbers’, ‘variable namings’, ‘arithmetic expressions’ and ‘enclosing-operator expressions’. The arithmetic expression consists of an ‘in-line operator’ and two ‘operands’ that are both unsigned algebraic expressions. The enclosing operator expression consists of an ‘enclosing structure’ and one or several operands, which are algebraic expressions (signed or unsigned). The UML diagram of the described structure is shown in Figure 6.3. The actual operations contained in arithmetic expressions and enclosing operator expressions are shown in Figure 6.4.

![Figure 6.1.: Elements of a variable naming.](image1.png)

![Figure 6.2.: Equations in the data model.](image2.png)
6.1. The symbolic notation used in MOSAIC

Variable Naming  The variable naming is composed of one ‘base name’, one fixed ‘subscript’, several ‘superscripts’, and ‘indices’. Only the ‘base name’ is required, the number of superscripts and indices is not fixed. Indices are characterized by their ‘index identifier’, which has a corresponding ‘maximum value symbol’. All the elements introduced are of the type ‘symbol’. There are two possible types of indices: the ‘index instance’ which has an actual value, and the ‘generic index’, which is used to represent one or several equations. The generic indices are translated into index instances in an instantiation step. The number of variables generated in the instantiation step depends on the value specified for the maximum value symbol. Several ways of controlling the behavior of generic indices are expressed by ‘instantiation directives’. The UML class diagram corresponding to a variable naming is shown in Figure 6.5, see also Figure 6.1.

Unsigned numbers  In the data model used, the unsigned numbers represent real numbers $r$, so that $r \in \mathbb{R}^{\geq 0}$.¹ Real numbers contain an integer $z$ and a fractional part $f$, which are combined by the decimal separator $\sigma$ in the form of

$$z\sigma f$$

so that for unsigned numbers it is

$$n\sigma f$$

with the natural part $n \in \mathbb{N}$ and the fractional part $f \in \mathbb{R}, 0 \leq f < 1$. Furthermore, the e-notation can be used to express multiplication by powers of ten. With a delimiter $\eta$,

¹It should be noted that in MOSAIC, integer numbers can be written without the decimal separator and the fractional part; internally, integer numbers are treated as real numbers.
6.1. The symbolic notation used in MOSAIC

Figure 6.4.: In-line operators and functions in the data model.

usually ‘E’ or ‘e’ and an exponent \( t \in \mathbb{Z} \), the e-notation is expressed as

\[
n\sigma f \cdot 10^t = n\sigma f\eta t
\]  

(6.7)

So that, if \( n = 3, f = 0.764, t = 3, \sigma = '; (as is the case in Germany), and \( \eta = 'e', \) equation 6.7 reads

\[
3,764 \cdot 10^3 = 3,764e3
\]  

(6.8)

For the sake of completeness, the UML diagram of an unsigned number is shown in Figure 6.6.
6.1. The symbolic notation used in MOSAIC

Figure 6.5.: Data model of possible variable names.

Figure 6.6.: Data model of unsigned real numbers.
6.1. The symbolic notation used in MOSAIC

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbols</th>
<th>Description</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition Operator</td>
<td>( a + b )</td>
<td>Fraction</td>
<td>( \frac{a}{b} )</td>
</tr>
<tr>
<td>Subtraction Operator</td>
<td>( a - b )</td>
<td>Power</td>
<td>((a)^b)</td>
</tr>
<tr>
<td>Multiplication</td>
<td>( a \cdot b )</td>
<td>Exponential</td>
<td>( \exp(a) )</td>
</tr>
<tr>
<td>Division Operator</td>
<td>( a/b )</td>
<td>Logarithm</td>
<td>( \ln(a) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cosinus</td>
<td>( \cos(a) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sinus</td>
<td>( \sin(a) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Root</td>
<td>( \sqrt{a} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Brackets</td>
<td>( (a) )</td>
</tr>
</tbody>
</table>

Table 6.1.: Symbolic notation for the representation of operators.

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summation</td>
<td>( \sum_{a=1}^{b} c_a )</td>
</tr>
<tr>
<td>Differential Operator</td>
<td>( \frac{d^a}{b} )</td>
</tr>
<tr>
<td>Equality</td>
<td>( a = b )</td>
</tr>
</tbody>
</table>

Table 6.2.: Additionally used symbols.

6.1.3. The symbolic representation

Operators

**Arithmetic and enclosing operators** The symbols used to express the operations are presented in Table 6.1.

**Other operators** Table 6.2 shows further operators that are supported by the symbolic notation, but not discussed in more detail. Accordingly, no corresponding data model is presented.

Variable Namings

As shown in Figure 6.5, Variable Namings consist of a combination of symbols. Each symbol is a character string. Valid characters are small and capital letters from the Latin and Greek alphabet. The character strings used as operators in Table 6.1 ii, \( \exp, \ln, \sin, \) and \( \cos \), cannot be used as symbols.

If more than one superscripted or subscripted symbol is used, then they must be separated by a comma. In the case of index instances, the index identifier is displayed together with the index value and both are separated by an equals sign. The instantiation directives of generic indices are represented in Table 6.3. For the sake of completeness, it should be noted that other symbolic notations could be used to represent the algebraic expressions supported by the given data model. One example would be to use other
6.1. The symbolic notation used in MOSAIC

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>All values</td>
<td>( j )</td>
</tr>
<tr>
<td>All values and offset</td>
<td>( j + 1, j - 2, \ldots )</td>
</tr>
<tr>
<td>Maximum value</td>
<td>( j = N_s )</td>
</tr>
<tr>
<td>Maximum value and offset</td>
<td>( j = N_s + 1, j = N_s - 4, \ldots )</td>
</tr>
</tbody>
</table>

Table 6.3.: Symbolic notation of index instantiation directives, if the index name is \( j \) and the maximum value symbol is \( N_s \). A definition of index expressions in the Backus-Naur Form is presented in section 11.2

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural part</td>
<td>Arabic numerals</td>
</tr>
<tr>
<td>Fractional part</td>
<td>Arabic numerals</td>
</tr>
<tr>
<td>Decimal separator</td>
<td>‘.’</td>
</tr>
<tr>
<td>E delimiter</td>
<td>‘e’ or ‘E’</td>
</tr>
<tr>
<td>E exponent</td>
<td>Arabic numerals</td>
</tr>
</tbody>
</table>

Table 6.4.: Symbolic notation of numbers.

ways of separating superscripted symbols\(^2\). However, without any separation, multiple superscripted symbols containing several characters cannot be distinguished, so e.g. \( x^{LI} \) could not be resolved, since there are the two options \( \{L,I\} \) and \( \{LI\} \).

It should further be noted that in order to ensure unambiguity, two pieces of information must be specified outside of the algebraic expressions. First, any subscript must be declared as such, in order to distinguish it from indices. Second, the maximum-value symbols must be attributed to the corresponding indices. In MOSAIC, this additional information is stored in the model element Notation, cf. section 7.1. This model element represents the terminological notation and contains a description for each used symbol.

**Numbers**

The symbolic notation of the unsigned numbers described above is expressed as shown in Table 6.4. It should be noted that even though the used representation of numbers is common in science, the description of that representation is an important part of the symbolic notation, since other notations are possible, and a common understanding in this point cannot be assumed.

**6.1.4. Application of the methodology**

In the following, the methodology presented in section 3.4 is applied to the symbolic notation used. The aim is to show that the notation is unambiguous and that it can be used as a meta model for code generation.

\(^2\)cf. example in section 3.4, page 30
Definition and characteristics

The applicability according to the characteristics defined in section 3.4.1 on page 28 is verified as follows: The presented notation does use the displacements

\[ \square \text{ and } \square \] (6.9)

and all displacements are used one degree at a time. The fact that the displacement patterns

\[ \square \text{ and } \square \] (6.10)

are not used, constitutes a special case. However, the characteristic CN1 applies, since only stipulated displacement patterns are used. The characteristic CN2 applies, since the variable names have the structure

\[ \square \square \] (6.11)

which is a special case of

\[ \square \square \] (6.12)

The characteristics CN3 through CN5 apply, since the symbolic notation used includes the use of Greek letters, special symbols for operators (e.g. ‘\cdot’ for multiplication), and first and second-group operators (cf. Table 6.1(ii)). CN6 applies, cf. Figure 6.3.

The presented notation is defined (in the previous pages) according to the checklist introduced for that purpose in section 3.4.1 on page 29: The ‘specification of the supported data model’ is given in section 6.1.2 and the ‘specification of the symbolic representation’ is given in section 6.1.3.

Fulfillment of requirements

The fulfillment of the requirements for unambiguity of symbolic notations introduced in section 3.4.1 on page 30 is discussed in the following. For the sake of conciseness no formal proofs are presented.

Variables are represented using a different set of characters than numbers. Therefore, numbers are distinguishable from variables and requirement RN1 a) is fulfilled. In order to represent numbers unambiguously, the existing conventions for that purpose have been used as described above; RN1 b) is thereby fulfilled. In the presented notation variable names are considered to be equal, if and only if they contain the same base name, superscripts, subscript, and indices, and if the contained indices have the same values. Since all symbols and values used in that definition of equality are displayed, different variable names are distinguishable from each other and RN1 a) is fulfilled. As for the distinguishability of operations, Table 6.1 shows that all operations have a different structure, i.e. a different combination of displacements and symbols. Therefore, RN2 is fulfilled.\(^3\) The fulfillment of requirement RN3 can be verified easily for the

\(^3\)It should be noted that the operators presented in Table 6.2 are not considered here, since they are not used in algebraic expressions, cf. Figure 6.3 and 6.4.
6.2 Storage formats

arithmetic operators (cf. Table 6.1(i)): The symbols used for the representation of the operation are not allowed as symbols in variable names. The symbols ‘+’ and ‘-’ can be contained within the subscripted symbols of generic variable names (cf. Table 6.3 and eq. 6.3), however, since there is no operation that is represented with an algebraic expression as an argument, a distinction between usage in instantiation directives and usage as arithmetic operator can be made. Thereby, all possible variable names can be distinguished from arithmetic operators. As for the enclosing operators, Table 6.1(ii) shows that the operations ‘root’ and ‘fraction’ have a structure that separates their operands from each other, as well as from the surrounding expression, and thereby from other operands. The other contained enclosing operators use brackets to make their operands distinguishable from the surrounding expression. The names of enclosing operators like ‘sin’ are reserved keywords that cannot be used in variable names. In conclusion, it was verified that requirement RN3 is fulfilled.

Significance It was shown above that the presented notation has the characteristics CN1 through CN6 and that it fulfills the requirements RN1 through RN3. According to the methodology presented in section 3.4, the notation above can therefore be used to represent algebraic expressions that serve as platform-independent models within a model-driven software engineering approach. More specifically, program code can be generated unambiguously for stored expressions, and the transfer of expressions between documentation-level formats can be performed without loss using off-the-shelf software.

Additional remarks It should be noted that the representation of the multiplication by an actual symbol, as well as the representation of the operator ‘power’ by \((a)^b\) follows Statements N2 and N3, cf. section 3.4.1. It should further be noted that the operators listed in Table 6.2 are not discussed in the methodology and therefore, their unambiguity cannot be shown with the help of the stated requirements. The extension of the methodology to include these operators is possible but is not in the scope of this work.

6.2. Storage formats

The methodology presented in section 3.4 will now be applied to two storage formats.

LaTeX Symbols are represented individually, the formulas are not stored as images, and format information is given in a way that the symbols themselves are still recognizable. Therefore, requirement RS1 is fulfilled. As far as requirement RS2 is concerned, the necessary patterns are supported through \(\texttt{\textbackslash underset}\{\texttt{\textbackslash overset}\{\texttt{\textbackslash overset}\}\}\) and \(\_\_\_\_\). Furthermore, the beginning and end of displaced lines can be determined: Displaced lines are placed in curly brackets. If no curly brackets are used in displacement patterns, then the displaced line contains exactly one symbol. As far as requirement RS3 is concerned, the corresponding patterns are present: \(\texttt{\textbackslash sqrt}\{\texttt{\textbackslash sqrt}\}\) and \(\texttt{\textbackslash frac}\{\texttt{\textbackslash frac}\}\). Requirement RS4 is fulfilled, since all letters, displacement patterns, and special operator environments can be represented.
MathML in the Presentation Markup  Requirement RS1 is fulfilled, since all symbols are represented through a text code and therefore recognizable. The necessary patterns for requirement RS2 are the elements \texttt{munder}, \texttt{mover}, \texttt{msup}, and \texttt{msub}. Displaced lines can be identified since they are the child elements of the above displacement environments. The content of the displaced lines can be determined unambiguously through considering the type of the child elements. The environments necessary for requirement RS3 are the elements \texttt{msqrt} and \texttt{mfrac}. Requirement RS4 is fulfilled that thanks to existing entities such as \&alpha;, \&beta;, etc.

Additional notes  It should be noted that both LaTeX and Presentation MathML offer a subdivision of lines that does not include a displacement (in LaTeX through \{, in Presentation MathML through \texttt{mrow}). However, since all symbols of such subdivisions can be associated unambiguously with the containing line, the methodology can still be applied.

It should further be noted that Presentation MathML contains additional information that is not considered in the proposed methodology. In MathML, the elements \texttt{mo}, \texttt{mi}, and \texttt{mn} are provided for the representation of operators, identifiers, and numbers. The necessary information for defining a symbolic notation introduced in section 3.4.1, however, does not include any rendering information or a-priory attribution of symbols to terminals or operators. The semantic notion of the above elements is therefore not considered in the algorithms used in MOSAIC. This leads to a more robust approach of recognition, since contradictions in the input such as \texttt{<mi>1.3<\mi>} are ignored (i.e. 1.3 will be recognized as a number according to the definition of symbolic notation). The same applies for the segmentation of symbols as in the expression \texttt{<mi>H</mi><mi>U</mi> for the variable name HU. The preservation of unambiguity in this respect is important for the fulfillment of requirement RS1(c).

6.3. Specialized storage formats

6.3.1. MosaicLatex

MosaicLatex is a subset of LaTeX that has been introduced as a means for the quick and user-friendly input of mathematical expressions. The set of available LaTeX commands is restricted to the ones that are necessary for the definition of mathematical expressions following the symbolic notation used.

\texttt{\cdot \frac{}{} \sin \cos \exp \ln \sum \limits \sqrt{}}

Furthermore, all commands for the representation of Greek letters are supported:

\texttt{\alpha \beta \gamma ... \Alpha \Beta \Gamma ...}

Additional rules  Small differences between MosaicLatex and LaTeX will be described next. Curly brackets are required for all replacements, even if only one character is involved. When using the summation command, the abstract operand must be enclosed in curly brackets, as in
For the expression of the total differential operator in DAE systems, the command \( \texttt{diff\{\}} \) has been introduced, so that \( \texttt{diff\{a\}\{b\}} \) is rendered as

\[
\frac{d\ a}{d\ b}
\]

It should be noted that in order to make a translation by available LaTeX compilers possible, the \( \texttt{\textbackslash diff\{}\texttt{\}} \) command must be defined by\(^4\)

\[
\texttt{\textbackslash newcommand\{\textbackslash diff\}[2]\{\textbackslash frac\{d\ \#1\}\{d\ \#2\}\}}
\]

**Syntax and semantic checks** A compiler for MosaicLatex has been implemented which creates a subset PresentationMathML that will be described next. During the translation, the compiling software performs two tests simultaneously: First, it checks, whether or not the given character string corresponds to the MosaicLatex syntax. Second, it checks whether or not the resulting mathematical expression corresponds to the symbolic notation described above. If a syntactical or semantical error is detected, the user is informed of the location and type of error. The compliance of MosaicLatex with the requirements for documentation-level formats can be shown in a similar way to LaTeX, see above.

### 6.3.2. MosaicMathML

In MOSAIC, Presentation MathML is the core documentation-level format. During the modeling process, different parsing operations and transformations of the mathematical expressions are necessary, which are directly performed on the MathML expressions; more specifically, the mathematical content is never translated to an internal object model, but remains held in Presentation MathML documents. This increases the transparency, since the result of transformations on the mathematical expressions can be visualized without any offset resulting from the regeneration of the presentation markup. However, even if access to the mathematical content is possible according to section 3.4, the parsing effort is higher when using presentation concepts instead of content concepts, such as Content MathML. In order to remedy this shortcoming, additional patterns have been introduced, which reflect the elements of the data model presented above. In addition, as was the case for MosaicLatex, the set of possible MathML entities has been reduced to the ones necessary within MOSAIC. The resulting documentation-level format MosaicMathML is a subset of Presentation MathML, i.e. all expressions valid in MosaicMathML are also valid in Presentation MathML and can be rendered by common web browsers and dedicated MathML software. The specific rules of MosaicMathML are described in the following.

\(^4\)Partial and higher derivatives are not supported yet. A possible symbolic notation of the partial differential operator is \( \frac{\partial^2 a}{\partial x \partial y} \), which could be represented in MosaicLatex by the command \( \texttt{\textbackslash pdiff\{}\texttt{\}} \texttt{\}} \texttt{\}} \) defined in LaTeX by \( \texttt{\textbackslash newcommand\{\textbackslash pdiff\}[3]\{\textbackslash frac\{\textbackslash ^2 d\ \#1\}\{d\ \#2\}\ d\ \#3\}}\).
6.3. Specialized storage formats

<table>
<thead>
<tr>
<th>Description</th>
<th>Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math environment</td>
<td>math</td>
</tr>
<tr>
<td>Identifier</td>
<td>mi</td>
</tr>
<tr>
<td>Operator</td>
<td>mo</td>
</tr>
<tr>
<td>Number</td>
<td>mn</td>
</tr>
<tr>
<td>Displacement</td>
<td>msub, msup, msubsup</td>
</tr>
<tr>
<td>Line</td>
<td>mrow</td>
</tr>
<tr>
<td>Under-over for summation</td>
<td>munderover</td>
</tr>
</tbody>
</table>

Table 6.5.: Supported MathML tags in MosaicMathML.

Figure 6.7.: MosaicMathML subscript environment: Row environments hold the involved lines.

Supported Elements The supported tags are shown in Table 6.5. The number of supported MathML entities has been used considerably, only the Greek letters, the center dot used to represent the multiplication and the summation sign are supported.

&Alpha; &Beta; ... &alpha; &beta; ...
&CenterDot; &middot; ... &sum;

Additional marking of displaced lines In order to facilitate access to displaced expressions, both the displaced and the undisplaced lines must be children of row elements, even if such lines contain only one element, cf. Figure 6.7. This rule applies to msub, msup, and msubsup.

Variable Namings All symbols used in the variable naming must be placed individually into an identifier environment. Likewise, all operators must be put into individual operator environments. The commas used for the separation of several indices and superscripts are also considered as operators, cf. Figure 6.8.

Enclosing operator environments In all enclosing operator environments, the operands are nested in row environments. In bracket expressions, for example, one row environment contains the bracket characters in operator tags (mo), while another row
6.3. Specialized storage formats

\[ x_{i,L}^{j} \quad x_{j,i-1} \quad x_{j=2,i=5} \]

\[ \begin{align*}
& <\text{msubsup}> \\
& \quad <\text{mrow}> \\
& \quad \quad <\text{mi}>x</\text{mi}> \\
& \quad \quad <\text{mi}>i</\text{mi}> \\
& \quad \quad <\text{mi}>I</\text{mi}> \\
& \quad \quad <\text{mi}>L</\text{mi}> \\
& \quad \quad <\text{mrow}> \\
& \quad \quad <\text{mo}>,\text{mo}</\text{mo}> \\
& \quad \quad <\text{mn}>1</\text{mn}> \\
& \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& \quad \quad <\text{mi}>j</\text{mi}> \\
& \quad \quad <\text{mi}>i</\text{mi}> \\
& \quad \quad <\text{mi}>I</\text{mi}> \\
& \quad \quad <\text{mi}>L</\text{mi}> \\
& \quad \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& \quad <\text{mrow}> \\
& <\text{mrow}> \\
& <\text{mn}>2</\text{mn}> \\
& <\text{mn}>5</\text{mn}> \\
& \end{align*} \]

Figure 6.8.: Examples of correct expressions for Variable Namings in MosaicMathML.

element between the brackets serves as the parent node for the contained algebraic expression, cf. Figure 6.9 left. The operators of the symbolic notation that include brackets, \(\exp()\), \(\ln()\), \(\sin()\), and \(\cos()\), are structured in the same way, cf. Figure 6.9 right; likewise, the operators using dedicated environments, such as \(\sqrt{\Box}\), use a row element as a container for each operand, cf. Figure 6.10.
6.3. Specialized storage formats

Figure 6.9.: Brackets and operators involving brackets in MosaicMathML.

\[
\begin{align*}
(AE) & \quad \ln(AE) \\
<mrow> & <mrow> \\
<mo>(</mo> & <mo>\ln</mo> \\
<mrow> & <mrow> \\
..AE.. & ..AE.. \\
</mrow> & <mrow> \\
<mo>)</mo> & <mo>)</mo> \\
</mrow> & <mrow> \\
</mrow> & <mrow>
\end{align*}
\]

Figure 6.10.: Fraction and root environments in MosaicMathML: Row environments wrap contained algebraic expressions.

\[
\begin{align*}
\frac{NUM}{DEN} & \quad \sqrt{AE} & \quad r\sqrt{AE} \\
<mfrac> & <msqrt> & <mroot> \\
<mrow> & <mrow> & <mrow> \\
..NUM.. & ..AE.. & ..RE.. \\
</mrow> & <mrow> & <mrow> \\
<mrow> & </mrow> & </mrow> \\
..DEN.. & </mrow> & </mrow> \\
</mrow> & </msqrt> & </mroot> \\
</mfrac> & </mrow> & </mrow> \\
</mrow> & </mrow> & </mrow>
\]
7. Modular modeling approach

7.1. Model elements

In the following, the model elements of MOSAIC will be presented in dedicated sections. Unless indicated otherwise, all presented model elements are stored in separate XML files and are reusable in different contexts. The application of a model element is generally realized by specifying a reference to its location, which is usually represented by a URL, but may also include database queries. Furthermore, most model elements contain a mandatory description in order to monitor assumptions, design decisions, and the intended purpose, which is essential for correct reuse.

The Notation is a model element adapted from model presentations in the literature where authors often provide a list of symbols explaining their meaning. In the literature, the presence of such lists, as well as their comprehensiveness, depend on the attitude of the author, who might accidentally or deliberately fail to mention certain symbols. In contrast, in MOSAIC the Notation is a mandatory model element. Every model element that introduces variable names into the problem description in a certain way must reference a Notation. Moreover, during the creation and the modification of model elements that reference a Notation, checks are made whether or not the symbols of the introduced variable names are described by the Notation. In this way, it is ensured that for a given MOSAIC model all contained variable names are explained by a Notation and a complete list of symbols can be generated for documentation purposes in the program code or in the documentation output. Furthermore, the documentary information can be accessed during modeling, which improves transparency and thus facilitates both the creation of models and the reuse of model parts.

The symbols used within the variable names may have several characters, which are restricted in their scope to Latin and Greek letters. Adhering to the general rule that the MOSAIC model elements are stored in the documentation-level format Presentation MathML, the Greek letters are stored in the form of MathML entities, whereas they are entered in the form of LaTeX code.

The UML representation of a MOSAIC Notation is provided in Figure 7.1. The model element consists of lists of four classes of variable elements: base symbols, subscripts, superscripts, and indices. All elements consist of a symbol and a description. For indices, a symbol for the maximum value must also be provided.

The Equation is the smallest model element containing mathematical content, which can either be one (single) algebraic equation or one (single) explicit ordinary differential equation. The mathematic expression is stored in MosaicMathML. The equation type
7.1. Model elements

Figure 7.1.: UML representation the Notation and the symbols used in the Variable Names. The operation `isValidSymbol()` represents the validation of the user-input.

(algebraic or first-order ordinary differential equation) is given implicitly by the mathematical expression. Each Equation is self-contained and can be used independently in several Equation Systems; the consequence of this independency of (re)use is that an Equation System and all contained Equations may have been created by different authors. To assist in correct usage, each Equation includes a description, in which the author can state the general purpose and the underlying assumptions. Furthermore, each Equation must reference a Notation that covers all symbols used in the mathematical expression. The descriptions provided by the referenced Notation can be displayed during the application of the Equation and they can be included in both the generated program code and the documentation output. On one hand the combined presence of a description of the whole Equation and the description of each used symbol on the other, help authors to provide transparency and thereby make the Equations better qualified to be reused correctly in different contexts. The UML representation of an Equation, including the above-mentioned properties, is shown in Figure 7.2. In order to give a complete reference the figure contains the optional use of a ParameterList, which will be explained later in this section.

It should be noted that the model element Equation does not include any interfaces specifying input or output variables, nor does it contain any explicit classification of its variables in ‘given’, ‘calculated’, ‘algebraic’, ‘state’ or ‘differential’ values, since defining such interfaces or classifications at this stage would restrict the reuse of Equations.

Figure 7.2.: UML representation of the Equation.
7.1. Model elements

The Equation System is in the most general sense a set of Equations created by connecting existing Equations and Equation Systems. From the perspective of an Equation System, the directly-connected Equations and Equation Systems are its Connected Elements, which can stem from different authors and can use different Notations. The question of how the coupling of the Equations is specified under these circumstances will be answered later in this section. In addition to the referenced Connected Elements, an Equation System must contain a Notation as well as a description, in which the purpose and the assumptions should be laid down, cf. Figure 7.3. Optional model elements of Equation Systems are Functions, Ports and Streams, which will be covered later.

Like Equations, Equation System do not have interfaces or explicit variable classifications. The questions of how such information is added and how Equation Systems are used as a modular part of full problem specifications will be answered later.

Variables and Variable Namings As stated above, the Equations and Equation Systems are stored in separate files and can be used in different contexts and thus in different combinations. An important question in that respect can be focused as follows: How is the coupling of any two Equations defined when the only information contained is the mathematical expressions and the referenced Notation? In order to answer this question, the concepts used to distinguish variable names in MOSAIC must be explained. Before the mechanisms proposed in this work are described, the way of solving this problem in conventional programming should be remembered. In most general-purpose programming languages, as well as in equation-oriented languages, the general rule is that identifiers bearing the same name are considered as representing the same variable as long as they appear in the same scope (i.e. in the same method, sub model, module, etc.). This intuitive rule is also used in MOSAIC. However, since the Equations are defined independently, new concepts must be introduced to represent a scope, and also to test the validity of a combination of Equations.

In the above common rule, the variables are distinguished by their names. Accordingly, the MOSAIC modeling concept provides the model elements Variable and Variable Naming. However, since in the general reuse situation several Notations are involved, and since the assumption that the same Variable Naming does represent the same Variable cannot hold if the compared Variable Namings belong to different Notations, a
translation of some or all Variable Namings of an Equation from one Notation to another is necessary in many cases in order to ensure a meaningful comparison. In such translations, the original Variable Naming is stored as a synonym, whereas the new Variable Naming is subsequently considered to be the valid distinguishing property of the Variable concerned. Attention should be drawn to the fact that, due to the recursive character of the reuse of Equation Systems (cf. Figure 7.3), several translations can take place consecutively and thus several synonyms may exist for each Variable Naming. By representing the model variables in the above way, the mathematical information and the displayed representation are separated.

In MOSAIC, a Variable is characterized by its value, its classification (‘algebraic’, ‘state’, ‘differential’, etc.), and its unique index. Furthermore, a Variable contains exactly one distinguishing Variable Naming, which will be called Top-Level Naming in the following, and it may contain one or several synonymous Variable Namings, which will be called Synonyms cf. Figure 7.4. In order to maintain a link between the Variables and the mathematical expressions, each Variable contains pointers to each of its occurrences in the mathematical expressions of an Equation System. Figure 7.4 also contains the basic attributes and operations of the Variable Naming. The operations ‘equals()’ and ‘hashCode()’ make the objects of this class distinguishable in a meaningful way.

![Figure 7.4.: UML representation of Variable and Variable Naming.](image)

The UML diagram shown in Figure 7.5 takes the above-mentioned translation of Variable Namings into account. The involved model element Connector will be explained next, and the concepts for using Equations and Equation Systems in different contexts will be discussed in section 7.2.1. The latter section also includes the concept of naming policies, which are present in Figure 7.5 as an attribute of the class ConnectionProperties.

![Figure 7.5.: UML representation of the connection of Equations or Equation Systems to an Equation System.](image)
7.1. Model elements

The Connector is used to specify the translation of Variable Namings, cf. Figure 7.5. Such translations are considered to be directed from the ‘sub element’ to the ‘super element’. The requirement of reusability also applies to the Connector, which is consequently defined independently from Equations or Equation Systems. In order to establish a characteristic for the correct usage of the Connector, both the Sub Notation and the Super Notation involved must be referenced. A Connector contains one or several Matching Entries, each of which define a Sub Variable Naming and a Super Variable Naming and may contain several Index Matching Entries, if the involved Variable Namings have more than one index. The UML class diagram of the Connector is shown in Figure 7.6.

![UML representation of the Connector.](image)

The Interface is a model element, through which the user can define fixed sets of Variable Namings that should be used when two model elements are interconnected. Interfaces are aimed at facilitating reuse by offering cooperating engineers the possibility to focus on a small set of Variable Namings instead of having to review entire models. Furthermore, Interfaces are used to close the gap between the general-purpose character of Equation Systems in MOSAIC and platform-dependent reflections of specific purpose functionalities such as calls to thermodynamic routines etc. Attention should be drawn to the point that the above-mentioned gap exists because the classification of Variables in MOSAIC is above all a numerical one, i.e. like in general numerical routines of Matlab or C++, engineering concepts are not included, even though such concepts must be perceived and checked by the user. In MOSAIC, the user is supported in the maintenance of engineering concepts mainly by documentary means, i.e. by providing means of describing the used Variables in a systematic way and giving the user access to the descriptions in a practical way. However, since the descriptions are provided by the user, they cannot be evaluated by code-generating software for special-purpose routines. Offering a selective way to control such special-purpose code generation if necessary, the Interface is the model element, through which the user can supply Variable Namings with predefined engineering concepts in order to enable code generation for engineering-specific software. The structure used to implement this functionality is described in the following:

The user can define several Field Entries in one Interface, where each Field Entry represents one Variable Naming combined with an additional set of attributes to reflect the engineering concepts. The attributes are a fieldName, which can be used as a label for code generation, a dimension to distinguish between scalar and vector-valued quantities, an engineeringUnit, and a direction to distinguish between input and output variables. The documentation-level concept of MOSAIC is supported by the
7.1. Model elements

mandatory reference to a Notation, which must describe all Variable Namings contained in the Interface, cf. Figure 7.7.

![UML representation of the Interface.](image)

**Functions and Function Applications.** Functions are useful when values need to be calculated and the introduction of additional variables or equations is not desired. Possible applications include the calculation of physical properties. Functions in MOSAIC have one output value and several input values. It is possible to equip both the output value and the input values with indices. Furthermore, Functions contain a mandatory description, which should be used to establish the proposed application and any underlying assumptions. As with Equations, the mathematic expression of a Function is stored in MosaicMathML. The UML diagram of a Function is shown in Figure 7.8.

![UML representation of the Function in the general case.](image)

In the following the application of Functions in MOSAIC is motivated and explained. At first, the traditional way of applying functions will be presented, in the course of which some drawbacks will be pointed out, which are of a structural nature and do not depend on the usage of a two-dimensional or a one-dimensional representation.

Consider the function defined in (7.1) and (7.2), which represents the Antoine equation for the calculation of the vapor pressure of a compound.

\[ f_{PLV} : \mathbb{R}^+ \times \mathbb{R}^3 \mapsto \mathbb{R}^+ \]  

\[ P = f_{PLV}(T, A, B, C) = 10(A - \frac{B}{C + T}) \]

A very common way to apply functions is to assign the input variables in the same order as their conceptual counterparts in the function interface. Therefore, within a model containing the vapor pressure \( P_{i=1}^{PLV} \), the temperature \( T_{sys} \), and the parameters
A_{i=1}^{plv}, B_{i=1}^{plv}, C_{i=1}^{plv}, a typical documentation-level use, as well as a programming language-like application (or function call) would look similar to

\[ P_{i=1}^{LV} = f_{PLV} \left( T_{sys}, A_{i=1}^{plv}, B_{i=1}^{plv}, C_{i=1}^{plv} \right) \] (7.3)

This way of using functions is unambiguous, but it comes with a possible source of errors, since the function definition (7.2) usually appears at a different place in the model description or program code than the function call (7.3), and although experienced modelers will testify that this is usually not a problem, they might also admit having dealt with faulty applications like

\[ P_{i=1}^{LV} = f_{PLV} \left( T_{sys}, C_{i=1}^{plv}, B_{i=1}^{plv}, A_{i=1}^{plv} \right) \] (7.4)

Furthermore, in order to fully specify the represented quantities, the variables contained in the function definition (7.2) must either be associated with sufficiently qualified types, or they must be explained by descriptive elements. In this context, a qualified type represents an engineering quantity including its unit, as well as its minimum and maximum values. Since the capacities of typing vary considerably between modeling languages, qualified types are not considered in this work. The existence of descriptive elements, on the other hand, is not ensured in most modeling languages, so the modeler is left to interpret the used naming concept and the function body, if he has access to it.

In order to gather experience with an alternative approach that avoids the above-mentioned drawbacks, the modeling concept for function definitions and applications proposed in this work is more detailed than necessary for an unambiguous model specification. The Function is applied by assigning the Variable Namings of the input Variables and the output Variable of the Function explicitly to the Namings of the corresponding Variables in the Equation System. Each usage defined in such a way represents a Function Application, which is an integral part of the Equation System, cf. Figure 7.9. During the process of assignment, the user has visual access to the Function body and to the Notation information of all Variable Namings used in the Function’s interface. Furthermore, the visualization of an Equation System always includes the presentation of each used Function together with the corresponding Applications, so that correct usage can be verified at any time. Such visualizations include both the model presentations within the software, and the presentations in the documentation output. Therefore, the use of a Function would be shown in a way similar to

\[ P = f_{PLV} \left( T, A, B, C \right) = 10^{(A^L - B^L)} \] (7.5)

\[
\begin{align*}
    P_{i=1}^{LV} & \rightarrow P \\
    T_{sys} & \rightarrow T \\
    A_{i=1}^{plv} & \rightarrow A \\
    B_{i=1}^{plv} & \rightarrow B \\
    C_{i=1}^{plv} & \rightarrow C
\end{align*}
\] (7.6)

In addition to the increase in transparency attained by presenting functions and corresponding applications side by side, a further advantage is that the description of the
interface variables is guaranteed, since it can at any time be determined from the elements of the corresponding Variable Namings together with the used Notation. Errors that result from passing arguments in the wrong order and thereby applying Variables in the wrong context are reduced in this way.

It should be noted that the definition of the mapping, eq. (7.1), is often unwritten and is not explicitly reflected in many modern programming languages, such as Matlab. This information is also contained only implicitly in MOSAIC.

In order to provide a further means of standardization, the model element Interface can be used to specify the input and output Variable Namings of a Function. The corresponding UML diagram is given in Figure 7.10. Notably, the advanced capacities of Interfaces can be exploited, in order to address special-purpose code generation, e.g. for platform-dependent thermodynamic property function calls or invocations of CAPE-OPEN property packages, cf. section sec:code.gen.specific.interfaces.

Figure 7.9.: UML representation of a Function Application and the context in which it is used.
Units, Ports, and Streams. Flowsheeting is an important modeling technique in process systems engineering, and it is therefore reflected in the MOSAIC modeling systematic. In order to make an Equation System represent a process unit, it must be equipped with Ports. Several Units created in this way can be connected through Streams.

A Port consists of a uniqueName and of a reference to an Interface, which specifies the Port’s quantities through Variable Namings, and a reference to a Connector, which defines the translation of the Variable Namings from the Equation System to the Interface, cf. Figure 7.11.

The definition of a Stream is given by an identification number (uniqueId), a reference to an Interface, and two Port Entries, each of which specifies a Unit and a corresponding Port. If the Port and Stream Interfaces differ, the Port Entry must reference a Connector governing the correct translation of the Variable Namings involved, cf. Figure 7.12. As far as the aspect of modularization is concerned, Ports and Streams do not represent independent model elements; they are defined as part of the Unit or Process, respectively.
This modular setup is highly versatile, since it places the definition of quantities within Ports and Streams in the user's hands, instead of predefining such sets. The user defines, reuses, and shares types of Ports and Streams by creating appropriate Interfaces. It should be noted that this concept allows the user to define different kinds of connections including material streams and data streams. The user decides for the properties of the model's ports and streams by defining applicable variable names in the Interface.

**The Parameter List** can be used to mark certain Variable Namings within the Equation System in order to treat them as global parameters. This model element consists of a description and a list of Variable Namings, which must be described by the referenced Notation, cf. Figure 7.13.

Whenever a Connected Element contains a Parameter List, no synonyms to the specified Variable Namings are searched for. During the evaluation of the Equation System, all global parameters declared in this way are presented in a separate list and can be stored independently, see Figure 7.16.
The Evaluation is the model element, in which a simulation problem is formulated based on an Equation System. The pieces of necessary additional information are described in the following:

**Indexing.** As described in section 6.1, Equations and therefore Equation Systems may contain generic indices. When a generic Equation System is used as part of a numerical problem, values must be assigned to the maximum-value symbols of the generic indices, cf. Figure 7.14. The Indexing is an important part of the problem specification, since it defines the dimension of the instantiated Equation System. In practice, the maximum values represent characteristics of the model, e.g. the number of column plates, the number of compounds in a mixture, etc.

![Figure 7.14.](image)

**Variable Specification.** After the Indexing is provided and the Equation System is instantiated, an actual set of Equations and Variables exists. The user decides which Variables should be fixed (i.e. the design Variables) and which Variables should be calculated (i.e. the state or iteration Variables, depending on the type of problem). After this classification, the user gives values to the Variables. Both the classification and the values must be chosen carefully in order to maintain a consistent numerical problem. The corresponding information is stored in a separate model element labeled Variable Specification, which is a list containing the classification, the value, and the Top-Level Naming for each Variable cf. Figure 7.15.

![Figure 7.15.](image)
7.1. Model elements

Language specification and code generation. After the above information is specified, the code generation of MOSAIC must be initiated in order to generate program code that is able to solve the numerical problem. It is possible to choose between several Language Specifiers, which define the output for numerical libraries or dedicated environments. Some of the Language Specifiers define program code that can be evaluated on the MOSAIC modeling server. In this way, MOSAIC can be employed as a stand-alone modeling environment, even though the core feature of the tool is the versatile translation of the described numerical problems into different languages for the use in external applications.

Reuse of evaluation results. The results of a numerical calculation executed on the modeling server are stored in the model element Result Specification. As is illustrated in Figure 7.15, Variable Specification and Result Specification are of the same type, and obtained results can therefore be reused easily as Variable Specification in order to define problems based on existing solutions.

The Evaluation. Representing the assembly of a complete numerical problem from modular model elements, the Evaluation contains references to an Equation System and a Variable Specification. The attribute Indexing and the reference to a Parameter Specification both depend on the used Equation System. The complete UML diagram of an Evaluation is depicted in Figure 7.16.

![UML representation of the Evaluation and the related model elements.](image)
7.2. Modular reuse concepts

7.2.1. Connection Techniques

In this section, the techniques of the combined use and reuse of the model elements Equation, Equation System, Connector and Notation will be introduced, and the representation of flowsheeting concepts will be explained. When describing the proposed reuse mechanisms, the involved model parts will be addressed in terms of ‘sub’ and ‘super’ elements as follows:

In the left part of Figure 7.17, Equation 1 and Equation 2 both use Notation I and are connected to Equation System A, which uses Notation II. In this context Notation I and Notation II are called sub Notation and super Notation respectively, whereas Equation 1 and Equation 2 are sub Connected Elements and Equation System A is the super Connected Element. Furthermore it will be necessary to consider the ‘origin’ of a Variable Naming, which corresponds to its scope in the sub Connected Element; this information is stored in the Variable Naming’s property Name Space.

Figure 7.17.: Reuse: Creation of Equation Systems based on existing Equations and Equation Systems.

The different situations in which reuse may occur can be categorized into four cases, which are shown in Figure 7.18 and described in the following:

Case 1. The first kind of application discussed here is the case, in which a modeler has created several Equations with the intention to use them together in an Equation System. Since the Equations and the yet to be created Equation System share the same engineering context, the same Notation is used for all Equations, as well as for the Equation System. When the Variable Namings of the different Equations are compared to each other, it is preferable that their origin is not taken into account, i.e. if there is a Variable Naming \( x \) in Equation 1 and in Equation 2 both are considered to represent the same Variable.

Case 2. Now let us assume that the modeler wants to use an Equation that has been created by a different institute, and in which another Notation is used. The modeler decides that all Variables should be visualized in the same Notation, so that all Variable Namings of the reused Equation need to be translated to fit the Notation the modeler uses in his Equation System.
Case 3. In another project, the modeler has obtained access to an entire Equation System created by a cooperating institute. In the provided Equation System all Variable Namings have the same scope but they refer to a different Notation than the one used in his project. The modeler could have all Variable Namings translated, as in the former case, but he decides that he wants to use the Equation System as it is. To avoid any accidental coupling of the Equation Systems, the Name Space of the Variables is taken into account when comparing the Variable Namings. It is necessary, however, to translate the desired coupling Variables of the sub Equation System into the corresponding names in the super Equation System.

Case 4. Finally, the modeler wants to reuse two of his Equation Systems that both use the same Notation. In this case, considering the identical Variable Namings of different origins as representing one Variable is not desirable, since this would result in unwanted coupling. The modeler chooses a behavior in which the Name Space of the Variable Namings is taken into account even though the Notations do not differ. The Variable Namings of the coupling Variables must be translated formally.

The above four cases are depicted in Figure 7.18 and can be dealt with by introducing two Naming Policies:

Figure 7.18.: Four cases of reusing Equations and Equation Systems. TA stands for ‘Translate All Variable Namings’, TC stands for ‘Translate the Coupling Variable Namings’.
7.2. Modular reuse concepts

- **Naming policy ‘integrate’**: The resulting Equation System uses the same Notation for all Variable Namings. If a Connected Element (Equation or Equation System) uses a different Notation, then all of its Variable Namings must be translated to the super Notation. When Variable Namings that correspond to the same Notation are compared, their Name Space is not considered to be a distinguishing property. Cases 1 and 2 in Figure 7.18 fall into this category.

- **Naming policy ‘encapsulate’**: The Name Space of the Variable Namings is used as an additional distinguishing property. The Variable Namings of the coupling Variables must be translated from the sub Notation to the super Notation. Cases 3 and 4 in Figure 7.18 are examples of this naming policy.

A detailed description of the connection techniques is provided in section 12.
8. Generation of program code and documentation

8.1. Introduction

The models specified as described in sections 6 and 7 are transferred into different types of program code. The application of a general, implementation-independent model description as a basis for code generation is a technique used in model-driven software engineering. An introduction to that field is given in section 3.2.2, a deeper insight is provided in [Fra03, RFW+04] and [MM03]. This section focuses on the stepwise transformation of the models and the specification of the generated output. It is good practice to use a meta model stored in XML, that contains the content to be translated and to use XSLT for any transformations, cf. [W3C10]. However, the documentation level is the focused of this work and therefore, the model is stored in Presentation MathML, which is not a content format. Accordingly, the transformations are not specified in XSLT but in Java and in XML. This allows for more flexibility during development and reduces the number of technologies applied, which is a relevant aspect with respect to long-term maintenance in a situation with constantly changing developers who are generally not computer scientists.

8.2. General applied principles

8.2.1. The involved models

The focus of this section is the generation of program code based on models that are defined in the documentation level. In section 3.3, it was stated that such an approach involves two data models. The respective data models for the documentation and the implementation level will be introduced next. In order to focus on the basic concepts, functions are excluded from the explanations.

The approach presented uses the basic concept of model-driven architecture, cf. [MM03]. The general principle is the transformation of a platform-independent model (PIM) into a platform-specific model (PSM) with the help of a platform model. In the application presented in this work, the platform-specific model corresponds to the complete program code representing the mathematical model, whereas the platform corresponds to the environment that runs this code.\footnote{The environment can be a dedicated software tool, a runtime environment, or an operating system running a compiled version of the code.} The platform-independent model is present on the documentation level and contains more information than is needed for the speci-
8.2. General applied principles

Figure 8.1.: The transformation process and the involved models.

The documentation-level data model

The documentation-level data model used in MOSAIC was introduced in section 7.1. The documentation-level model obtained after parsing a full numerical problem is shown in Figure 8.2. The documentation-level data model is intended for the creation of unambiguous mathematical expressions that are also informative in the presentation aspect. Therefore, the documentation-level data model contains more information than is needed for implementation. In the case of the MOSAIC documentation-level data model, the surplus information can be found in the complexity of variable namings and in the fact that several such namings exist for one variable. Likewise, the encapsulation structure, cf. Figure 7.3, is stored in the documentation-level model but not necessary in implementation models.

The implementation-level data model

As described above, the data model of the implementation level is less complex and more general than that of the documentation level. On the implementation level, a numerical problem based on equation systems consists of a set of equations, variables, and parameters. In the case of steady-state problems, i.e. root finding problems based on non-linear algebraic equation systems (NLE’s), a numerical problem is composed of equations, algebraic variables, and parameters. The class algebraic variable is a subclass of variable, which contains a name in the form of a string, a value and bounds in the form of a real number, and an index number in the form of a natural number. The algebraic equation is

2Please note that the classes ‘variable’ and ‘equation’ are contained in the data models of both levels. The elements of the implementation level are qualified by their package ‘transfer’. 
8.2. General applied principles

Figure 8.2.: The documentation-level model: Relations between Equation Systems, the contained Variables and the corresponding MathML expressions.

Figure 8.3.: The implementation-level data model of NLE problems.

In the case of NLE problems, the value of the variables represents the guess value for the numerical calculation. The types of the attributes ‘value’, ‘lower bound’, and ‘upper bound’ of the class variable are represented here as ‘real’ numbers to emphasize that the actual data type (e.g. double, float, etc.) depends on the platform, i.e. on the target programming language. In the implementation-level data model, variables and equations have an index number so that a specific order can be considered during code generation.

In the case of dynamic problems, i.e. problems based on differential-algebraic equation systems (DAE’s), the implementation-level data model is composed of differential and
algebraic equations, explicit and implicit state variables, parameters, and the independent variable of the differential subproblem. Both the explicit and the implicit state variable are subclasses of variable, and the differential and algebraic equations are subclasses of equation, cf. Figure 8.4. It should be noted that the explicit and implicit state variables are associated with the differential and the algebraic equations, respectively. In the case of DAE problems, the value of the state variable represents the initial value of the numerical integration. The content tree contained in the class equation represents a non-linear algebraic expression in a parsed form. The UML representation of a content tree is shown in Figure 8.5, an example was presented in Figure 3.15 (b), and the principles of its creation were discussed in section 3.4.3. The advantage of such a representation is that the specification of comparatively small translation rules for each contained node type is sufficient to generate program code for any non-linear algebraic expression.

**Significance** The presented implementation-level data model is implicit, i.e. it has not been implemented in actual Java classes in MOSAIC. The data model is shown here, because it represents the assumptions made about the supported programming languages, and because it contains exactly the information that is mapped during code generation. It was found that all numerical environments investigated within the scope of this work use the above information, in other words, in all considered environments, the presented information is applicable for the formulation of the given numerical problems. In this sense, the introduced data model represents the most general type of model. Other aspects exist, such as the representation of functions, the invocation of external routines based on standardized interfaces, and the exchange between generated models and flowsheeting environments, cf. section 8.3.
The platform model

The platform model is a description of the generated program code. With respect to the introduced implementation-level model, a platform model contains information concerning the representation of operations, numbers, and variables. Furthermore, it describes how equations are represented based on algebraic expressions, how the values are assigned to the variables before the numerical evaluation, etc. Finally, the structure of the program is represented, which is defined by the placement of the code fragments specified previously, such as blocks of equations, variable specifications, etc. Thereby the entire problem formulation can be represented in two transformation steps. In MOSAIC, the basic platform models including transfer rules for operations, numbers, and variables are contained in the class LanguageSpecificator. Interfaces dedicated to the different types of the equation system contain methods for the definition of the respective program structure, cf. Figure 8.14 on page 98.

Steps of code generation

As described above, the code generation based on the model documentation is effected through the transformation of the documentation-level model via the implementation-level model to program code.

Obtaining the implementation-level model  From the perspective of model-driven software engineering, the implementation-level model corresponds to the platform-independent model for problem-solving code. In MOSAIC, the implementation-level model is created implicitly by ignoring or replacing information in the documentation-level model. As described in section 6.3.2, the MathML expressions of the documentation-level model are used throughout the modeling process for both the display and the content representation of the algebraic expressions. In order to obtain an equivalent of the content tree for the algebraic expressions, cf. Figure 8.5 and to prepare code generation, several operations are performed on the MathML expressions. The goal of these transformations is to obtain the generalized structure of equations on the implementation level, based on the more complex differential and algebraic equations containing...
8.2. General applied principles

variable namings on the documentation-level. If the equations of the documentation-level model are identified as algebraic equations, then the original MathML expression containing $l(x) = r(x)$ is transformed into one algebraic expression $l(x) - r(x)$. In the case of ordinary differential equations, the MathML expression containing $\frac{d}{dx} x_i = a(x)$ is transformed into $a(x)$ and the association $(x_i, a(x))$ is registered. Subsequently, the MathML structures representing the variable namings are replaced by unique string identifiers, and a table that contains a mapping from these identifiers to the corresponding variables is created. In this way, all information in the variables are accessible and associated with the algebraic expressions $l_i$, $r_i$, and $a_i$. Consequently, the distinction between independent variable, state variable, algebraic variable, and parameter can be made for each identifier.

Obtaining the program code As described in the previous section, the generation of the platform-specific model is performed successively in two steps. First, the code for the algebraic expressions is generated in a recursive method that traverses the content tree. The result is a string representing the correct algebraic expression in the target programming language for each equation. Second, the complete code structure is generated. This step includes the placement of the obtained algebraic expression strings within the generated code. Depending on the target language, i.e. depending on the implementation of the language specificator, this step includes the creation of code fragments for the declaration and initialization of variables, for passing variables and parameters through method interfaces, etc. The information necessary for such code fragments is obtained during the first step: Whenever a variable is encountered in the content tree, it is translated and registered in the language specificator. Depending on the desired program structure, the corresponding code fragments, cf. Figure 8.6, can be used more than once, cf. Figure 8.19 on page 103.
8.2. General applied principles

<table>
<thead>
<tr>
<th>Language/Platform</th>
<th>NLE</th>
<th>ODE</th>
<th>DAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++/BzzMath</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>C/GSL</td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matlab</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Scilab</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Aspen Custom Modeler</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>GAMS</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML Spreadsheet/MS Excel, OO Calc</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>gPROMS(^\ast)</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>C++/sDACL, [BKWAG11](^\ast)</td>
<td>S</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1.: Currently implemented platform models (Language Specificators), cf. also [KBK+11]. ‘S’: executable on the numerical server, ‘L’: local usage, (*): not implemented within the responsibility of the author.

### 8.2.2. The software structure and extensibility

The general structure of the code generation is simple. A code generator translates an equation system with the help of a platform model, which is labeled LanguageSpecificator, cf. Figure 8.7. As described above, the equation system is initially present as a documentation-level model and in an intermediate step translated into a implementation-level model. Since a main feature of MOSAIC is the generation of program code into many languages, the extensibility of the software through new platform models is an important subject. In order to profit from the similarities of the output code fragments, an inheritance tree has been developed, which shifts the focus of the non-abstract platform models to the creation of the parts of the code that are characteristic for that language. In this sense, the translation of variables, numbers, and algebraic expressions is defined in classes near the root of the inheritance tree. Platform models for the same language are represented by subtrees so that common code fragments are defined in the corresponding subtree root, cf. Figure 8.15 on page 99. The currently implemented platform models are listed in Table 8.1.

Figure 8.7.: Model elements involved in code generation; the basic situation.
8.3. Representation of problem-specific interfaces

Motivation

The calculation of physical properties is a research field of its own and many models, approaches, and corresponding calculation routines exist. Not all approaches are described by algebraic equations and functions, so specialized algorithms must often be used. Specialized software exists for this purpose and applying such software is generally more efficient than defining the corresponding models within equation systems. The problem of invoking thermodynamic software from within equation system models can be solved by representing problem-specific interfaces in the corresponding platform-independent models. Ensuring platform independence, such an approach focuses on the type of problem that needs to be solved externally instead of establishing a fixed link to a specific software. If the represented interfaces follow a standard, e.g. CAPE-OPEN, on the platform level, then it is also easy to choose between different thermodynamic approaches.

Many flowsheeting tools offer the definition of custom unit models and the support of code generation for such units is desirable. In this sense, flowsheeting software represents a special type of platform, which includes the exchange of data via the invocation of thermodynamic routines and via ports defining points of interconnection with other units. Through the platform-independent representation of interfaces, both of these types of data exchange can be realized.

The complexity arising through the model-driven approach

The representation of problem-specific interfaces within the model-driven approach followed in this work results in a drastic increase of complexity. For each specific interface represented within the platform-independent data model, a corresponding platform model must be defined for each supported programming language. However, the support of the problem-specific calculation or data exchange cannot be assumed for all platforms and therefore a definition of a platform model does not always make sense. For a given set of specific interfaces and platforms, a support matrix can be visualized, as is shown in Figure 8.8. It should be noted that neither the actual set of specific interfaces nor the set of platforms are fixed. Both sets are growing with the maturity of the software and the development of new models and routines. In order to meet this demand of extensibility and to separate concerns, both the platform-independent definition of the specific interfaces, and the corresponding platform models must be implemented as modular software elements. Since the restriction of the concept to a fixed set of external tools is not required, the support of problem-specific interfaces has a heterogeneous character. Accordingly, the basic assumption must be that a specific interface is not supported in all platforms. Therefore, the implementation of platform-specific models is separated from the existing platform models. In this way, the platform models of specific interfaces can be added selectively for a set of supported languages without influencing the existing platform models of other languages.
The characteristic elements of the platform-independent definition of a problem-specific interface can be summarized as follows: The quantities that are represented by the exchanged data must be specified. For each quantity it is necessary to state an identifier for the underlying concept, the engineering unit, the dimension, and the direction of the exchange, cf. Figure 8.9. In this sense, the concept is a specification of the physical meaning of the quantity, e.g. ‘activity coefficient’. In this work, the dimension of a quantity is assumed to be either ‘scalar’ or ‘vectorial’; the possible directions are assumed to be ‘in’ or ‘out’. An example of a specific interface definition is given in Figure 8.10.

In section 8.3, the use of specific interfaces has been narrowed down to function calls and data exchange via ports. Since the existing model element Interface is a means of standardization within MOSAIC that can be referenced
8.3. Representation of problem-specific interfaces

Figure 8.10.: Characteristic elements of a specific interface for the calculation of the activity coefficient.

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Concept</th>
<th>Engineering unit</th>
<th>Dimension</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Activity coeff.</td>
<td>-</td>
<td>vector</td>
<td>out</td>
</tr>
<tr>
<td>2</td>
<td>Mole fraction</td>
<td>mol/mol</td>
<td>vector</td>
<td>in</td>
</tr>
<tr>
<td>3</td>
<td>Temperature</td>
<td>K</td>
<td>scalar</td>
<td>in</td>
</tr>
<tr>
<td>3</td>
<td>Pressure</td>
<td>Pa</td>
<td>scalar</td>
<td>in</td>
</tr>
</tbody>
</table>

in both Functions and ports of Units, cf. sections 7.1 and 7.2.1, the documentation-level models of specific interfaces are represented by this model element.

Coordination of the models  As in the general code generation described above, three types of models must be synchronized. In the case of specific interfaces, the implementation-level data model must be defined explicitly, since it serves as a reference for both the documentation-level model and the platform model. Each specific interface is defined by its own implementation-level data model. The link between documentation-level models and implementation-level data models is shown in Figure 8.16 on page 100. The documentation-level models are represented by the model element Interface, whereas the implementation-level data models are of the type InterfaceSpecification. A synchronization of concrete Interfaces and InterfaceSpecifications is possible, since both classes implement the interface SpecificInterface, which provides access to the characteristic elements of specific interfaces as described above, cf. Figure 8.9 and 8.10, where the ‘concept’ is represented through ‘field name’. In practice, the link between both types of models is executed as follows: During the creation of an Interface, the user can access the InterfaceSpecification describing the desired external function call. The characteristic elements, such as engineering units etc., of the InterfaceSpecification are visualized. The user can assign the contained concepts to VariableNamings and test whether all specifications of the Interface correspond to those imposed by the InterfaceSpecification, cf. Figure 8.11.

The definition of the platform models, as well as the synchronization between the implementation-level data model and platform models, is shown in Figure 8.17 on page 101. In that figure, the concrete platform models are FS_MatlabCOLiquidActCoeff, FS_MatlabCOLiquidEnthalpy, etc. All possible platform models are subclasses of SpecificInterfaceLangSpec, through which they have three important connections to other classes: First, each SpecificInterfaceLangSpec references exactly one InterfaceSpecification, which constitutes the link to the implementation-level data model. Second, the inheriting classes of SpecificInterfaceLangSpec that are defined for the same platform reference a SpecificEnvironmentLangSpec, which contains code specifications for the import, the initialization, etc., of eventually used external libraries. Third, each SpecificInterfaceLangSpec is assigned to a LanguageSpecificator, which represents the platform model of the entire program code. This assignment activates the support of the InterfaceSpecification on that platform.
8.3. Representation of problem-specific interfaces

Figure 8.11.: Creation of an Interface for the calculation of the activity coefficient based on an InterfaceSpecification.

**Extensibility** The implemented structure is easily extensible, since the programmers can focus on the platform models, which are directly concerned with the program code to be generated. The decision to extend the support of external interface invocation to a certain platform, implies that the platform is used, and that there is substantial expertise in the corresponding programming language. The extension of the support has two basic directions: the adding of implementation-level data models and platform models, cf. Figure 8.8. The creation of new implementation-level data models merely corresponds to listing the properties to be imposed. The effort involved in the creation of the platform models depends on the output language, but a certain familiarity with the language to be generated can be assumed, see above. The registration of the implementation-level data models and the platform models within the structure corresponds to a small number of function calls; in that step the existing classes can serve as templates. This registration is sufficient to activate the new models including all functionalities such as consistency and support tests.
8.4. User-defined language specification

Since modeling languages are under constant development, it is desirable to react quickly to changes in languages by modifying the corresponding platform models. For the same reason it is desirable to be able to add platform models for new languages. The specification of the platform model has therefore been declared a user-defined model element, which is stored in XML and created, reused, and shared in the same way as the other MOSAIC model elements (such as Notation, Equation, etc.).

As described in section 8.2.1, the transformation of program code is performed in two steps. In the first step, the algebraic expressions are translated and the variables are registered, whereas the second step is concerned with the creation of the complete code fragment. In this section the definitions of the platform model parts for both steps are discussed separately.

8.4.1. Variables and non-linear algebraic expressions

As described above, the translation of algebraic expressions can be subdivided into three major tasks: The translation of variables, numbers, and operators. The corresponding definitions of the platform model will be discussed in the following.

Variables

The Variable objects used in MOSAIC have two distinguishing properties: The top-level Variable Naming and the unique identification number, which is an index starting at zero. Either of these distinguishing properties can be used in the creation of distinguishable program code identifiers. If the identification number is used, then the user can assign a prefix and suffix string that will be added to the number’s string representation. If the top-level Variable Naming is used, then all symbols of the Variable Naming, including the index values, will be present in the generated program code identifier and the user can specify prefixes, suffixes, and infixes.

Numbers

For the specification of numbers, the user can assign a prefix and a suffix. The corresponding adaption to the output programming language includes the casting of numbers, e.g. by defining the suffix as ’D0’ in Fortran. Furthermore it is possible to force the program to represent the numbers as floating point numbers, so that e.g. the number 17 is translated to ’17.0’.

Algebraic operations

The platform model of the algebraic operations is the specification of the strings that separate and enclose the operands. Therefore, for an operation taking \( N \) operands, \( N + 1 \) strings must be specified by the user. In the used implementation-level data model, the maximum number of operands is two, which leads to the definition of at most three strings, a prefix, an infix, and a suffix. An example for platform model definitions for
Table 8.2.: Example of platform model definitions of the fix elements for the operation ‘plus’.

<table>
<thead>
<tr>
<th>prefix</th>
<th>infix</th>
<th>suffix</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘‘</td>
<td>‘+’</td>
<td>‘‘</td>
<td>‘LEFT+RIGHT’</td>
</tr>
<tr>
<td>‘add(‘</td>
<td>‘;’</td>
<td>‘)’</td>
<td>‘add(LEFT,RIGHT)’</td>
</tr>
<tr>
<td>‘&lt;apply&gt;’</td>
<td>‘ ’</td>
<td>‘&gt;’</td>
<td>‘&lt;apply&gt;’</td>
</tr>
<tr>
<td><code>&lt;plus/&gt;</code></td>
<td>‘ ’</td>
<td><code>&lt;/apply&gt;</code></td>
<td>&lt;plus/&gt;</td>
</tr>
<tr>
<td></td>
<td>LEFT</td>
<td>RIGHT</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;&lt;/apply&gt;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The addition is given in Table 8.2. The order of the operands in the platform model is assumed to be the same as in the implementation-level data model.3

8.4.2. Definition of the complete code fragment

Complexity of the problem

The complete generated program code consists of blocks of predictable content in an unpredictable order. For example, it is certain that the code contains the translation of the equations and it can be assumed that a variable initialization is present. However, no general assumptions can be made about the placement of these sections within the code. It can be assumed that the variable initialization has the same structure for every variable and that the generated variable names and the variable values are part of that structure, but nothing can be assumed about the representation order of these pieces of information; depending on the target programming language, the equation section can be part of a separate method, dedicated section, etc., cf. Figure 8.19 on page 103. These examples indicate that the explicit consideration of the possible program code structures would mean reflecting all different concepts of existing programming and modeling languages in the user-defined platform specification. However, such an approach must be avoided, since it would lead to an increase in the complexity of the complete implementation-level data model with each added platform model.

The implemented approach

In order to avoid restrictions with respect to extensibility, only a small set of assumptions about the programming language have been made, so that an explicit representation of the program structure is avoided. The implemented approach allows the user to access the code fragments generated in the first translation step, cf. sec. 8.4.1. All elements of the implementation-level data model can be represented within text blocks with the help of place holders, which are referred to as ‘handles’ in the following. By these means,

3This limitation of the approach that could be remedied easily by implementing a switch functionality for the operands. For the languages considered at the time of writing, such a switch of operands was not necessary.
suitable building blocks can be created in a free manner. The created building blocks can be nested so that a representation of complex program structures is possible. In order to avoid conflicts with commands and key words in the described languages, all handles can be renamed if necessary.

An example for a subdivision of a program code into blocks is shown in Figure 8.20 on page 104. The basic properties of the class Block are shown in Figure 8.12. Each block contains a specification that is the actual text of the represented code fragment. The specification of a block can be modified by the contained subblocks. All handles of subblocks found in the specification of a block are replaced with the specification of the subblocks. In the same way, the following elements can be introduced through handles: code fragments generated in the first step described above, i.e. variable names, variable values, algebraic expressions, etc., as well as case-dependent counters such as the number of equations. In addition to blocks that are substituted once per handle occurrence, there are blocks that are substituted several times. Such blocks are labeled LoopBlock and are used to represent reappearing structures, such as the individual variable specifications, parameter declarations, equation specifications, etc. The complete set of existing blocks is shown in Figure 8.18 on page 102. Further information about the implemented approach is given in [Buc11].

**Application by the user** The basic functioning of the implemented approach will now be explained in a step-by-step manner. First, the desired code to be generated should be created and debugged within the platform-specific environment. Second, the full code fragment is copied into the platform definition. Third, all problem-specific parts, such as algebraic expressions, variable instantiation, parameter definition, function calls, etc., are replaced with suitable generic code fragments, which are defined in subblocks based on the information obtained from the documentation-level model. In the definition of each of such subblocks, the copy of the respective original code fragment serves as the basis.

**Supported programming languages** Since the basis for the generated code is the textual output copied from the target platform, and since no restrictions are made on the placements and on the names of the handles that are used to substitute the case-dependent information, it can be stated that the presented approach makes it possible to define a platform model for all relevant programming languages.
8.5. Generation of documentation

8.5.1. Requirements and complexity

The generation of documentation is more complex than the generation of program code, since all information contained in the documentation-level model must be represented. The supplementary information includes the terminological notation, the two-dimensional symbolic representation of the algebraic expressions, the generic equations, connection structure of the equation systems, and all contained descriptions, cf. section 7.1.

8.5.2. The implemented approach

Structure  The task of transferring the information contained in the documentation-level model into representations of documentation-level formats has the same nature as the model-driven approach used for the program code generation described above. Accordingly, the implemented software part has a similar structure. A documentation generator uses platform models, which are labeled documentation format specificator, cf. Figure 8.13. In this application the platforms are different documentation-level formats.

Supported formats  The implementations realized for this work include the generation of plain text and LaTeX. Thanks to the model-driven character of the approach, an extension to other documentation-level formats is facilitated. However, since the high complexity of the documentation-level models results in complex platform models, the extension of the supported documentation output is subject to a high level of development effort. On the other hand, external software tools exist (e.g. [Hun09, Gri12]) that make the transformation between documentation-level formats possible. Therefore, it is justified to state that the production of such software does not lie within the scope of this work.

![Figure 8.13.: Structure used for the generation of documentation output.](image)
8.5. Generation of documentation

Figure 8.14.: Language Specificator, methods and interfaces.

```java
public class LanguageSpecificator {
    // Methods and interfaces
    // Implementation details
}
```

```java
// Interface LSEqSystemNLE

public interface LSEqSystemNLE {
    String createFullProblemSolvingCodeFragmentNLE(String rootFuncsCode, String directFuncImplsCode, String directFuncCallsCode);
}
```

```java
// Interface LSEqSystemODE

public interface LSEqSystemODE {
    String createFullProblemSolvingCodeFragmentODE(String rootFuncsCode, String directFuncImplsCode, String directFuncCallsCode);
    void setDifferentiaVariable(String var);
}
```

```java
// Interface LSEqSystemDAE

public interface LSEqSystemDAE {
    String createFullProblemSolvingCodeFragmentDAE(String rootFuncsCode, String directFuncImplsCode, String directFuncCallsCode);
    void registerExplicitStateVariable(String var);
}
```

```java
// Interface LSFunctionsByCall

public interface LSFunctionsByCall {
    void createDirectFunctionCall(String outputVar, String uniqueFuncName, String inputVars, String params);
    void createDirectFunctionImpl(String outputNmg, String uniqueFuncName, String inputNmgns, String calcExpression, String params);
}
```
Figure 8.15.: Language Specificator, heritage tree and realized interfaces. For a complete list of implemented Language Specificators refer to Table 8.1 on page 89.
Figure 8.16.: Classes involved in the definition of specific interfaces.
Figure 8.17.: Classes involved in the definition of special-purpose language specificators.
Figure 8.18: Heritage tree of the block classes involved in the user-defined language specification (platform model).
Variable sections

The code fragments generated in the first step are used in different ways.

General structure

The occurrence, the form, and the order of the marked code fragments depends on the language. A general assumption about the structure cannot be made.

Figure 8.19.: Generated Matlab code that is divided up in functions and therefore has a certain complexity.
8.5. Generation of documentation

Figure 8.20.: A possible separation of the given Matlab code in blocks. The blocks marked '*' are instances of LoopBlock, the blocks holding the three functions are instances of Block, which are held by an instance of FullCode.
Part III.

Examples
9. Examples

9.1. Transfer between documentation-level formats

In section 3.4 a methodology was introduced that permits the development of unambiguous symbolic notations for algebraic expressions. This methodology was applied to create the symbolic notation used in MOSAIC, cf. section 6.1. It was shown that the content of expressions in an unambiguous notation can be transferred between different storage formats without a loss of information. Therefore, the use of unambiguous symbolic notations makes it possible to define models in existing documentation-level formats and to use such models in dedicated numerical environments with little effort.

![Figure 9.1.: Possible translation steps with the implemented software.](image)

Figure 9.1 shows a graph of transfer steps between a number of documentation-level formats. In this example, three algebraic expressions were created in different storage formats using off-the-shelf software, cf. Table 9.1. Subsequently, the expressions were transferred in one or several steps using software tools that are not designed to interpret the used symbolic notation semantically, cf. Table 9.2. After such translation steps the expressions could successfully be transferred using the software MosaicLatex2MosaicMathML into the format that is used within the MOSAIC modeling environment. It should be noted that, of all software used in this example, only MosaicLatex2MosaicMathML interprets the symbolic notation, whereas the other transfer tools only consider the visual representation. The translation to program code of the expressions is demonstrated in section 9.4.

In the following, the creation of sample formulas and the path from each storage format to MosaicMathML is briefly described. Finally, the scanning and subsequent recognition of the sample formulas is discussed.
9.1. Transfer between documentation-level formats

<table>
<thead>
<tr>
<th>Start at</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Firemath [Bon12]</td>
</tr>
<tr>
<td>2</td>
<td>MathType [Des10]</td>
</tr>
<tr>
<td>5</td>
<td>Microsoft Word 2007</td>
</tr>
<tr>
<td>3</td>
<td>Standard user input in MOSAIC</td>
</tr>
</tbody>
</table>

Table 9.1.: Tools used for the creation of the expressions in the different documentation-level formats.

<table>
<thead>
<tr>
<th>Translation</th>
<th>Software</th>
<th>Symb. notation interpreted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>MathParser [WW04]</td>
<td>no</td>
</tr>
<tr>
<td>2-3</td>
<td>Latex2MosaicLatex(*)</td>
<td>no</td>
</tr>
<tr>
<td>5-3</td>
<td>Microsoft Word 2007 and Word2MosaicLatex(*)</td>
<td>no</td>
</tr>
<tr>
<td>3-4</td>
<td>MosaicLatex2MosaicMathML(*)</td>
<td>yes</td>
</tr>
<tr>
<td>4-1</td>
<td>- no translation necessary -</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 9.2.: Tools used for the transfer between documentation-level formats.

(*): part of MOSAIC.

Figure 9.2.: Example expressions created with the mathematical editor in Microsoft Word 2007.

**Importing from Microsoft Word 2007**

The expression activated in Figure 9.2 is copied from Microsoft Word and then pasted into the input field of the translator Word2MosaicMathML. The input has the following appearance:

\[ K_{j,i}^{L,V} = \frac{\gamma_{j,i} \cdot p_{o,j,i}^{L,V}}{p_{j}} \]  

The output of the translation software is:

\[ K_{\{j,i\}^{L,V}} = \frac{\gamma_{\{j,i\}} \cdot p_{\{o,j,i\}^{L,V}}} {p_{\{j\}}} \]  

(\text{Str.1})

(\text{Str.2})
For comparison, the MosaicLatex expression above (String 2), which is also a correct 
LaTeX expression, cf. section 6.3.1, is used as a source in this document and thereby 
rendered as:

\[ K_{j,i}^{L,V} = \frac{\gamma_{j,i} \cdot p_{o,j,i}^{L,V}}{p_j} \]  \hspace{1cm} (9.1)

Subsequently, String 2 is translated to MosaicMathML, the output is shown in section
13.13.

In MOSAIC, the library JEuclid [MSB08] is used to render Presentation MathML 
expressions. The generated MathML expression of the example is rendered as:

\[
K_{j,i}^{L,V} = \frac{\gamma_{j,i} \cdot p_{o,j,i}^{L,V}}{p_j}
\]

**Importing from MathType**

The second equation in Figure 9.2 was implemented in MathType [Des10], cf. Figure
9.3.

![Figure 9.3.: The second equation of Figure 9.2 created in MathType.](image)

In the first step, the translator built into MathType is used with the option ‘\TeX --
LaTeX 2.09 and later’ to create a LaTeX expression of the rendered formula. The 
string obtained is:

\[
P_{o,i}^{L,V} = \left(10\right)^{A_{i} - \frac{B_{i}}{T + C_{i}}}
\]  \hspace{1cm} (Str.3)

In the second step, the translator Latex2MosaicLatex transforms String 3 into the 
correct MosaicLatex expression:

\[
P_{o,i}^{L,V} = (10)^{A_{i} - \frac{B_{i}}{T + C_{i}}}
\]  \hspace{1cm} (Str.4)

Again, String 4 is rendered in this document:

\[
P_{o,i}^{L,V} = (10)^{A_{i} - \frac{B_{i}}{T + C_{i}}}
\]  \hspace{1cm} (9.2)

In the third step, the MosaicMathML expression that is generated by MosaicLatex2-
MosaicMathML based on Str.9.2 is given in section 13.13.
Import of Presentation MathML generated with Firemath

The third equation in Figure 9.2 was implemented in Firemath [Bon12], cf. Figure 9.4.

![Figure 9.4: The third equation of Figure 9.2 created in Firemath.](image)

As pointed out in section 6.2, Presentation MathML expressions contain more information than necessary according to the methodology presented in section 3.4. For example, the elements \texttt{mo}, \texttt{mi}, and \texttt{mn} are intended for the markup of operators, identifiers, and numbers. The proposed methodology does not consider such rendering information in order to enlarge its scope of application. In order to demonstrate that the transformation of the content is independent from the use of the above elements, the subscripts of the expression in Figure 9.4 have been created in an irregular manner, cf. the generated MathML expression in section 13.13.

The software MathParser [WW04] is used to generate a LaTeX expression which is subsequently translated into MosaicLatex with \texttt{Latex2MosaicLatex}. The generated expression is:

\[
F_j \cdot z_{j,i} + L_{j+1} \cdot x_{j+1,i} + V_{j-1} \cdot y_{j-1} = L_j \cdot x_{j,i} + V_j \cdot y_{j,i} \tag{9.3}
\]

which is rendered as

\[
F_j \cdot z_{j,i} + L_{j+1} \cdot x_{j+1,i} + V_{j-1} \cdot y_{j-1} = L_j \cdot x_{j,i} + V_j \cdot y_{j,i}
\]

The generated MosaicMathML expression is shown in section 13.13.

As a final test, the generated MosaicMathML expression is imported as described above:

\[
F_j \cdot z_{j,i} + L_{j+1} \cdot x_{j+1,i} + V_{j-1} \cdot y_{j-1} = L_j \cdot x_{j,i} + V_j \cdot y_{j,i} \tag{Str.6}
\]

Scanning and formula recognition

The scanning of documentation with the aim of obtaining the contained mathematical model can only be of limited reliability: Scanning can be an aid but, in order to ascertain correctness, the obtained and translated content must be reviewed by the modeling
engineer. In this sense, the result of scanning should be ‘correct enough’ that the effort involved in purchasing and using the corresponding hardware and software is justified by a gain in productivity, in comparison to manually copying the mathematical content using editors or languages.

The use of scanned content was tested with all three expressions of the above example. The step of most importance is the optical character recognition (OCR), in which images containing text and formulas are transferred to document formats that are character based (such as LaTeX and MathML) and thereby made accessible for further processing. In this example, the tool InftyReader [Suz12] was used for OCR. The following file types were taken as input: First, PDF-documents obtained by translating LaTeX or publishing the Microsoft Word document; second, TIFF images obtained by scanning printed versions of all documents above. LaTeX was chosen as the output language of the OCR process, the generated LaTeX strings were used as input for MOSAIC.

PDF-documents and scans obtained on the basis of LaTeX→PDF were translated without errors. The generated LaTeX expressions could be used directly in MOSAIC. For PDF-documents and scans obtained from Microsoft Word (using both the integrated editor and MathType) the OCR worked well but the translation contained small errors: Some characters in super or subscripts were misinterpreted (e.g. the superscript $V$ was interpreted as $\nabla$, the index $j$ was interpreted as an $l$), in some cases formatting instructions were added and had to be removed manually. The number of misinterpretations was small, the effort involved in correcting the obtained LaTeX expressions was negligible. As stated initially, reviewing a computerized model obtained by scanning is imperative and therefore, the goal is a small effort of correction. In this sense, these results are more than satisfactory.

9.2. Use of terminologic notations

The equations and nomenclature elements of a CSTR model in [WK05] were entered in MOSAIC. Figure 9.5 shows the equations and some notation elements as they are given in the paper and as they are represented in MOSAIC.

The analogy of representation in the paper and in the modeling environment makes the documentation-level model highly comparable with the model in the document of origin. The nomenclature information is stored with the names of variables and can be displayed throughout the modeling process, cf. Figure 9.6 for the selected variable $C_A$.

The nomenclature information in MOSAIC refers strictly to single symbols. This permits a systematic use of qualifying superscripts and subscripts, e.g. the use of superscripts $L$ and $V$ to distinguish variables of the liquid and vapor phase. In this example, the capital letter $C$ is used in both the concentration and the heat capacity. In this case both descriptions for $C$ are added and the physical meaning of the variable name becomes clear from the descriptions of the subscripts $A$ and $p$. 

9.2. Use of terminologic notations

\[ \frac{dC_A}{dt} = \frac{F}{V}(C_{A0} - C_A) - k_0 C_A e^{-E/RT}, \]
\[ \frac{dT}{dt} = \frac{F}{V}(T_0 - T) + \frac{UA}{\rho C_p V}(T_f - T) + \frac{(-\Delta H)}{\rho C_p} k_0 C_A e^{-E/RT}, \]

\[ t \]

Figure 9.5.: Comparability between the model representation in the article and in MOSAIC as a result of the application of symbolic mathematical expressions and the use of a terminologic notation.

**C**
- Concentration variable in simulation example, kmol/m³

**Subscripts**
- pertaining to species A in simulation example

**Figure 9.6.** User interface for problem specification in MOSAIC. The information contained in the terminologic notation is displayed for the selected variable.
9.3. Case study within the UniCat research project

MOSAIC is developed and applied in the cluster of excellence UniCat [Pre07], which has the aim to find profitable realizations on an industrial scale of the oxidative coupling of methane. One of the tasks to be solved in this research group is the design of the down stream process (cf. [STG+12]), another is the development of a new reactor concept. The models developed in this context must be available in different numerical environments. Relevant applications are the use of Matlab and the integration of the developed model in an AspenPlus flow sheet, which can be achieved by providing a model in Aspen Custom Modeler. This example shows the modeling of a membrane reactor with the help of MOSAIC. The chemical reaction parameters are taken from [LDMHM97], for further information refer to [JGAG+10].

Model equations and results. The change of the concentrations on tube and shell side are described by:

\[
\begin{align*}
\frac{d C_t^i}{d l} &= r_i \cdot w \cdot \frac{A^t}{v^t} - \frac{D_i}{\delta m} \cdot (C_t^i - C_s^i) \cdot \frac{p_{\text{circ},t}}{v^t} \\
\frac{d C_s^i}{d l} &= \frac{D_i}{\delta m} \cdot (C_t^i - C_s^i) \cdot \frac{p_{\text{circ},t}}{v^s}
\end{align*}
\] (9.4)

(9.5)

The reaction mechanism is reduced to the four main reactions, cf. Table 9.3 top left. An Arrhenius based approach is used for the reaction rates:

\[
r_k = k_{a,k} \cdot (e)^{-\frac{E_{A,k}}{RT}} \cdot (C_{a,k})^{m_k} \cdot (C_{b,k})^{n_k}
\] (9.6)
9.3. Case study within the UniCat research project

Figure 9.7.: Plots of the calculation results in MOSAIC (top), Matlab (bottom left), and Aspen Custom Modeler (bottom right).

with \( k \in [1..NK] \), and \( NK = 4 \). The function is applied according to Table 9.3 bottom\(^1\). The production rates of the components are calculated by:

\[
    r_i = \sum_{k=1}^{NK} \nu_k \cdot r_k
\]

(9.7)

with \( NK = 4 \), \( i \in [1..NC] \), and \( NC = 6 \). The mass balances, (9.4) and (9.5), have been implemented in MOSAIC as equations and the reaction and the production rates, (9.6) and (9.7), have been implemented as functions. The model was tested in MOSAIC using C++ and the BzzMath library, code was generated for Matlab and Aspen Custom Modeler. The numerical results of all three environments are shown in Figure 9.7.

Fragments of the generated code. The three languages differ in their area of application and in their generality. Out of the three languages, C++ is the most general, whereas Aspen Custom Modeler is the most problem-specific. Matlab and C++ are procedural languages, whereas the Aspen Custom Modeler language is declarative. These differences have an effect on the complexity of the program code of the modeling problem as will be shown in the following.

The definition of the differential variable in the three languages is done as follows:

- In the C++ code that uses the BzzMath library, an equation system object is created and initialized with a vector containing the design variables. Then a solver

\(^1\)The concentrations in this function must be entered without their engineering unit.
object is created and initialized with the initial values of the state variables, the start value of the differential variable, and the equation system object. Finally, the solver object is used to execute the integration to the end value of the differential variable:

```c
double x_start=0.0; // e0_l
double x_end=0.01;
EquationSystem eqSystem(params);
BzzOdeStiffObject solverObj(y0, x_start,&eqSystem);
[...]
y = solverObj(x_end);
```

• For the use in **Matlab**, the equation system must be implemented in a separate function which is evoked from within the solver method with the help of a function handle. The code for solving the problem with the the Runge-Kutter 4,5 algorithm is:

```matlab
X_START=0.0;
X_END=0.05;
X_INTERVAL=[X_START X_END]; % e0_l
[...]
[X,Y] = ...
    ode45(@(X,Y)calculateDifferentials(X,Y,PARAMS),X_INTERVAL,Y_INIT');
```

• For the use in **Aspen Custom Modeler**, the differential variable of the simulation problem is declared as a ‘differential domain’, which contains the start and the end value as well as information about the grid size. The solver is chosen automatically and there is no modularization of the program, i.e. all variables, parameters and equations are specified in the same block of code. Therefore, the specification of the differential problem is contained in only one line:

```aspen
e0_l as Domain of
    RealParameter(Section(1).Location:0.0;Length:0.01;NumSections:5);
```

The handling of functions in the different languages is implemented as follows:

• For the use in **C++**, the functions are defined once

```c
double fun_uniCat_arrhenius(double std_C_i1,double std_C_i2,...){
    std_r_b=...*pow((std_C_i1),std_m)*pow((std_C_i2),std_n);
    return std_r_b;
}
```

and are called once or several times, according to their application.

```c
double e0_r_k2=fun_uniCat_arrhenius(e0_C_t_i4,e0_C_t_i2,...);
```

• For the use in **Matlab**, functions are translated similarly to C++.
9.4. Modular modeling and reuse

The Aspen Custom Modeler language does not support the definition of functions. Therefore, the functions must be translated into model equations, e.g.

\[ 0 = e_0\cdot r\cdot k_2(i) - (f_0 \cdot k_2 \cdot ((e_0\cdot e))^-((f_0\cdot E\cdot k_2)/(e_0\cdot R\cdot e_0\cdot T)))*([...]); \]

The following examples are designed to demonstrate the modular modeling concept including reuse and generation of program code and documentation. The concept can be used for highly complex models; it should be noted, however, that the correlations are not the focus of the examples.

9.4.1. A basic flash.

First, several necessary equations are introduced. A file name is stated for each model element. If not stated otherwise, all equations and equation systems in this example use the same nomenclature, which is stored as ‘nota_F’.

The component balance for the steady state is created as ‘comp_bal_flash_steady’ with

\[ x_i^{L} \cdot F = x_i^{L} \cdot B + x_i^{V} \cdot D \]  
(9.8)

The summation relations for the molar fractions are created in files ‘flash_sum_liquid’ and ‘flash_sum_vapor’:

\[ 1 = \sum_{i=1}^{N_c} x_i^{L} \]  
(9.9)
\[ 1 = \sum_{i=1}^{N_c} x_i^{V} \]  
(9.10)

And the phase equilibrium is stored as ‘flash_phase_equilib’:

\[ x_i^{V} = K_i \cdot x_i^{L} \]  
(9.11)

In the next step, instead of putting all equations into one system at the same level, smaller equation systems are created as follows: An equation system ‘flash_constit_plain’ is created that contains all constitutive equations and is connected, together with the component balance, in an equation system ‘basic_flash_steady’, cf. Figure 9.8.

An evaluation is created that uses ‘basic_flash_steady’ with two components. The documentation of the example is given in section 13.1, page 156, the program codes for C++ and Matlab are given in section 13.12.1 on pages 198 and 199.
9.4. Modular modeling and reuse

The model ‘basic_flash_steady’ is improved by introducing an equation that calculates the $K$-values based on Raoult’s law and the Antoine equation. This is done by introducing ‘ph_eq_k_raoult_antoine’ as

$$K_i = \frac{(10)^{A_i - \frac{B_i}{T + C_i}}}{P_{sys}}$$

(9.12)

and by creating a new equation system ‘flash_raoult_antoine_a’ that contains ‘basic_flash_steady’ and ‘ph_eq_k_raoult_antoine’, cf. Figure 9.9 left. The documentation for this equation system is contained section 13.2 on page 161.

A better way to augment equation system ‘basic_flash_steady’ is to combine it with an equation for Raoult’s law and thereby permit an independent decision on the physical properties model of the vapor pressure. The equation ‘ph_eq_k_raoult’ is created:

$$K_i = \frac{P_{L,V,o,i}}{P_{sys}}$$

(9.13)

This equation is used together with equation system ‘basic_flash_steady’ in a new system ‘flash_raoult’, cf. Figure 9.9 right.

Now the model ‘flash_raoult’ can be used with different physical property models. First, the Antoine equation is created in file ‘plv_antoine’ (the mathematical expression was used as example in section 9.1):

$$P_{L,V,o,i} = (10)^{A_i - \frac{B_i}{T + C_i}}$$

(9.2)

Second, a DIPPR equation (‘plv_dippr’) is used which, for the sake of this example, is assumed to be part of an existing model library that uses another notation (‘nota_D’) and is introduced using the naming policy ‘encapsulate’ and an appropriate connector.
9.4. Modular modeling and reuse

The DIPPR equation is defined by

\[ v_l = \exp\left( a_l \cdot \frac{b_l}{\theta} + c_l \cdot \ln(\theta) + d_l \cdot (\theta)^{c_l} \right) \]  

(9.14)

The connection structures of the resulting equation systems are shown in Figure 9.10. Both models are used for calculations with the compounds methanol and water. The Antoine parameters are obtained from the NIST data base (http://webbook.nist.gov/chemistry/), the DIPPR parameters are obtained from Chemcad (version 6.1). The documentation for the two cases is shown in sections 13.3 and 13.4, pages 164 and 170.

9.4.3. Use of functions and specific interfaces

Another way to introduce mathematical formulas in MOSAIC models is through the use of functions. When this technique is used, the code generation is adapted automatically to the characteristic of the output language: If the target programming language supports functions, then this way of modeling reduces the size of the equation system. If the target language does not support the formulation of functions, then the corresponding expressions are introduced as equations and added to the equation system.

The function ‘plv_dippr’ is created according to equation (9.14) and applied to the equation system, cf. Figure 9.11, left. Matlab is a language that supports the definition of functions, Aspen Custom Modeler does not support user-defined functions. The corresponding code for both languages is shown in section 13.12.2. The documentation for the model is shown in section 13.5.

The combined use of interfaces and functions allows for the generation of specialized function calls, e.g. code for the implementations of CAPE-OPEN interfaces in different languages or for corresponding interfaces in platforms like Aspen Custom Modeler. The function ‘plv_dippr’ is applied that allows for the calculation of the vapor pressure with the help of a CAPE-OPEN interface, cf. Figure 9.11. Two examples of generated code are shown in section 13.12.3: the code for Matlab in combination with CAPE-OPEN Thermo Import [vB09], and the code for gPROMS using Multiflash [INF12]. The documentation of the model is shown in section 13.6.

9.4.4. Techniques for combining equation systems

Equations and equation systems can be combined using the naming policy ‘encapsulate’ in order to prevent accidental coupling of variables that have the same name but different meanings, cf. section 7.1. A sequence of two flash units is obtained using this feature as shown in Figure 9.12 (1) and (2). Such a sequence can also be represented using
9.4. Modular modeling and reuse

Figure 9.11.: Connection structure of ‘flash_raoult_fun_dippr’ and ‘flash_raoult_fun_co’.

Figure 9.12.: (1) Connection structure of ‘flash_sequence_enc’. (2) Sequence using encapsulation, (3) sequence using ports and streams.

streams. To do so, the flash model must be equipped with ports in a first step; then two instances of this flash model can be connected, cf. Figure 9.13 and Figure 9.12 (3). The documentation of the model ‘basic_flash_ports’ is shown in section 13.8. The documentation of both sequence models is shown in sections 13.7 and 13.9.

9.4.5. Overview of calculation results

All of the above models have been calculated with physical property parameters for the compounds methanol and water and with the configuration

\[ x_{i=1}^{L,V} = 0.5 \frac{mol}{mol}, \quad F = 50 \frac{mol}{s}, \quad T = 353K \quad \text{and} \quad P_{sys} = 101325Pa. \]

The calculation results are shown in Table 9.4.

Figure 9.13.: Connection structures of a flash using ports (left) and a sequence of such flashes (right).
9.4. Modular modeling and reuse

Table 9.4.: Overview of the calculation results for the different flash models.

<table>
<thead>
<tr>
<th>Model</th>
<th>( x^V [\frac{mol}{mol}] )</th>
<th>( D [\frac{mol}{s}] )</th>
<th>( P^L,V_{\alpha,i=1} [Pa] )</th>
<th>( P^L,V_{\alpha,i=2} [Pa] )</th>
<th>( K_{i=1} [\cdot] )</th>
<th>( K_{i=2} [\cdot] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic flash steady</td>
<td>0.7083</td>
<td>14.286</td>
<td>-</td>
<td>-</td>
<td>1.7(^*)</td>
<td>0.5(^*)</td>
</tr>
<tr>
<td>flash raoult antoine b</td>
<td>0.7253</td>
<td>14.394</td>
<td>179749</td>
<td>47084</td>
<td>1.7740</td>
<td>0.4647</td>
</tr>
<tr>
<td>flash raoult fun dippr**</td>
<td>0.7243</td>
<td>14.599</td>
<td>180138</td>
<td>47136</td>
<td>1.7778</td>
<td>0.4652</td>
</tr>
<tr>
<td>flash raoult fun co</td>
<td>0.7442</td>
<td>12.530</td>
<td>180240</td>
<td>44566</td>
<td>1.7788</td>
<td>0.4398</td>
</tr>
</tbody>
</table>

*) design value, **) the model ‘flash raoult dippr’ has the same result.

Figure 9.14.: Inheritance aspect of the modeling concept.

9.4.6. Object-oriented point of view

The extension of models by reuse has an analogy in the class inheritance of object-oriented languages. Equation systems can be considered as classes that inherit the variables from the equation systems they include. From this point of view, a class diagram can be drawn for the models in the example above, cf. Figure 9.14. Another analogy to object-orientation can be found in the use of equation systems that are generic due to contained indices. The specification of the maximum values for all contained indices is done in the problem specification and constitutes an instantiation after which an actual set of variables exists. Then, each problem that is defined based on an equation system uses an instance of it that is manipulated through the definition of the design and initial or guess values, cf. Figure 9.15.

Figure 9.15.: Instantiation – creation of actual problems based on one equation system.
9.5. A distillation column model

The model equations and equation systems introduced above are reused in order to create a reboiler model. A heat balance equation is created and used together with the existing component balance equation, ‘comp_bal_flash_steady’ (eq. 9.8), in a new equation system ‘flash_conserv_steady’. The new heat balance is given by ‘heat_bal_flash_steady’, cf. also Figure 9.16:

\[(10)^{-6} \cdot (h^F \cdot F + (10)^6 \cdot Q^{in}) = (10)^{-6} \cdot (h^L \cdot B + h^V \cdot D)\]  
\[(9.15)\]

In addition, activity coefficients are now considered in the equilibrium constants with the help of equation ‘ph_eq_k_non-ideal’:

\[K_i = \frac{P_{\alpha,i}^{L,V} \cdot \gamma_i}{P}\]  
\[(9.16)\]

Functions that contain CAPE-OPEN interfaces are used for the calculation of physical properties, and the flash model is equipped with ports in order to represent a process unit, cf. Figure 9.17.

A distillation column model is created using the modeling techniques introduced above. A column stage is created of conservative equations for mass and heat and constitutive equations for phase equilibrium and summation relations, cf. Figure 9.18. The condenser model ‘total_cond_steady’ consists of a mass balance and constitutive equations for temperature and mole fractions, cf. Figure 9.18. The stage model represents the complete separation tower and is combined with the condenser in equation system ‘tower_cond_steady’. Subsequently, the equation system ‘tower_cond_steady’ is equipped with ports and connected to the reboiler model with the help of streams, cf. Figure 9.19. The equations can be viewed in the documentation of the model, which is shown in section 13.11. The column model was used for a calculation with the compounds methanol and water and the dimension of eight stages. The results of this calculation are shown in Figure 9.20.
9.5. A distillation column model

SYS 'flash stdy non-id co' Nota: F

SYS 'flash steady h' Nota: F
EQU 'ph eq k non-ideal' Nota: F
FUN 'plv fun cape-open' Nota: D
FUN 'gaml fun cape-open' Nota: D
FUN 'hv fun cape-open' Nota: D

Nota: F
Port: 1 (in) - 'Feed'
Port: 2 (out) - 'Top'
Port: 3 (out) - 'Bottom'

Figure 9.17.: Connection structure of 'flash stdy non-id co'.

SYS 'stage steady' Nota: S
SYS 'stage conserv steady' Nota: S
SYS 'stage constit steady' Nota: S
EQU 'ph eq k gamma' Nota: S
EQU 'phase eq col stage' Nota: S
EQU 'sum liq mf col stage' Nota: S
EQU 'sum vap mf col stage' Nota: S
EQU 'phase eq col stage' Nota: S

Nota: S
SYS 'total cond steady' Nota: S
EQU 'mass bal total cond' Nota: S
EQU 'mole frac total cond' Nota: S
EQU 'temper total cond' Nota: S
EQU 'heat total cond' Nota: S
EQU 'stage comp steady' Nota: S
EQU 'stage energy steady' Nota: S

Figure 9.18.: Connection structure of 'stage steady' and 'total cond steady'.

SYS 'tower cond steady' Nota: S
SYS 'total cond steady' Nota: S
SYS 'stage steady' Nota: S

total cond steady
stage steady
tower cond steady
tower cond std ports
flash stdy non-id ports

SYS 'column steady' Nota: S
SYS 'tower cond std ports' Nota: S

Port: 1 (in) - 'Bot vap in'
Port: 2 (out) - 'Bot liq out'

Figure 9.19.: Connection structure of the column model and its sub models.
9.5. A distillation column model

Figure 9.20.: Simulation results using the column model.
Part IV.

Conclusion
10. Conclusion

In this work, different aspects of the modeling process of custom models in process systems engineering were examined. The desired directions of improvement were formulated in four aims that were used to derive the desired characteristics of a new modeling software. In order to fulfill these characteristics, a new modeling concept was described that focuses on the specification of PSE models on the documentation level. Furthermore, a novel modularization and reuse concept was proposed that focuses on variable names and separates the classification of variables from their definition in equations, and thereby avoids restrictions of reuse. The presented concepts have been implemented in the web-based modeling environment MOSAIC, which made it possible to test the new concepts in practice.

10.1. Practical experience with the implemented modeling environment

The main workflow in MOSAIC is to create the various model elements and to generate program code from the fully specified simulation problem. The aspect of documentation-level modeling is mainly realized by the use of a mathematical symbolic notation, the presence of a terminologic notation, and textual descriptions for most model elements. The used symbolic notation permits the definition of algebraic expressions in a concise and easily readable form. Notably, the use of single characters for the composition variable names does not endanger the comprehensibility, since it is ensured that all used symbols are described by a terminologic notation.

User feedback. The standard language to enter the mathematical content is a subset of LaTeX. The familiarization with LaTeX and the rules of the valid symbolic notation were not reported to be difficult. The import functionality for algebraic expressions from other tools was not utilized by many modelers and therefore, no statements can be made about this aspect. The generation of documentation was requested repeatedly before its implementation and is now used successfully.

The modular modeling concept was reported to be cumbersome. This is especially the case where the modeling tasks were comparatively small, a quick solution was desired, and documentation and applicability on different platforms was not important. In general it could be observed that the use of symbolic notations and modularization slow down the modeling process initially, especially when no reuse is possible. It should be noted that the starting point of this comparison is in favor of programming: Most users state that they are very efficient in the modeling languages they are familiar with. In addition to that, programmed models are defined in a small number of files, whereas in MOSAIC every single equation is separately focused on and stored in an individual file.
However, the reuse of problems in different programming languages and the combination and reuse of existing model elements proved to be very useful and justified the initial modeling effort. The sharing of models between cooperating engineers also played a role in the gain of efficiency; in several cases entire models were reused by colleagues, often for the application on different platforms. Furthermore, code generation permitted to both exploit and compare the strengths of different simulation environments and mathematical libraries. It was reported several times that the generated code was helpful to learn the corresponding modeling language.

10.2. Related publications

An overview of this work is given in [KBK+11]. Different aspects of this work have been discussed in conference papers, cf. [KAGW09, KAGW10, KMBFW11, KAGW11]. The research was supported by the diploma and master theses [Zha10, Gaw10], and [Buc11]. A detailed introduction to the usage of MOSAIC is provided in [Kun10].

The modeling environment has been applied successfully in several projects that were not directly related to this work, cf. [FLR11, Nem11, MMAG+11, Wan11].

10.3. Outlook

10.3.1. Current research projects

The MOSAIC modeling environment is currently used and extended in the collaborative research center InPROMT, [Col10]. In this context a model database is established. Furthermore, the possibility to define optimization problems is provided. An evaluation of mathematical models with respect to different aspects will be implemented and the modular modeling concept will be exploited in order to purposefully select suitable models for specific applications. In order to optimize the use of the database and the models contained, new concepts will be implemented that allow for the assignment and use of structured meta information. Further developments include the analytic generation of derivatives on the documentation level (cf. [MKAGW11]) as well as the automatic reformulation of the numerical problem. Some of the results of the ongoing development of MOSAIC are presented in [KMAGW12] and [MKB+12].

10.3.2. Potential directions and ongoing development

Apart from the above points, several development steps are desirable. As far as the range of supported problems is concerned, the symbolic notation can be extended to comprise partial differential equations. This extension was avoided up to now in order to establish in a robust way the aspects presented in this work. Further adaptations of the symbolic notation can be the of use of index values as algebraic operators, the introduction of special operations (such as rounding), as well the conditional use of equations and equation systems. It must be noted that such changes effect all layers of MOSAIC, since all translation steps are involved. In addition, the specification of adequate representations in the symbolic notation, MosaicLatex, and MosaicMathML
must be considered as well as the general applicability in the various programming languages, which includes the avoidance of numerical errors.

The implementation independence is a good basis for further development. The transformation of the existing problems can be exploited in different ways, e.g. to make the problems available in computer algebra systems. Another promising development step can be the application of CAPE-OPEN interfaces to permit the use of pertinent thermodynamic libraries installed on the modeling server. This step includes the definition of interface-related platform models of CAPE-OPEN interfaces in C++.

The implementation of a visualization of the building block structure of the modular equation systems is another potential feature. In combination with the connection technique of ports and streams, processes build of MOSAIC units could then be visualized in a flowsheet. Such a graph visualization can enhance the usability with the help of interactive creation or adaption of model elements. A more complex step in this context is the exchange of data with existing software such as process data management systems, since it depends on the generality or adaptability of the vendors’ interfaces for simulation data import.

From the side of the principle users, an application of all implemented functionalities, including the described flow-sheeting related connection techniques, is desirable. The improvement of existing concepts and functionalities can only be effected on the basis of user feedback. Finally, the maintenance of the software must be taken into account as an ongoing process.
Bibliography


Part V.

Annex
11. Documentation-level modeling

11.1. Additional proofs

Statement P2: In Algorithm 1, the code lines that read the storage file and operate on the retrieved lines work correctly.

Assumptions for parsing: A line containing a complete non-linear algebraic expression is the input of the algorithm. The non-linear algebraic expression is represented using a notation that is based on the characteristics CN1 through CN6 and that fulfills the requirements RN1 through RN3. The line is stored in a documentation-level format fulfilling the requirements RS1 through RS4.

Proof. The correct working of the code lines will be discussed separately:

Lines 4 and 6. According to requirement RS1, all symbols can be read. According to RS2, all operators can be associated with a line. Finally, according to RN2, each operator found can be associated with an operation. Therefore, it is also possible to determine the correct precedence of the operation and the corresponding operator. Since brackets are considered as enclosing operators, only operators are considered here in which a split is meaningful.

Line 7. Since $\lambda$ is considered to be an alternating sequence of arithmetic operators and their operands, both the left and the right part after a split at both sides of $\omega_o$ are also such sequences. Since $\lambda$ contained a valid expression, $\lambda_{\text{left}}$ and $\lambda_{\text{right}}$ also contain valid expressions.

Lines 8 and 9. The algorithm is applied recursively. This is possible because $\lambda_{\text{left}}$ and $\lambda_{\text{right}}$ are valid lines, cf. Line 7.

Line 12. According to requirements RN1 and RN2, enclosing operators, numbers and variables are distinguishable from one another by their structure. Since the requirements RS1 through RS3 are met, all structure elements can be read and therefore, a determination of the case at hand is possible.

Line 14. According to requirement RS1, all symbols can be read and according to RS2, all operators can be associated with a line. Finally, each operator found can be associated with an operation according to RN2.

Line 15. The arguments of enclosing operators can be retrieved because of requirement RN3. According to the characteristics CN4 and CN5 of the notations considered, each operand is stored in a fragment of a line (which includes entire displaced lines or lines enclosed in special environments). Therefore, the operands are retrievable. Due to the
compliance with requirements RS1 through RS3, the displacements and symbols that characterize the enclosing operators and their operands can be read from the file.

**Line 16** The algorithm is applied recursively. This is possible because all $\lambda_i$ are valid lines, cf. Line 15.

**Lines 19 and 22.** An unambiguous parsing based on the structure is possible because of requirement RN1. The necessary structural information can actually be retrieved due the compliance of the storage format with requirements RS1 through RS3.

**Statement T2:** In Algorithm 2, the code lines that read from and write to the storage files and the code lines that perform operations on obtained lines both work correctly.

**Assumptions for transfer** Both storage formats involved fulfill requirements RS1 through RS4.

**Lines 3 and 4.** The input of the algorithm is an entire line. Since an entire line can be read, the first element of it can be read. Since the line concept is present in the storage format of origin (RS2) and since special operators are present in the form of an environment (RS3), all patterns that are present in one line can be determined. Because the first element is distinguishable from the other elements by its pattern, it can be extracted from the line.

**Lines 6 and 7.** Because of requirement RS1, all symbols are stored individually, and can therefore be read. Due to the assumed compliance with requirement RS4, all symbols that are present in the notation can be represented in both storage formats involved. Therefore, the undisplaced symbols can be read and transferred.\(^1\)

**Line 8.** Since the target storage format supports the line concept (RS2) and the symbols are stored individually (RS1), it is possible to place symbols in an order and therefore append the symbol $\sigma'$.\(^1\)

**Line 10.** The entire lines contained in the patterns can be determined, since the storage format of origin supports special operator environments (RS3).

**Line 11.** This line contains the recursive procedure call, which means that the line works correctly if the whole algorithm works correctly. Any direct transfer operations, as well as read and write operations, are performed in other lines. Therefore, this line is only mentioned here for the sake of completeness, without further proof.

**Lines 12 through 14.** Both storage formats involved comply with requirement RS4, and therefore support all pertinent displacement patterns and enclosing operators. Accordingly, a mapping for undisplaced and displaced lines, and for special operators can be defined so that the concepts of the notation involved are represented. \(\square\)

\(^1\)Since the requirement allows for a surjective instead of a bijective mapping, the transfer can be ambiguous as far as actual characters are concerned, i.e. for each concept several techniques of representation are allowed. However, since the mapping onto the range of represented concepts must be surjective, the concepts are represented unambiguously. More specifically, even if the technique of representation used is chosen at random from the set corresponding to a given concept, unambiguity is maintained since each technique represents the concept.
11.2. EBNF of notation elements in mathematic expressions in MOSAIC

Basic elements

<symbol> ::= <letter> { <letter> } ;

<letter> ::= <Greek letter> | <Latin letter> ;

<Greek letter> ::= ? Greek small and capital letters ? ;

<Latin letter> ::= ? Latin small and capital letters ? ;

<natural number> ::= <non-zero digit> { <digit> } ;

<non-zero digit> ::= "1" | "2" | "3" | "4" | "5"
 | "6" | "7" | "8" | "9" ;

<digit> ::= "0" | <non-zero digit> ;

Notation elements

<fixed subscript> ::= <symbol> ;

<index name> ::= <symbol> ;

<index max val symbol> ::= <symbol> ;

Application

<subscript expression> ::= <fixed subscript>
 | <index expression list>
 | ( <fixed subscript> "," <index expression list> ) ;

<index expression list> ::= <index expression>
 { "," <index expression> } ;

<index expression> ::= <index instance expression>
 | <generic index expression> ;

<index instance expression> ::= <index name> "=" <natural number> ;

<generic index expression> ::= <all values expression>
 | <all values and offset expression>
 | <maximum value expression>
 | <maximum value and offset expression> ;
11.2. EBNF of notation elements in mathematic expressions in MOSAIC

<all values expression> ::= <index name> ;

<all values and offset expression> ::= <index name> ("+" | "-") <natural number> ;

<maximum value expression> ::= <index name> "=" <index max val symbol> ;

<maximum value and offset expression> ::= <index name> "=" <index max val symbol> ("+" | "-") <natural number> ;
12. Detailed description of connection strategies

This section covers the creation of Equation Systems in more detail. First, the basic connection concept is presented that uses the naming policies introduced in section 7.2.1. Then the representation of the PSE concepts of units and streams, and thereby processes, is demonstrated.

Symbols used for Connection Strategies

Connected Element (Equation or Equation System) named $K$, containing the identifiers $a$, $b$, $c$, and $d$, using the Notation $X$. Variable Namings are matched with those found in the implementing super Equation Systems.

Connected Element (Equation or Equation System) named $K$, containing the identifiers $a$, $b$, $c$, and $d$, using the Notation $X$. Upon connection to a superior Equation System, Variable Namings specified in a Connector are translated and then matched. The other Variable Namings are not translated and are given a separate name space.

Name spaces: Equation System named $K$, in whichVariable Namings from different name spaces ($L$ and $K$) and from different Notations ($Y$ and $Z$) are present. In this Equation System, variables $c$ and $d$ belong to name space $L$, variables $\alpha$ and $\beta$ belong to name space $K$.

Notation named $X$ containing descriptions for the Variable Namings $a$, $b$, $c$, $d$, $e$, and $f$.

Connector translating Variable Namings $a$ to $A$ and $b$ to $B$ where $a$ and $b$ are described by the Sub Notation and $A$ and $B$ by the Super Notation.
Connection: The arrow points from the Sub Equation System to the Super Equation System.

Integrating connection: All Variable Namings of the Sub Equation System have to be translated, the rules for the translation are specified in the Connector.

Encapsulating connection: Only the Variable Namings specified in the Connector are translated, the other Variable Namings are given a separate name space.

Interface named \( m \) using Notation \( U \) and containing Variable Namings \( q \) and \( r \).

Port with the ID name ‘In’ using Interface \( m \).

Stream with the ID number 1 using Interface \( o \).

**Naming Policy ‘integrate’**

The naming policy ‘integrate’ should be applied whenever the modeling task includes building up Equation Systems, in which all Variable Namings are described by the same Notation. This policy is useful for the integration of Variable Namings of Equations that use other Notations into the naming concept of the user’s own project.

**Rules**

- One super Notation describes all Variable Namings.
- If different Notations are used, all Variable Namings that do not refer to the super Notation must be translated.

**Behavior and consequences**

- In the super Equation System, Variable Namings are distinguished only with respect to their name in the super Notation.
- The Name Space of the Connected Elements is not used as a distinguishing property.
12. Detailed description of connection strategies

Applications

- Building up a model from scratch using one well-defined Notation.
- Integrating small model parts from other authors into a model in a way that all variables use the same naming concept.
- Building up a library using a well-defined Notation.

Main advantage All Variables are viewed and documented according to the super Notation (only one Notation is used).

Main disadvantage When working with several Notations and big sub Equation Systems, it can be tedious to define translations for all Variable Namings concerned.

Example one All Connected Elements use the same Notation. The Variables are matched according to their Variable Namings in the sub Connected Element. This example is illustrated in Figure 12.1 and corresponds to case 1 in Figure 7.18.

Example two One sub Connected Element uses a different Notation than the super Connected Element as is shown in Figure 12.2. This example corresponds to case 2 in Figure 7.18.

Example three All involved Connected Elements use different Notations. All variables have to be translated into the naming concept of the super Notation, cf. Figure 12.3. This example also corresponds to case 2 in Figure 7.18.

Figure 12.1.: Naming policy ‘integrate’ where all Connected Elements reference the same Notation.
12. Detailed description of connection strategies

**Naming Policy ‘encapsulate’**

Applying this naming policy is useful either for building up Equation Systems with different Notations, when a translation of all Variable Namings into one naming concept is not desired, or for whenever faulty coupling must be avoided for Equation Systems referencing the same Notation.

**Rules**

- For each connection subject to the naming policy ‘encapsulate’, a Connector must be referenced that governs the translation of the Variable Namings of the coupling Variables.

- Only Variable Namings specified in a Connector are translated and integrated into the super Notation.

- All Variable Namings of the sub Connected Element that are not translated are assigned to a different name space.

**Behavior and consequences**

- The super Notation describes the Variable Namings of the coupling Variables.

- The encapsulated sub Equation Systems keep their naming concept, with the exception of the coupling Variables, which are substituted in the mathematical expressions by their Top-Level Naming corresponding to the super Notation.

- Naming conflicts between encapsulated sub Equation Systems and super Equation Systems are avoided.

- The coupling of Equation Systems is effected selectively by defining appropriate Connectors.
12. Detailed description of connection strategies

Figure 12.3.: Naming policy ‘integrate’ with different Notations in all of the involved Connected Elements.

Applications

- The reuse of large and complex Equation Systems from cooperating engineers within the user’s own Equation System.

- The combined use of large Equation Systems, regardless of the referenced Notations.

- The combined use of Equation Systems referencing the same Notation, when coupling based on the plain Variables Namings does not make sense.

Main advantage The focus is put on the modularity of the Equation Systems. The reuse of well-tested modules is facilitated.

Main disadvantage Variable Namings are not generally translated into the super Notation, therefore this policy leads to the combined use of different naming concepts within the mathematical expression of certain Equations. This may cause confusion in some cases, which can be remedied by displaying the Variable Namings together with their Name Spaces.

Example one One sub Connected Element uses a different Notation and is connected using the policy ‘encapsulate’, cf. Figure 12.4. This example corresponds to case 3 in Figure 7.18. It should be noted that the Variable Namings $A$ through $D$ bear the Name Space $K$, which stems from the super Connected Element. This behavior comes from
the fact that the Connected Element \( M \) is added using the naming policy ‘integrate’. In contrast, the non-translated Variable Namings \( c \) and \( d \) bear the Name Space \( L \) of their Connected Element of origin, since in that element, the naming policy ‘encapsulate’ was applied.

**Example two**  All involved Notations are different. As a result, the super Notation is only relevant for the Variable Namings of the coupling Variables, cf. Figure 12.5. This example also corresponds to case 3 in Figure 7.18.

**Example three**  All Connected Elements use the same Notation. Each sub Connected Element represents a self-sufficient model part. Therefore, coupling through exclusively considering the Variable Namings is not desirable for most Variables and the naming policy ‘encapsulate’ is applied. The coupling Variables are specified as such by a pro-forma translation specified in a Connector. It should be noted that, in this case, the same Connector can be used for each connection. This example corresponds to case 4 in Figure 7.18.

![Diagram](image)

**Figure 12.4.** Naming policy ‘encapsulate’ used for one Connected Element that has a different Notation

### Units, ports, and streams

In several modeling approaches, sub models are divided up into building blocks that are interconnected by streams of data. This technique does not actually have to be visualized, since applying the concept of separation is often sufficient. The step-wise consideration of building blocks can be reflected in numerical calculation (e.g. the tearing method in flowsheeting tools). In process systems engineering, the operating units are building blocks at a high level of modeling. In flowsheeting, the operating units are interconnected by material or power streams. On a different level of modeling, the control connections within a process could be considered to be streams of information. It should be noted that the connection techniques introduced in the above sections do actually provide a means for reusing building blocks, however, they cannot conveniently
12. Detailed description of connection strategies

reflect the connection of such blocks by streams, since they do not offer the possibility to define standardized points of interconnection. A corresponding standardization is attained by adding ports to the building blocks and by realizing the interconnections between building blocks as streams between ports.

The separation of large models in the above way has a general character, and a feature that applies such concepts should not be bound to a certain domain or modeling situation. Therefore, the specification of the quantities contained in the ports and streams should be left to the modeler. Under this premise, the user is able to adapt the ports to the model, instead of having to do it the other way around. Furthermore, the proposed concept of ports and streams can be applied to various purposes that are independent of

Figure 12.6.: Naming policy ‘encapsulate’ with the same Notation referenced by all Connected Elements. Accidental coupling is avoided by considering the name spaces of the Variable Namings.
the design of the used modeling environment. This characteristic represents an impor-
tant life cycle aspect, since the above-mentioned subject and tool independence makes
the concept adaptable to future situations.

The next sections describe the realization of units, ports, streams, and processes in
the MOSAIC modeling environment.

**Using External Ports**

It is possible to add standardized points of input and output to Equation Systems. In
MOSAIC, such points are labeled External Port, owing to the fact that one prominent
goal of this technique is the incorporation of the PSE concepts of operating units that
have ports for the purpose of interconnection. Accordingly, an Equation System pos-
sessing External Ports will be regarded as a Unit, cf. the UML presentation in Figure
7.11.

The Interfaces of External Ports can be furnished with additional platform-specific
information so that the models generated by MOSAIC can be integrated directly into
flowsheeting environments, cf. section 8.3.

**Rules**

- An External Port contains an Interface specifying the imposed Variable Namings,
  and a Unique Name to distinguish it from other External Ports of the same Unit.

- External Ports realized by a connection subject to the naming policy ‘encapsulate’,
in which the sub element is the Equation System representing the Unit and the
  super element is the contained Interface. It should be noted that this rule implies
  the use of a Connector.

- The connection to the contained Interface must be comprehensive, i.e. the refer-
  enced Connector must specify sub Variable Namings for each Variable Naming of
  the Interface, and Variables must exist for each specified sub Variable Naming.

- If an Equation System contains External Ports, then it can only be used as a Unit;
  more specifically, it cannot act as a sub Connected Element.

**Behavior and consequences**

- Equation Systems can be given customizable, standardized points of input or out-
  put.

- The PSE concept of operating units bearing ports is introduced.

- The reusability of the original Equation System can be maintained.

- One Equation System can be used as the basis for various independent port con-
  figurations.
The last two of the above points should be explained in more detail. The reusability of Equation Systems employed to form a Unit seems to be impaired by the last rule listed above. In order to avoid restrictions in this respect, Equation Systems should always be created and stored as a whole, representing the full characteristics of the model, and then be connected exclusively to a super Equation System that will contain the External Ports. In this way, the exclusively-connected Equation System itself does not contain External Ports and any connection to a wrapper Equation System containing External Ports only represents an incidence of reuse.

**Applications**

- The use of Equation Systems as operating units in flowsheeting or as building blocks in similar approaches through standardized points of input and output.
- The preparation of Units to work together in processes as described in the next section.
- The definition of Units in a way that the code generated from them can be used in flowsheeting software.

**Main advantages**

- This feature offers a standardization that is highly customizable, since the involved Interfaces, which act as defining elements, are created and reused by the modeler.
- The existence of standardized points of input and output shifts the focus away from the internal behavior of the Equation System, in order to strengthen the view of it as a building block fulfilling a task in a certain context. Building blocks provided in such a way are easier to reuse.
- Although standardized interfaces for data exchange are provided, the original Equation System is not changed.

**Main disadvantage**

- The user must specify a Connector for each added port. To reduce the related effort and error potential, the MOSAIC modeling environment provides functionalities that help the modeler to efficiently create the necessary Connectors.

**Example** Figure 12.7 shows an Equation System named $K$, which is furnished with two External Ports named $One$ and $Two$. Both External Ports use the same Interface named $m$, which references the Notation $U$ and contains the Variable Namings $q$ and $r$. The Connectors specify that $a$ and $c$ correspond to the Variable Naming $q$ in External Ports $One$ and $Two$ respectively, whereas $b$ and $d$ correspond to the Variable Naming $r$ of the respective External Ports. This model can now be reused as a building block by just considering the two External Ports and the Variable Namings of the used Interface $m$. 

It should be noted that the shown use of the External Ports implies that \(a\) and \(c\) represent the same quantity as \(q\) but at different contexts within Equation System \(K\), e.g. a heat stream at the top and at the bottom of an operating unit. The External Ports constitute a representation of the contexts \(One\) and \(Two\), e.g. \(One\) corresponding to the top and \(Two\) corresponding to the bottom.

![Diagram of equation system using ports](image)

**Using Internal Streams**

The previous section showed the way Units can be created based on Equation Systems by adding External Ports. This section describes the creation of processes by selecting a set of Units and interconnecting their External Ports through Internal Streams. The UML representation of this situation is shown in Figure 7.12.

**Rules**

- An Internal Stream connects two External Ports.

- An Internal Stream contains an Interface, which specifies the imposed Variable Namings, and an identification number, which is unique in the super Equation System.

- The connection between External Port and Internal Stream is subject to the naming policy ‘encapsulate’, in which the sub element is the Interface of the External Port and the super element is the Interface of the Internal Stream. This implies the following circumstances:
  - All Variable Namings contained in the External Port’s Interface must be translated into the corresponding Variable Naming of the Internal Stream’s Interface.
  - A Connector must be referenced governing the translation of the Variable Namings.
  - The Interfaces used in the External Port and in the Internal Stream must contain the same number of Variable Namings, which must represent the same quantities.
Behavior and consequences

- Units can be connected, even if their External Ports do not reference the same Interface, as long as the Variable Namings of the concerned Interfaces represent the same quantity.

- This way of connecting existing building blocks gives the model an efficient structure. The Variable Namings represented by Internal Streams can accessed quickly.

Applications

- The creation of PSE process models, in which the operating units and the interconnecting streams are conceptually discernible.

- The use of standardized connections between building blocks whenever it is convenient or necessary, with respect to the problem type or the intended type of goal platform.

Main advantage

- The specification of the quantities in a stream is left to the modeler.

- Since different specifications are allowed for ports and streams, different naming concepts are possible without impairing reuse.

Main disadvantage

- In the most extreme case, many Connectors are involved in each connection by Internal Streams. In order to avoid an increase of modeling errors and effort, appropriate features are implemented in the MOSAIC modeling environment that guide the user through the correct creation of Connectors.

Example  Two independent research teams (Team One and Team Two) have created Equation Systems and attached Ports to them as described above. Both teams have worked independently and have used their own Notations and their own Interfaces, cf. Figure 12.8. Research Team Three creates a process based on the existing units, and decides to use a different Interface for the Internal Streams. The connection of the External Ports within the process model, as well as the use of the Interfaces and the necessary Connectors, are presented in Figure 12.9.
Figure 12.8.: Units using ports created by two different teams.
Figure 12.9.: Connection using ports and streams.
13. Generated documentation for selected example cases

Note: The content of the following sections is automatically generated with the help of MOSAIC (rev. no. 585). No manual changes to the LaTeX source code have been made.

13.1. Basic flash

Evaluation '12273: eval_basic_flash_steady_one.moseva'

Description: Calculation of a simple flash unit in steady state. The K-values do not represent a concrete set of compounds.
Equation System: 12270: basic_flash_steady.moseqs
IndexSpecification: e[0]12270.Nc = 2

Variable Specification: 12272: var Specs bfs one.mosvar
Results Specification: 13352: res bfs one.mosvar

Hierarchical view of equations:

<table>
<thead>
<tr>
<th>Equation System: 12270: basic_flash_steady.moseqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: Basic model of a flash in steady state</td>
</tr>
</tbody>
</table>

Connected Equations:
- Eq: 12264: comp_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Steady-state component balance of a simple flash.

\[ x_i^{L,V} \cdot F = x_i^L \cdot B + x_i^V \cdot D \] (13.1)

Connected EQ-Systems:
13.1. Basic flash

- (1)12268: flash_constitutive_plain.moseqs

Connection Level (1) – EQ-Systems connected to 12270: basic_flash_steady.moseqs:

**Equation System: 12268: flash_constitutive_plain.moseqs**

Description: Constitutive equations of a flash unit. Contains no summation relation for the feed molar fraction, so all must molar fractions of feed must be design values

Connected Equations:
  \[ x_i^V = K_i \cdot x_i^L \]  \hspace{1cm} (13.2)
  \[ 1 = \sum_{i=1}^{N_c} x_i^L \]  \hspace{1cm} (13.3)
  \[ 1 = \sum_{i=1}^{N_c} x_i^V \]  \hspace{1cm} (13.4)

Equation instances:
  \[ \sum_{i=1}^{L,V} x_{i=1}^{L,V} \cdot F = \sum_{i=1}^{L} x_{i=1}^{L} \cdot B + \sum_{i=1}^{V} x_{i=1}^{V} \cdot D \]  \hspace{1cm} (13.5)
\[ x_{i=2}^L V \cdot F = x_{i=2}^L B + x_{i=2}^V D \]  
\hspace{1cm} (13.6)


\[ x_{i=1}^V = K_{i=1} \cdot x_{i=1}^L \]  
\hspace{1cm} (13.7)

\[ x_{i=2}^V = K_{i=2} \cdot x_{i=2}^L \]  
\hspace{1cm} (13.8)


\[ 1 = (x_{i=1}^L + x_{i=2}^L) \]  
\hspace{1cm} (13.9)


\[ 1 = (x_{i=1}^V + x_{i=2}^V) \]  
\hspace{1cm} (13.10)

**Variable Specs '12272: var\_specs\_bfs\_one.mosvar’**

**Design variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0.F</td>
<td>50.0</td>
</tr>
<tr>
<td>e0.K_{i=1}</td>
<td>1.7</td>
</tr>
<tr>
<td>e0.K_{i=2}</td>
<td>0.5</td>
</tr>
<tr>
<td>e0.x_{i=1}^L V</td>
<td>0.5</td>
</tr>
<tr>
<td>e0.x_{i=2}^L V</td>
<td>0.5</td>
</tr>
</tbody>
</table>

**Iteration variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0.B</td>
<td>25.0</td>
</tr>
<tr>
<td>e0.D</td>
<td>25.0</td>
</tr>
<tr>
<td>e0.x_{i=1}^L</td>
<td>0.5</td>
</tr>
<tr>
<td>e0.x_{i=1}^V</td>
<td>0.5</td>
</tr>
<tr>
<td>e0.x_{i=2}^L</td>
<td>0.5</td>
</tr>
<tr>
<td>e0.x_{i=2}^V</td>
<td>0.5</td>
</tr>
</tbody>
</table>

**Results '13352: res\_bfs\_one.mosvar’**

**Design variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0.F</td>
<td>50.0</td>
</tr>
</tbody>
</table>
13.1. Basic flash

\[ e_0 K_{i=1} = 1.7 \]
\[ e_0 K_{i=2} = 0.5 \]
\[ e_0 x_{L,V}^{L,V} = 0.5 \]
\[ e_0 x_{i=2}^{L,V} = 0.5 \]

**Iteration variables**

\[ e_0 B = 35.71428558733966 \]
\[ e_0 D = 14.28571433759471 \]
\[ e_0 x_{I=1}^L = 0.4166666666666667 \]
\[ e_0 x_{I=1}^V = 0.7083333333333334 \]
\[ e_0 x_{I=2}^L = 0.5833333333333334 \]
\[ e_0 x_{I=2}^V = 0.2916666666666667 \]

**Notation '12263: nota_F.mosnot’**

**Base line symbols**

- \( B \): bottom output stream [mol/s]
- \( D \): head output stream [mol/s]
- \( F \): material stream [mol/s]
- \( K \): equilibrium constant [-]
- \( P \): pressure [Pa]
- \( Q \): heat input or output [MJ/s]
- \( T \): temperature [K]
- \( \gamma \): activity coefficient [-]
- \( h \): enthalpy [J/mol]
- \( x \): mole fraction [mol/mol]

**Superscripts**

\( F \): feed
\( L \): liquid
\( V \): vapor
\( in \): input

**Subscripts**

\( o \): reference conditions
\( sys \): system

**Indices**

\( i \): 1..\( Nc \): compound index
— (end of generated documentation) —
13.2. Flash raoult antoine a

Note: Notation ‘example/nota_F.mosnot’ see section 13.1. This section contains the documentation of the generic equation system only. For a full evaluation of a similar model refer to section 13.3.

— (begin of generated documentation) —

EQ System '12280: flash_raoult_antoine_a.moseqs'

Description: 'Basic flash. K-value based on raoult’s law and the antoine equation.'
Notation: '12263: nota_F.mosnot'

Connected Elements

- (0) 12270: basic_flash_steady.moseqs
  Naming policy: integrate
  Used Connector: none
  External indices: none

- (1) 12279: ph_eq_k_raoult_antoine.moseq
  Naming policy: integrate
  Used Connector: none
  External indices: none

Hierarchical view of contained equations:

<table>
<thead>
<tr>
<th>Equation System: 12280: flash_raoult_antoine_a.moseqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: Basic flash. K-value based on raoult’s law and the antoine equation.</td>
</tr>
<tr>
<td>Connected Equations:</td>
</tr>
<tr>
<td>- Eq: 12279: ph_eq_k_raoult_antoine.moseq (using Nota: 12263: nota_F.mosnot)</td>
</tr>
<tr>
<td>Desc.: The phase equilibrium constant expressed by using Raout’s law. The Antoine equation is used to calculate the vapor pressure. Parameter List: 12278: std_phys_props_param_list.mospar</td>
</tr>
</tbody>
</table>
| \[
| K_i = \frac{(A_i - \frac{n_i}{T + b_i})}{P_{sys}} \quad (13.11) 
| \]
| Connected EQ-Systems: |
| - (0)12270: basic_flash_steady.moseqs |
13.2. Flash raoult antoine a

Connection Level (1) – EQ-Systems connected to 12280: flash_raoult_antoine_a.moseqs:

<table>
<thead>
<tr>
<th>Equation System: 12270: basic_flash_steady.moseqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: Basic model of a flash in steady state</td>
</tr>
<tr>
<td>Connected Equations:</td>
</tr>
<tr>
<td>• Eq: 12264: comp_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot)</td>
</tr>
<tr>
<td>Desc.: Steady-state component balance of a simple flash.</td>
</tr>
<tr>
<td>[ x_i^{L,V} \cdot F = x_i^L \cdot B + x_i^V \cdot D ] (13.12)</td>
</tr>
<tr>
<td>Connected EQ-Systems:</td>
</tr>
<tr>
<td>• (1)12268: flash_constitutive_plain.moseqs</td>
</tr>
</tbody>
</table>

Connection Level (2) – EQ-Systems connected to 12270: basic_flash_steady.moseqs:

<table>
<thead>
<tr>
<th>Equation System: 12268: flash_constitutive_plain.moseqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: Constitutive equations of a flash unit. Contains no summation relation for the feed molar fraction, so all must molar fractions of feed must be design values</td>
</tr>
<tr>
<td>Connected Equations:</td>
</tr>
<tr>
<td>• Eq: 12266: flash_sum_vapor.mosequ (using Nota: 12263: nota_F.mosnot)</td>
</tr>
<tr>
<td>Desc.: Sum relation of molar fractions of flash head output</td>
</tr>
<tr>
<td>[ 1 = \sum_{i=1}^{N_c} x_i^V ] (13.13)</td>
</tr>
<tr>
<td>• Eq: 12265: flash_sum_liquid.mosequ (using Nota: 12263: nota_F.mosnot)</td>
</tr>
<tr>
<td>Desc.: Sum relation of molar fractions of flash bottom output</td>
</tr>
<tr>
<td>[ 1 = \sum_{i=1}^{N_c} x_i^L ] (13.14)</td>
</tr>
</tbody>
</table>
13.2. *Flash raoult antoine a*

- Eq: 12267: flash_phase_equil.mosequ (using Nota: 12263: nota_F.mosnot)
  
  Desc.: General phase equilibrium.

  \[ x_i^V = K_i \cdot x_i^L \quad (13.15) \]

— (end of generated documentation) —
13.3. Flash raoult antoine b

Note: Notation ‘example/nota_F.mosnot’ see section 13.1.

— (begin of generated documentation) —

**Evaluation '12291: eval_basic_flash_steady_two.moseva’**

Description: Evaluation of a steady-state flash for the compounds methanol and water.
Equation System: 12289: flash_raoult_antoine_b.moseqs
IndexSpecification: e[0]12289.Nc = 2
12287.NComps = 2

Variable Specification: 12290: var_specs_bfs_two.mosvar
Parameter Specification: 12294: par_specs_antoine_methanol_water_eval_two_b.mosvar
Results Specification: 13361: res_bfs_two.mosvar

Hierarchical view of equations:

**Equation System: 12289: flash_raoult_antoine_b.moseqs**

Description: Steady-state flash using Raoult’s law and Antoine equation for the calculation of the phase equilibrium.

Connected Equations:
- Eq: 12287: plv_antoine.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Antoine equation for vapor pressure calculation Parameter List: 12278: std_phys_props_param_list.mospar

\[ P_{o,V}^{L,V} = (10)^{\left( A_i - \frac{B_i}{T + C_i} \right)} \]  \hspace{1cm} (13.16)

Connected EQ-Systems:

- (0)12288: flash_raoult.moseqs

Connection Level (1) – EQ-Systems connected to 12289: flash_raoult_antoine_b.moseqs:
13.3. Flash raoult antoine b

**Equation System: 12288: flash_raoult.moseqs**

Description: Model of a steady-state flash using Raoult’s law for the phase equilibrium

Connected Equations:
- Eq: 12286: ph_eq_k_raoult.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Equilibrium constant according to Raoult’s law

\[
K_i = \frac{P_{o,i}^{L,V}}{P_{sys}}
\]

(13.17)

Connected EQ-Systems:
- (0)12270: basic_flash_steady.moseqs

---

Connection Level (2) – EQ-Systems connected to 12288: flash_raoult.moseqs:

**Equation System: 12270: basic_flash_steady.moseqs**

Description: Basic model of a flash in steady state

Connected Equations:
- Eq: 12264: comp_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Steady-state component balance of a simple flash.

\[
x_i^{L,V} \cdot F = x_i^L \cdot B + x_i^V \cdot D
\]

(13.18)

Connected EQ-Systems:
- (1)12268: flash_constitutive_plain.moseqs

---

Connection Level (3) – EQ-Systems connected to 12270: basic_flash_steady.moseqs:
Equation System: 12268: flash_constitutive_plain.moseqs

Description: Constitutive equations of a flash unit. Contains no summation relation for the feed molar fraction, so all must molar fractions of feed must be design values

Connected Equations:

- Eq: 12267: flash_phase_equilib.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: General phase equilibrium.
  
  \[ x^V_i = K_i \cdot x^L_i \quad (13.19) \]

  Desc.: Sum relation of molar fractions of flash head output
  
  \[ 1 = \sum_{i=1}^{N_c} x^V_i \quad (13.20) \]

- Eq: 12265: flash_sum_liquid.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Sum relation of molar fractions of flash bottom output
  
  \[ 1 = \sum_{i=1}^{N_c} x^L_i \quad (13.21) \]

Equation instances:

Eq: 12264: comp_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot). Description: Steady-state component balance of a simple flash..

\[ x^L_{i=1} \cdot F = x^L_{i=1} \cdot B + x^V_{i=1} \cdot D \quad (13.22) \]

\[ x^L_{i=2} \cdot F = x^L_{i=2} \cdot B + x^V_{i=2} \cdot D \quad (13.23) \]

Eq: 12267: flash_phase_equilib.mosequ (using Nota: 12263: nota_F.mosnot). Description: General phase equilibrium..

\[ x^V_{i=1} = K_{i=1} \cdot x^L_{i=1} \quad (13.24) \]
\[ x_i^{V} = K_i = 2 \cdot x_i^{L} \]  

(13.25)


\[ 1 = (x_{i=1}^{L} + x_{i=2}^{L}) \]  

(13.26)


\[ 1 = (x_{i=1}^{V} + x_{i=2}^{V}) \]  

(13.27)

Eq: 12286: ph_eq_k_raoult.mosequ (using Nota: 12263: nota_F.mosnot). Description: Equilibrium constant according to Raoult’s law.

\[ K_{i=1} = \frac{P_{L,V_{i=1}}^{L,V}}{P_{sys}} \]  

(13.28)

\[ K_{i=2} = \frac{P_{L,V_{i=2}}^{L,V}}{P_{sys}} \]  

(13.29)


\[ P_{o,i=1}^{L,V} = (10)^{\left(A_{i=1} - \frac{B_{i=1}}{T + C_{i=1}}\right)} \]  

(13.30)

\[ P_{o,i=2}^{L,V} = (10)^{\left(A_{i=2} - \frac{B_{i=2}}{T + C_{i=2}}\right)} \]  

(13.31)

**Variable Specs '12290: var_specs_bfs_two.mosvar'**

Design variables

\[
\begin{align*}
e_0.F &= 50.0 \\
e_0.P_{sys} &= 101325.0 \\
e_0.T &= 353.0 \\
e_0.x_{i=1}^{L,V} &= 0.5 \\
e_0.x_{i=2}^{L,V} &= 0.5
\end{align*}
\]

Iteration variables

\[
\begin{align*}
e_0.B &= 25.0 \\
e_0.D &= 25.0
\end{align*}
\]
13.3. Flash raoult antoine b

\[
e0. K_{i=1} = 1.0 \\
e0. K_{i=2} = 1.0 \\
e0. P_{L,V}^{L,V}_{i=1,o} = 100000.0 \\
e0. P_{L,V}^{L,V}_{i=2,o} = 100000.0 \\
e0. x_{L}^{L,V}_{i=1} = 0.5 \\
e0. x_{L}^{L,V}_{i=2} = 0.5 \\
e0. x_{V}^{L,V}_{i=1} = 0.5 \\
e0. x_{V}^{L,V}_{i=2} = 0.5
\]

Parameter Specs ’12294: par_specs_antoine_methanol_water_eval_two_b.mosvar’

Parameters

\[
12287. A_{i=1} = 10.20409 \\
12287. A_{i=2} = 10.0768 \\
12287. B_{i=1} = 1581.341 \\
12287. B_{i=2} = 1659.793 \\
12287. C_{i=1} = -33.5 \\
12287. C_{i=2} = -45.854
\]

Results ’13361: res_bfs_two.mosvar’

Design variables

\[
e0. F = 50.0 \\
e0. P_{sys} = 101325.0 \\
e0. T = 353.0 \\
e0. x_{L,V}^{L,V}_{i=1} = 0.5 \\
e0. x_{L,V}^{L,V}_{i=2} = 0.5
\]

Iteration variables

\[
e0. B = 35.59041155248006 \\
e0. D = 14.409588447304948 \\
e0. K_{i=1} = 1.773982320865163 \\
e0. K_{i=2} = 0.4646880179007732 \\
e0. P_{L,V}^{L,V}_{i=1,o} = 179748.75866166264 \\
e0. P_{L,V}^{L,V}_{i=2,o} = 47084.513413795845 \\
e0. x_{L}^{L}_{i=1} = 0.408846433020475 \\
e0. x_{V}^{L}_{i=1} = 0.7252863441270567 \\
e0. x_{L}^{V}_{i=2} = 0.5911535669759374 \\
e0. x_{V}^{V}_{i=2} = 0.2747136558729215
\]
**13.4. Flash raoult dippr**

Note: Notation ‘example/nota.F.mosnot’ see section 13.1 (removed here for reasons of space).

— (begin of generated documentation) —

**Evaluation ’12303: eval_basic_flash_steady_three.moseva’**

Description: Steady state flash calculation for the compounds methanol and water.
Equation System: 12301: flash_raoult_dippr.moseqs
12299.NComps = 2
e[0]12301.Nc = 2

Variable Specification: 12302: varspecs_bfs_three.mosvar
Parameter Specification: 12304: parspecs_antoine_methanol_water_eval_three.mosvar
Results Specification: 13379: res_bfs_three.mosvar

Hierarchical view of equations:

---

**Equation System: 12301: flash_raoult_dippr.moseqs**

Description: Steady-state flash using Raoult’s law for the phase equilibrium and a DIPPR equation for the vapor pressure.

Connected Equations:

- Eq: 12299: plv_dippr.mosequ (using Nota: 12295: nota_D.mosnot)
  Desc.: DIPPR equation 101 Parameter List: 12296: dippr_phys_props_param_list.mospar
  \[ v_l = \exp \left( a_l + \frac{b_l}{\theta} + c_l \cdot \ln (\theta) + d_l \cdot (\theta)^{e_l} \right) \]  

(13.32)

Connected EQ-Systems:

- (0)12288: flash_raoult.moseqs

---

Connection Level (1) – EQ-Systems connected to 12301: flash_raoult_dippr.moseqs:
Equation System: 12288: flash_raoult.moseqs

Description: Model of a steady-state flash using Raoult’s law for the phase equilibrium

Connected Equations:

- Eq: 12286: ph_eq_k_raoult.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Equilibrium constant according to Raoult’s law

\[
K_i = \frac{P_{o,i}^{L,V}}{P_{sys}}
\]

Equation System: 12270: basic_flash_steady.moseqs

Description: Basic model of a flash in steady state

Connected Equations:

- Eq: 12264: comp_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Steady-state component balance of a simple flash.

\[
x_i^{L,V} \cdot F = x_i^L \cdot B + x_i^V \cdot D
\]

Equation System: 12270: basic_flash_steady.moseqs

Description: Basic model of a flash in steady state

Connected Equations:

- Eq: 12264: comp_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Steady-state component balance of a simple flash.

\[
x_i^{L,V} \cdot F = x_i^L \cdot B + x_i^V \cdot D
\]
13.4. Flash raoult dippr

Connection Level (3) – EQ-Systems connected to 12270: basic_flash_steady.moseqs:

**Equation System: 12268: flash_constitutive_plain.moseqs**

Description: Constitutive equations of a flash unit. Contains no summation relation for the feed molar fraction, so all must molar fractions of feed must be design values.

Connected Equations:

- Eq: 12267: flash_phase_equil.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: General phase equilibrium.
  \[ x_i^V = K_i \cdot x_i^L \]  \hspace{1cm} (13.35)

  Desc.: Sum relation of molar fractions of flash head output.
  \[ 1 = \sum_{i=1}^{N_c} x_i^V \]  \hspace{1cm} (13.36)

- Eq: 12265: flash_sum_liquid.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Sum relation of molar fractions of flash bottom output.
  \[ 1 = \sum_{i=1}^{N_c} x_i^L \]  \hspace{1cm} (13.37)

**Equation instances:**

Description: Steady-state component balance of a simple flash.
\[ x_i^{L,V} \cdot F = x_i^{L} \cdot B + x_i^{V} \cdot D \]  \hspace{1cm} (13.38)
\[ x_i^{L,V} \cdot F = x_i^{L} \cdot B + x_i^{V} \cdot D \]  \hspace{1cm} (13.39)

Description: General phase equilibrium.
\[ x_i^{V} = K_i \cdot x_i^{L} \]  \hspace{1cm} (13.40)
\[ x_i^{V} = K_i^{V} \cdot x_i^{L} \]  \hspace{1cm} (13.41)


\[ 1 = (x_i^{L} + x_i^{L}) \]  \hspace{1cm} (13.42)


\[ 1 = (x_i^{V} + x_i^{V}) \]  \hspace{1cm} (13.43)

Eq: 12286: ph_eq_k_raoult.mosequ (using Nota: 12263: nota_F.mosnot). Description: Equilibrium constant according to Raoult’s law.

\[ K_i = \frac{P_{o,i}^{L,V}}{P_{sys}} \]  \hspace{1cm} (13.44)

\[ K_i^{V} = \frac{P_{o,i}^{L,V}}{P_{sys}} \]  \hspace{1cm} (13.45)


\[ P_{o,i}^{L,V} = \exp \left( a_{l=1} + b_{l=1} \cdot \ln (T) + c_{l=1} \cdot (T)(e_{l=1}) \right) \]  \hspace{1cm} (13.46)

\[ P_{o,i}^{L,V} = \exp \left( a_{l=2} + b_{l=2} \cdot \ln (T) + c_{l=2} \cdot (T)(e_{l=2}) \right) \]  \hspace{1cm} (13.47)

Variable Specs ’12302: var_specs_bfs_three.mosvar’

Design variables

\[
\begin{align*}
e0.F &= 50.0 \\
e0.P_{sys} &= 101325.0 \\
e0.T &= 353.0 \\
e0.x^{L,V}_{i=1} &= 0.5 \\
e0.x^{L,V}_{i=2} &= 0.5
\end{align*}
\]

Iteration variables

\[
\begin{align*}
e0.B &= 25.0
\end{align*}
\]
13.4. Flash raoult dippr

\[
e0.D = 25.0 \\
e0.K_{i=1} = 1.0 \\
e0.K_{i=2} = 1.0 \\
e0.P_{i,L,V}^{i=1,o} = 100000.0 \\
e0.P_{i,L,V}^{i=2,o} = 100000.0 \\
e0.x_{i=1}^{L,V} = 0.5 \\
e0.x_{i=1}^{V} = 0.5 \\
e0.x_{i=2}^{L,V} = 0.5 \\
e0.x_{i=2}^{V} = 0.5
\]

Parameter Specs '12304:
\texttt{par_specs_antoine_methanol_water_eval_three.mosvar}'

Parameters

\[
\begin{align*}
12299.a_{i=1} & = 81.768 \\
12299.a_{i=2} & = 72.55 \\
12299.b_{i=1} & = -6876.0 \\
12299.b_{i=2} & = -7206.7 \\
12299.c_{i=1} & = -8.7078 \\
12299.c_{i=2} & = -7.1385 \\
12299.d_{i=1} & = 7.1926E - 6 \\
12299.d_{i=2} & = 4.046E - 6 \\
12299.e_{i=1} & = 2.0 \\
12299.e_{i=2} & = 2.0
\end{align*}
\]

Results '13379: res_bfs_three.mosvar'

Design variables

\[
\begin{align*}
e0.F & = 50.0 \\
e0.P_{sys} & = 101325.0 \\
e0.T & = 353.0 \\
e0.x_{i=1}^{L,V} & = 0.5 \\
e0.x_{i=2}^{L,V} & = 0.5
\end{align*}
\]

Iteration variables

\[
\begin{align*}
e0.B & = 35.38525262737454 \\
e0.D & = 14.61474777025367
\end{align*}
\]
13.4. Flash raoult dippr

\[
e0.K_{i=1} = 1.7778240179518887 \\
e0.K_{i=2} = 0.4652020747402776 \\
e0.P^{L,V}_{i=1,o} = 180138.01861897513 \\
e0.P^{L,V}_{i=2,o} = 47136.60022305863 \\
e0.x^L_{i=1} = 0.4074177521871323 \\
e0.x^V_{i=1} = 0.7243170651782745 \\
e0.x^L_{i=2} = 0.592582478129009 \\
e0.x^V_{i=2} = 0.2756829348216666
\]

**Notation ’12277: nota_P.mosnot’**

Base line symbols

- \( A \): physical properties parameter
- \( B \): physical properties parameter
- \( C \): physical properties parameter
- \( \gamma \): activity coefficient [-]
- \( \theta \): temperature [K]
- \( a \): dippr parameter
- \( b \): dippr parameter
- \( c \): dippr parameter
- \( d \): dippr parameter
- \( e \): dippr parameter
- \( f \): dippr parameter
- \( h \): enthalpy [J/mol]
- \( p \): pressure [Pa]
- \( x \): molar fraction [mol/mol]

**Superscripts**

- \( L \): liquid
- \( V \): vapor

**Indices**

- \( i \): 1..\( N\text{Comps} \) Number of components
- \( l \): 1..\( N\text{Comps} \) Number of components
Notation '12263: nota_F.mosnot'

Base line symbols

- $B$: bottom output stream [mol/s]
- $D$: head output stream [mol/s]
- $F$: material stream [mol/s]
- $K$: equilibrium constant [-]
- $P$: pressure [Pa]
- $Q$: heat input or output [MJ/s]
- $T$: temperature [K]
- $\gamma$: activity coefficient [-]
- $h$: enthalpy [J/mol]
- $x$: mole fraction [mol/mol]

Superscripts

- $F$: feed
- $L$: liquid
- $V$: vapor
- $in$: input

Subscripts

- $o$: reference conditions
- $sys$: system

Indices

- $i$: 1..$N_c$: compound index

— (end of generated documentation) —
13.5. Flash raoult function DIPPR

— (begin of generated documentation) —

EQ System '12674: flash_raoult_fun_dippr.moseqs'

Description: 'Steady-state flash model. Raoult’s law for phase equilibrium. Function call to DIPPR 101 for vapor pressure.'

Notation: '12263: nota_F.mosnot'

Connected Elements

- (0) 12288: flash_raoult.moseqs
  Naming policy: integrate
  Used Connector: none
  External indices: none

Functions

- Function 12673: plv_fun_dippr.mosfun
  Number of appliances: 1

Hierarchichal view of contained equations:

<table>
<thead>
<tr>
<th>Equation System: 12674: flash_raoult_fun_dippr.moseqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: Model of a steady-state flash using Raoult’s law for the phase equilibrium</td>
</tr>
</tbody>
</table>

Applied Functions:

  Uses Param List: 12672: dippr_phys_props_param_impl.mospar with index: 1

\[ v = f(\theta) \quad (13.48) \]

with

\[ f = \exp \left( a + \frac{b}{\theta} + c \cdot \ln (\theta) + d \cdot (\theta)^{\epsilon} \right) \quad (13.49) \]

applied as

\[ P_{o,a}^{L,V} = f(T) \quad (13.50) \]
Connected EQ-Systems:

- (0)12288: flash_raoult.moseqs

Connection Level (1) – EQ-Systems connected to 12674: flash_raoult_fun_dippr.moseqs:

**Equation System: 12288: flash_raoult.moseqs**

Description: Model of a steady-state flash using Raoult’s law for the phase equilibrium

Connected Equations:

- Eq: 12286: ph_eq_k_raoult.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Equilibrium constant according to Raoult’s law

\[
K_i = \frac{P_{o,i}^{L,V}}{P_{sys}}
\]  

(13.51)

Connected EQ-Systems:

- (0)12270: basic_flash_steady.moseqs

Connection Level (2) – EQ-Systems connected to 12288: flash_raoult.moseqs:

**Equation System: 12270: basic_flash_steady.moseqs**

Description: Basic model of a flash in steady state

Connected Equations:

- Eq: 12264: comp_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot)
13.5. Flash raoult function DIPPR

Desc.: Steady-state component balance of a simple flash.

\[ x_i^{LV} \cdot F = x_i^L \cdot B + x_i^V \cdot D \]  \hspace{1cm} (13.52)

Connected EQ-Systems:

- (1)12268: flash_constitutive_plain.moseqs

Connection Level (3) – EQ-Systems connected to 12270: basic_flash_steady.moseqs:

**Equation System: 12268: flash_constitutive_plain.moseqs**

Description: Constitutive equations of a flash unit. Contains no summation relation for the feed molar fraction, so all must molar fractions of feed must be design values

Connected Equations:

  Desc.: Sum relation of molar fractions of flash head output

\[ 1 = \sum_{i=1}^{N_c} x_i^V \]  \hspace{1cm} (13.53)

- Eq: 12267: flash_phase_equilib.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: General phase equilibrium.

\[ x_i^V = K_i \cdot x_i^L \]  \hspace{1cm} (13.54)

- Eq: 12265: flash_sum_liquid.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Sum relation of molar fractions of flash bottom output

\[ 1 = \sum_{i=1}^{N_c} x_i^L \]  \hspace{1cm} (13.55)

— (end of generated documentation) —
13.6. Flash raoult function CAPE-OPEN

— (begin of generated documentation) —

EQ System '12680: flash_raoult_fun_co.moseqs'

Description: 'Steady-state flash. Raoult’s law is used for phase equilibrium. CAPE-OPEN is used to calculate the vapor pressure.'

Notation: '12263: nota_F.mosnot'

Connected Elements

- (0) 12288: flash_raoult.moseqs
  Naming policy: integrate
  Used Connector: none
  External indices: none

Functions

- Function 12679: plv_fun_cape-open.mosfun
  Number of appliances: 1

Hierarchichal view of contained equations:

<table>
<thead>
<tr>
<th>Equation System: 12680: flash_raoult_fun_co.moseqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: Model of a steady-state flash using Raoult’s law for the phase equilibrium</td>
</tr>
<tr>
<td>Applied Functions:</td>
</tr>
<tr>
<td>• Fun: 12679: plv_fun_cape-open.mosfun (using Nota: 12277: nota_P.mosnot)</td>
</tr>
<tr>
<td>Desc.: calculate the vapor pressure of the compounds of a mixture using cape open.</td>
</tr>
</tbody>
</table>

\[ p_i^{L,V} = f(\theta) \]  
(13.56)

with \( f = \) calc. by: CO Calculate Vapor Pressure
applied as

\[ P_{o,i}^{L,V} = f(T) \]  
(13.57)

Connected EQ-Systems:

- (0)12288: flash_raoult.moseqs
13.6. Flash raoult function CAPE-OPEN

Connection Level (1) – EQ-Systems connected to 12680: flash_raoult_fun_co.moseqs:

**Equation System: 12288: flash_raoult.moseqs**

Description: Model of a steady-state flash using Raoult’s law for the phase equilibrium

Connected Equations:

- Eq: 12286: ph_eq_k_raoult.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Equilibrium constant according to Raoult’s law

\[ K_i = \frac{P_{o,i}^{L_v}}{P_{sys}} \]  \hspace{2cm} (13.58)

Connected EQ-Systems:

- (0)12270: basic_flash_steady.moseqs

Connection Level (2) – EQ-Systems connected to 12288: flash_raoult.moseqs:

**Equation System: 12270: basic_flash_steady.moseqs**

Description: Basic model of a flash in steady state

Connected Equations:

- Eq: 12264: comp_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Steady-state component balance of a simple flash.

\[ x_i^{L_v} \cdot F = x_i^L \cdot B + x_i^V \cdot D \]  \hspace{2cm} (13.59)

Connected EQ-Systems:

- (1)12268: flash_constitutive_plain.moseqs
Connection Level (3) – EQ-Systems connected to 12270: basic\_flash\_steady.moseqs:

**Equation System: 12268: flash\_constitutive\_plain.moseqs**

Description: Constitutive equations of a flash unit. Contains no summation relation for the feed molar fraction, so all must molar fractions of feed must be design values

Connected Equations:

- Eq: 12265: flash\_sum\_liquid.mosequ (using Nota: 12263: nota\_F.mosnot)
  Desc.: Sum relation of molar fractions of flash bottom output

\[
1 = \sum_{i=1}^{N_c} x_i^L
\]  
(13.60)

- Eq: 12267: flash\_phase\_equilib.mosequ (using Nota: 12263: nota\_F.mosnot)
  Desc.: General phase equilibrium.

\[
x_i^V = K_i \cdot x_i^L
\]  
(13.61)

- Eq: 12266: flash\_sum\_vapor.mosequ (using Nota: 12263: nota\_F.mosnot)
  Desc.: Sum relation of molar fractions of flash head output

\[
1 = \sum_{i=1}^{N_c} x_i^V
\]  
(13.62)

— (end of generated documentation) —
13.7. Flash sequence enc

--- (begin of generated documentation) ---

EQ System '12684: flash_sequence_enc.moseqs'

Description: 'Two flashes in a sequence.'
Notation: '12263: nota_F.mosnot'

Connected Elements

- (0) 12289: flash_raoult_antoine_b.moseqs
  Naming policy: integrate
  Used Connector: none
  External indices: none

- (1) 12289: flash_raoult_antoine_b.moseqs
  Naming policy: encapsulate
  Used Connector: 12683: con_II.moscon
  External indices: none

Connector '12683: con_II.moscon'

Subnotation: '12263: nota_F.mosnot'
Supernotation: '12263: nota_F.mosnot'
Variable matching

\[ x^V_i \rightarrow x^{LV}_i \quad \text{with} \quad | i \rightarrow i | \]
\[ D \rightarrow F \]

--- (end of generated documentation) ---
13.8. Flash ports

EQ System '12758: flash_sequence_ports.moseqs'

Description: 'A sequence of flash units that are connected via ports and streams.'
Notation: '12263: nota_F.mosnot'

Connected Elements

- (0) 12755: basic_flash_ports.moseqs
  Naming policy: streams
  Used Connector: none
  External indices: none

- (1) 12755: basic_flash_ports.moseqs
  Naming policy: streams
  Used Connector: none
  External indices: none

Internal Streams

- Stream 0
  Interface: 12747: itfc_port_F_simple.mosint
    - Port I:
      Connected element ID: 0
      Port ID at connected element: 2
      Connector: none
    - Port II:
      Connected element ID: 1
      Port ID at connected element: 1
      Connector: none

— (end of generated documentation) —
13.9. Flash sequence ports

--- (begin of generated documentation) ---

**EQ System 'example/flash_sequence_ports.moseqs'**

**Notation:** 'example/nota_F.mosnot'

**Connected Elements**

- (0) example/basic_flash_ports.moseqs
  Naming policy: streams
  Used Connector: none
  External indices: none

- (1) example/basic_flash_ports.moseqs
  Naming policy: streams
  Used Connector: none
  External indices: none

**Internal Streams**

- Stream 0
  Interface: example/props/itfc_prt_F.mosint

  - Port I:
    Connected element ID: 0
    Port ID at connected element: 2
    Connector: none

  - Port II:
    Connected element ID: 1
    Port ID at connected element: 1
    Connector: none

**Interface 'example/props/itfc_prt_F.mosint'**

**Notation:** 'example/nota_F.mosnot'

**Variable list**

\[ T \]
\[ P \]
\[ F \]
\[ x_i \]

--- (end of generated documentation) ---
13.10. Flash including heat balance

EQ System '12768: flash_steady_h.moseqs'

Description: 'Steady-state flash including heat balance with source. Calculation of phase equilibrium constant and enthalpies not specified.'

Notation: '12263: nota_F.mosnot'

Connected Elements

• (0) 12767: flash_conserv_steady.moseqs
  Naming policy: integrate
  Used Connector: none
  External indices: none

• (1) 12268: flash.constitutive_plain.moseqs
  Naming policy: integrate
  Used Connector: none
  External indices: none

Hierarchical view of contained equations:

```
Equation System: 12768: flash_steady_h.moseqs

Description: Steady-state flash including heat balance with source. Calculation of phase equilibrium constant and enthalpies not specified.

Connected EQ-Systems:

• (1) 12268: flash.constitutive_plain.moseqs
• (0) 12767: flash_conserv_steady.moseqs
```

Connection Level (1) – EQ-Systems connected to 12768: flash_steady_h.moseqs:
13.10. Flash including heat balance

Equation System: 12268: flash_constitutive_plain.moseqs

Description: Constitutive equations of a flash unit. Contains no summation relation for the feed molar fraction, so all must molar fractions of feed must be design values.

Connected Equations:
- Eq: 12267: flash_phase_equilib.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: General phase equilibrium.
  \[ x_i^V = K_i \cdot x_i^L \] \hspace{1cm} (13.63)
- Eq: 12265: flash_sum_liquid.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Sum relation of molar fractions of flash bottom output.
  \[ 1 = \sum_{i=1}^{N_C} x_i^L \] \hspace{1cm} (13.64)
  Desc.: Sum relation of molar fractions of flash head output.
  \[ 1 = \sum_{i=1}^{N_C} x_i^V \] \hspace{1cm} (13.65)

Equation System: 12767: flash_conserv_steady.moseqs

Description: Conservation equations for a steady-state flash model.

Connected Equations:
- Eq: 12264: comp_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Steady-state component balance of a simple flash.
  \[ x_i^{LV} \cdot F = x_i^L \cdot B + x_i^V \cdot D \] \hspace{1cm} (13.66)
- Eq: 12765: heat_bal_flash_steady.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Heat balance for a flash.
  \[ (10)^{(-6)} \cdot (h^F \cdot F + (10)^{(6)} \cdot Q_{im}) = (10)^{(-6)} \cdot (h^L \cdot B + h^V \cdot D) \] \hspace{1cm} (13.67)
13.10. Flash including heat balance

— (end of generated documentation) —
13.11. Distillation column

EQ System '13069: column_steady.moseqs’

Description: 'a column created from the units reboiler (represented by a flash with heat input) and a separation tower. the units are connected with each other using streams.'

Notation: '12771: nota_S.mosnot'

Connected Elements

- (0) 12799: tower_cond_std_ports.moseqs
  Naming policy: streams
  Used Connector: none
  External indices: none

- (1) 13062: flash_stdy_non-id_ports.moseqs
  Naming policy: streams
  Used Connector: none
  External indices: none

Internal Streams

- Stream 0
  Interface: 12796: itfc_streams.mosint

  - Port I:
    Connected element ID: 1
    Port ID at connected element: 2
    Connector: 13070: reboiler_to_stream.moscon

  - Port II:
    Connected element ID: 0
    Port ID at connected element: 1
    Connector: none

- Stream 1
  Interface: 12796: itfc_streams.mosint

  - Port I:
    Connected element ID: 0
    Port ID at connected element: 2
    Connector: none
13.11. Distillation column

- Port II:
  Connected element ID: 1
  Port ID at connected element: 1
  Connector: 13070: reboiler_to_stream.moscon

Hierarchical view of contained equations:

**Equation System: 13069: column_steady.moseqs**

Description: Separation tower including condenser but without reboiler.

Connected EQ-Systems:

- (0)12799: tower_cond_std_ports.moseqs
- (1)13062: flash_stdy_non-id_ports.moseqs

Connection Level (1) – EQ-Systems connected to 13069: column_steady.moseqs:

**Equation System: 12799: tower_cond_std_ports.moseqs**

Description: Separation tower equipped with ports for a reboiler unit.

Connected EQ-Systems:

- (0)12793: tower_cond_steady.moseqs

Connection Level (2) – EQ-Systems connected to 12799: tower_cond_std_ports.moseqs:

**Equation System: 12793: tower_cond_steady.moseqs**
Description: a distillation column and a separation tower. The units are connected with each other using streams.

Connected EQ-Systems:

- (1)12792: total cond steady.moseqs
- (0)12786: stage steady.moseqs

Connection Level (3) – EQ-Systems connected to 12793: tower cond steady.moseqs:

**Equation System: 12792: total cond steady.moseqs**

Description: Equations that represent total condensation at the head of a distillation column. Steady state model.

Connected Equations:

- Eq: 12788: mass bal total cond.mosequ (using Nota: 12771: nota S.mosnot)
  Desc.: mass balance for total condenser of a column
  \[ V_j = N_s = L_j = N_{s+1} + D \] (13.68)

- Eq: 12787: temper total cond.mosequ (using Nota: 12771: nota S.mosnot)
  Desc.: column head temperature equals condenser temperature
  \[ T_j = N_{s+1} = T_j = N_s \] (13.69)

- Eq: 12791: mole frac total cond.mosequ (using Nota: 12771: nota S.mosnot)
  Desc.: molar fraction for total condensation
  \[ y_j = N_{s,i} = x_j = N_{s+1,i} \] (13.70)

- Eq: 12790: heat total cond.mosequ (using Nota: 12771: nota S.mosnot)
  Desc.: heat balance for total condensation at the heat of a column
  \[ (10)^{(-6)} \cdot (h_j^V \cdot V_j = 8) = (10)^{(-6)} \cdot (h_j^L \cdot (D + L_j = 9) + (10)^{(6)} \cdot Q_{in}^{(3.71)}) \]
13.11. Distillation column

Equation System: 12786: stage_steady.moseqs

Description: Model of a separation stage of a distillation column. Equilibrium model in the steady state.

Connected Equations:
- Eq: 12773: ph_eq_k_gamma.mosequ (using Nota: 12771: nota_S.mosnot)
  Desc.: phase equilibrium constant specification. simplified gamma-phi concept.

\[ K_{j,i}^{L,V} = \frac{p_{o,j,i} \cdot \gamma_{j,i}}{p_{j}} \quad (13.72) \]

Connected EQ-Systems:
- (0)12785: stage_conserv_steady.moseqs
- (1)12781: stage_constitutive.moseqs

Connection Level (4) – EQ-Systems connected to 12786: stage_steady.moseqs:

Equation System: 12785: stage_conserv_steady.moseqs

Description: Conservation equations of a column stage equilibrium model in the steady state.

Connected Equations:
- Eq: 12784: stage_comps_steady.mosequ (using Nota: 12771: nota_S.mosnot)
  Desc.: steady-state component balance of a distillation column separation stage.

\[ F_j \cdot z_{j,i} + L_j \cdot x_{j,i} + V_j \cdot y_{j,i} = L_j \cdot x_{j,i} + V_j \cdot y_{j,i} \quad (13.73) \]

- Eq: 12783: stage_energy_steady.mosequ (using Nota: 12771: nota_S.mosnot)
  Desc.: energy balance column stage steady state. Exchange over column wall is neglected.

\[ h_{j}^{L,V} \cdot F_j + h_{j}^{L} \cdot L_j + h_{j}^{V} \cdot V_j = h_{j}^{L} \cdot L_j + h_{j}^{V} \cdot V_j \quad (13.74) \]
Equation System: 12781: stage_constitutive.moseqs

Description: constitutive equations for a column stage equilibrium model.

Connected Equations:

- Eq: 12774: sum_liq_mf_col_stage.mosequ (using Nota: 12771: nota_S.mosnot)
  Desc.: summation equation of liquid molar fractions of a column stage
  \[
  \sum_{i=1}^{N_c} x_{j,i} = 1 \quad (13.75)
  \]

- Eq: 12776: sum_vap_mf_col_stage.mosequ (using Nota: 12771: nota_S.mosnot)
  Desc.: summation equation of vapor molar fractions of a column stage
  \[
  \sum_{i=1}^{N_c} y_{j,i} = 1 \quad (13.76)
  \]

- Eq: 12779: ph_eq_col_sate.mosequ (using Nota: 12771: nota_S.mosnot)
  Desc.: phase equilibrium of a column stage
  \[
  y_{j,i} = K_{j,i}^{L,V} \cdot x_{j,i} \quad (13.77)
  \]

Equation System: 13062: flash_stdy_non-id_ports.moseqs

Description:

Connected EQ-Systems:

- (0)12987: flash_stdy_non-id_co.moseqs

Connection Level (2) - EQ-Systems connected to 13062: flash_stdy_non-id_ports.moseqs:

Equation System: 12987: flash_stdy_non-id_co.moseqs
13.11. Distillation column

Description:

Connected Equations:

- Eq: 12766: ph_eq_k_non-ideal.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: phase equilibrium constant specification. simplified gamma-phi concept.

\[ K_i = \frac{P_{o_i}^{L,V} \cdot \gamma_i}{P} \]  (13.78)

Connected EQ-Systems:

- (0)12768: flash_steady_h.moseqs

Connection Level (3) – EQ-Systems connected to 12987: flash_stdy_non-id_co.moseqs:

Equation System: 12768: flash_steady_h.moseqs

Description: and a separation tower. the units are connected with each other using streams.

Connected EQ-Systems:

- (1)12268: flash_constitutive_plain.moseqs
- (0)12767: flash_conserv_steady.moseqs

Connection Level (4) – EQ-Systems connected to 12768: flash_steady_h.moseqs:

Equation System: 12268: flash_constitutive_plain.moseqs

Description: Constitutive equations of a flash unit. Contains no summation relation for the feed molar fraction, so all must molar fractions of feed must be
13.11. Distillation column

design values

Connected Equations:

- Eq: 12265: flash_sum_liquid.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Sum relation of molar fractions of flash bottom output

\[ 1 = \sum_{i=1}^{Nc} x_i^L \]  

(13.79)

- Eq: 12267: flash_phase_equilib.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: General phase equilibrium.

\[ x_i^V = K_i \cdot x_i^L \]  

(13.80)

  Desc.: Sum relation of molar fractions of flash head output

\[ 1 = \sum_{i=1}^{Nc} x_i^V \]  

(13.81)

Equation System: 12767: flash_conserv_steady.moseqs

Description: Conservation equations for a steady-state flash model.

Connected Equations:

- Eq: 12264: comp_bal_flashsteady.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: Steady-state component balance of a simple flash.

\[ x_i^{L,V} \cdot F = x_i^L \cdot B + x_i^V \cdot D \]  

(13.82)

- Eq: 12765: heat_bal_flashsteady.mosequ (using Nota: 12263: nota_F.mosnot)
  Desc.: heat balance for a flash

\[ (10)^{(-6)} \cdot (h^F \cdot F + (10)^{(6)} \cdot Q^{in}) = (10)^{(-6)} \cdot (h^L \cdot B + h^V \cdot D) \]  

(13.83)

Interface '12796: itfc_streams.mosint'

Notation: '12771: nota_S.mosnot'
Variable list

\( p_s \)
\( T_s \)
\( z_{s,i} \)
\( F_s \)

Connector '13070: reboiler_to_stream.moscon'

Subnotation: '12263: nota_F.mosnot'
Supernotation: '12771: nota_S.mosnot'

Variable matching

\( F \to F_s \)
\( x_i \to z_{s,i} \text{ with } |i \to i| \)
\( T \to T_s \)
\( P \to p_s \)

Notation '12263: nota_F.mosnot'

Base line symbols

\( B \) bottom output stream [mol/s]
\( D \) head output stream [mol/s]
\( F \) material stream [mol/s]
\( K \) equilibrium constant [-]
\( P \) pressure [Pa]
\( Q \) heat input or output [MJ/s]
\( T \) temperature [K]
\( \gamma \) activity coefficient [-]
\( h \) enthalpy [J/mol]
\( x \) mole fraction [mol/mol]

Superscripts

\( F \) feed
\( L \) liquid
\( V \) vapor
\( in \) input

Subscripts

\( o \) reference conditions
\( sys \) system
Indices

\[ i \in 1..N_c \quad \text{compound index} \]

**Notation '12771: nota_S.mosnot’**

**Base line symbols**

\[ D \quad \text{distillate output stream [mol/s]} \]
\[ F \quad \text{overall molar stream [mol/s]} \]
\[ K \quad \text{equilibrium constant} \]
\[ L \quad \text{liquid molar stream [mol/s]} \]
\[ Q \quad \text{heat stream [MJ/s]} \]
\[ T \quad \text{temperature [K]} \]
\[ V \quad \text{vapor molar stream [mol/s]} \]
\[ \gamma \quad \text{activity coefficient} \]
\[ h \quad \text{molar enthalpy [J/mol]} \]
\[ p \quad \text{pressure [Pa]} \]
\[ x \quad \text{liquid molar fraction [mol/mol]} \]
\[ y \quad \text{vapor molar fraction [mol/mol]} \]
\[ z \quad \text{overall molar fraction [mol/mol]} \]

**Superscripts**

\[ F \quad \text{feed} \]
\[ L \quad \text{liquid} \]
\[ V \quad \text{vapor} \]
\[ \text{out} \quad \text{output} \]

**Subscripts**

\[ o \quad \text{reference} \]
\[ s \quad \text{stream} \]

Indices

\[ i \in 1..N_c \quad \text{Number of components} \]
\[ j \in 1..N_s \quad \text{Number of separation stages} \]
13.12. Generated program code for selected example cases

13.12.1. Simple Flash

Please note that the code relevant for the communication between the Applet and the Server is hidden for security reasons.

C++

```cpp
#include "BzzMath.hpp"

/* ********************************************************
 * The names have been normalized. The following
 * table shows the attribution.
 * Normalized Name --> Original Name
 * =================================
 * e0 --> e[0]12270
 * ******************************************************** */

/* Define and implement direct functions
 * */

void store_results (BzzNonLinearSystemObject *nls);

/* Declare and define the class for the Equation System
 * object */

class EquationSystem : public BzzMyNonLinearSystemObject {
  private:
    BzzVector params;
  public:
  EquationSystem (BzzVector &parv) {
    params = parv;
  }
  virtual void GetResiduals (BzzVector &x, BzzVector &f);
  virtual void ObjectBzzPrint (void);
};

void EquationSystem :: ObjectBzzPrint (void) {
  BzzPrint("Object Print for Numerical Jacobian");
  params.BzzPrint("Data");
}

void EquationSystem :: GetResiduals (BzzVector &x, BzzVector &f) {
  /* read out parameters */
  double e0_F = params[1];
  double e0_K_i1 = params[2];
  double e0_K_i2 = params[3];
  double e0_x_LV_i1 = params[4];
  double e0_x_LV_i2 = params[5];

  /* read out variables */
  double e0_B = x[1];
  double e0_D = x[2];
  double e0_x_L_i1 = x[3];
  double e0_x_V_i1 = x[4];
  double e0_x_L_i2 = x[5];
  double e0_x_V_i2 = x[6];

  /* perform direct function calls */

  /* evaluate the function values */
  f[1] = e0_x_LV_i1 * e0_F - (e0_x_L_i1 * e0_B + e0_x_V_i1 * e0_D);
  f[2] = e0_x_LV_i2 * e0_F - (e0_x_L_i2 * e0_B + e0_x_V_i2 * e0_D);
  f[3] = e0_x_V_i1 - (e0_K_i1 * e0_x_L_i1);
  f[4] = e0_x_V_i2 - (e0_K_i2 * e0_x_L_i2);
  f[5] = 1.0 - (e0_x_L_i1 + e0_x_L_i2);
  f[6] = 1.0 - (e0_x_V_i1 + e0_x_V_i2);
}

void solve_equation_system () {
}
```

```cpp
#include "BzzMath.hpp"

/* ********************************************************
 * The names have been normalized. The following
 * table shows the attribution.
 * Normalized Name --> Original Name
 * =================================
 * e0 --> e[0]12270
 * ******************************************************** */

/* Define and implement direct functions
 * */

void store_results (BzzNonLinearSystemObject *nls);

/* Declare and define the class for the Equation System
 * object */

class EquationSystem : public BzzMyNonLinearSystemObject {
  private:
    BzzVector params;
  public:
  EquationSystem (BzzVector &parv) {
    params = parv;
  }
  virtual void GetResiduals (BzzVector &x, BzzVector &f);
  virtual void ObjectBzzPrint (void);
};

void EquationSystem :: ObjectBzzPrint (void) {
  BzzPrint("Object Print for Numerical Jacobian");
  params.BzzPrint("Data");
}

void EquationSystem :: GetResiduals (BzzVector &x, BzzVector &f) {
  /* read out parameters */
  double e0_F = params[1];
  double e0_K_i1 = params[2];
  double e0_K_i2 = params[3];
  double e0_x_LV_i1 = params[4];
  double e0_x_LV_i2 = params[5];

  /* read out variables */
  double e0_B = x[1];
  double e0_D = x[2];
  double e0_x_L_i1 = x[3];
  double e0_x_V_i1 = x[4];
  double e0_x_L_i2 = x[5];
  double e0_x_V_i2 = x[6];

  /* perform direct function calls */

  /* evaluate the function values */
  f[1] = e0_x_LV_i1 * e0_F - (e0_x_L_i1 * e0_B + e0_x_V_i1 * e0_D);
  f[2] = e0_x_LV_i2 * e0_F - (e0_x_L_i2 * e0_B + e0_x_V_i2 * e0_D);
  f[3] = e0_x_V_i1 - (e0_K_i1 * e0_x_L_i1);
  f[4] = e0_x_V_i2 - (e0_K_i2 * e0_x_L_i2);
  f[5] = 1.0 - (e0_x_L_i1 + e0_x_L_i2);
  f[6] = 1.0 - (e0_x_V_i1 + e0_x_V_i2);
}

void solve_equation_system () {
```
/* Parameters */
BzzVector params(5,
/* e0_F = */ 50.0,
/* e0_K_i1 = */ 1.7,
/* e0_K_i2 = */ 0.5,
/* e0_x_LV_i1 = */ 0.5,
/* e0_x_LV_i2 = */ 0.5);

/* Guess values */
BzzVector x0(6,
/* e0_B = */ 25.0,
/* e0_D = */ 25.0,
/* e0_x_L_i1 = */ 0.5,
/* e0_x_V_i1 = */ 0.5,
/* e0_x_L_i2 = */ 0.5,
/* e0_x_V_i2 = */ 0.5);

/* Lower bounds */
BzzVector xMin(6,
/* e0_B .lb = */ -1.0E9,
/* e0_D .lb = */ -1.0E9,
/* e0_x_L_i1 .lb = */ -1.0E9,
/* e0_x_V_i1 .lb = */ -1.0E9,
/* e0_x_L_i2 .lb = */ -1.0E9,
/* e0_x_V_i2 .lb = */ -1.0E9);

/* Upper bounds */
BzzVector xMax(6,
/* e0_B .ub = */ 1.0E9,
/* e0_D .ub = */ 1.0E9,
/* e0_x_L_i1 .ub = */ 1.0E9,
/* e0_x_V_i1 .ub = */ 1.0E9,
/* e0_x_L_i2 .ub = */ 1.0E9,
/* e0_x_V_i2 .ub = */ 1.0E9);

EquationSystem eqSystem(params);
BzzNonLinearSystemObject solverObj(x0, &eqSystem);
solverObj.SetTolerance(1e-10,1e-6);
solverObj.SetConstraints(xMin,xMax);
solverObj.SetMaxNewtonCallsNumber(1e5);
solverObj();
bzzFilePrint("2_solver_out.txt");
solverObj.BzzPrint("Results");
store_results(&solverObj);
}

int main() {
RemoveWarningWindow();
solve_equation_system();
return(0);
}

void store_results ( BzzNonLinearSystemObject * nls ) {
// ... code for storage ... }
function [ROOTS] = solveEquationSystem()
% load variable init values
X_ITER(1) = 25.0; % e0_B
X_ITER(2) = 25.0; % e0_D
X_ITER(3) = 0.5; % e0_x_L_i1
X_ITER(4) = 0.5; % e0_x_V_i1
X_ITER(5) = 0.5; % e0_x_L_i2
X_ITER(6) = 0.5; % e0_x_V_i2

% load parameters
PARAMS(1) = 50.0; % e0_F
PARAMS(2) = 0.5; % e0_x_LV_i2
PARAMS(3) = 0.5; % e0_x_LV_i1
PARAMS(4) = 0.5; % e0_K_i2
PARAMS(5) = 1.7; % e0_K_i1

options = optimset ('MaxIter',1000,'TolFun',1e-10,'Display','Iter');
RES = fsolve (@( x_iter ) getFunVal(x_iter,PARAMS),X_ITER,options);

%***********************************************************************
% function[Y] = getFunVal(X_ITER,PARAMS)
% % Calculate the function value of a normalized equation system.
% % read out variables
% e0_x_LV_i2 = PARAMS(2);
% e0_x_LV_i1 = PARAMS(3);
% e0_K_i2 = PARAMS(4);
% e0_K_i1 = PARAMS(5);
% % read out parameters
% e0_F = PARAMS(1);
% e0_x_V_i2 = e0_F - (e0_x_L_i2 * e0_B + e0_x_V_i2 * e0_D);
% e0_V_i1 = e0_x_V_i1 - (e0_K_i1 * e0_x_L_i1);
% e0_V_i2 = e0_x_V_i2 - (e0_K_i2 * e0_x_L_i2);
% options = optimset ('MaxIter',1000,'TolFun',1e-10,'Display','Iter');
% RES = fsolve (@( x_iter ) getFunVal(x_iter,PARAMS),X_ITER,options);
%***********************************************************************

function [Y] = getFunVal ( X_ITER , PARAMS )
% Calculate the function value of a normalized equation system.
% read out variables
e0_B = X_ITER(1);
e0_D = X_ITER(2);
e0_x_L_i1 = X_ITER(3);
e0_x_V_i1 = X_ITER(4);
e0_x_L_i2 = X_ITER(5);
e0_x_V_i2 = X_ITER(6);
% read out parameters
e0_F = PARAMS(1);
e0_x_LV_i2 = PARAMS(2);
e0_x_LV_i1 = PARAMS(3);
e0_K_i2 = PARAMS(4);
e0_K_i1 = PARAMS(5);
% perform direct function calls
Y(1) = e0_x_LV_i1 * e0_F - (e0_x_L_i1 * e0_B + e0_x_V_i1 * e0_D);
Y(2) = e0_x_LV_i2 * e0_F - (e0_x_L_i2 * e0_B + e0_x_V_i2 * e0_D);
Y(3) = e0_x_V_i1 - (e0_K_i1 * e0_x_L_i1);
Y(4) = e0_x_V_i2 - (e0_K_i2 * e0_x_L_i2);
Y(5) = 1.0 - ((e0_x_L_i1 + e0_x_L_i2));
Y(6) = 1.0 - ((e0_x_V_i1 + e0_x_V_i2));

function [] = displayResults(X_ITER)
13.12. Generated program code for selected example cases

13.12.2. Use of functions

Target language supports function calls

Code for Matlab

```matlab
% print variable values to display
disp(['e0_B ', num2str(X_ITER(1))]);
disp(['e0_D ', num2str(X_ITER(2))]);
disp(['e0_x_L_i1 ', num2str(X_ITER(3))]);
disp(['e0_x_V_i1 ', num2str(X_ITER(4))]);
disp(['e0_x_L_i2 ', num2str(X_ITER(5))]);
disp(['e0_x_V_i2 ', num2str(X_ITER(6))]);
end
```

% *********************************************************
% The name spaces have been normalized. The following table shows the attribution.
% Normalized Name --> Original Name
% *********************************************************
% e0 --> e [0]
% f0 --> f [0]
% *********************************************************
% The variables are named according to the notation provided in the Mosaic model.
% The variable names can be read as follows:
% **************************************************
% e0_B
% E: bottom output stream [mol/s]
% e0_D
% D: head output stream [mol/s]
% e0_x_LV_i
% x: mole fraction [mol/mol]
% Superscripts
% L: liquid
% V: vapor
% Indices
% i: compound index
% *********************************************************
% e0_x_L_i
% x: mole fraction [mol/mol]
% Superscripts
% L: liquid
% Indices
% i: compound index
% *********************************************************
% e0_x_V_i
% x: mole fraction [mol/mol]
% Superscripts
% V: vapor
% Indices
% i: compound index
% *********************************************************
% f0_a_l
% a: dippr parameter
% Indices
% l: Number of components
% f0_b_l
% b: dippr parameter
% Indices
% l: Number of components
% f0_c_l
% c: dippr parameter
% Indices
% l: Number of components
% f0_d_l
% d: dippr parameter
% Indices
% l: Number of components
% e0_F
% F: material stream [mol/s]
% e0_K_i
% K: equilibrium constant [-]
% Indices
% i: compound index
% e0_P_sys
% P: pressure [Pa]
% Subscript
% sys: system
% e0_T
% T: temperature [K]

% *********************************************************

function [ROOTS] = solveEquationSystem()

% load variable init values
X_ITER (1) = 25.0; % e0_B
X_ITER (2) = 25.0; % e0_D
X_ITER (3) = 1.0; % e0_K_i1
X_ITER (4) = 1.0; % e0_K_i2
X_ITER (5) = 0.5; % e0_x_L_i1
X_ITER (6) = 0.5; % e0_x_V_i1
X_ITER (7) = 0.5; % e0_x_L_i2
X_ITER (8) = 0.5; % e0_x_V_i2

% load parameters
PARAMS (1) = 2.0; % f0_e_l2
PARAMS (2) = 2.0; % f0_e_l1
PARAMS (3) = 7.1926E-6; % f0_d_l1
PARAMS (4) = 0.5; % e0_x_LV_i1
PARAMS (5) = 4.046E-6; % f0_d_l2
PARAMS (6) = -7.1385; % f0_c_l2
PARAMS (7) = 353.0; % e0_T
PARAMS (8) = -8.7078; % f0_c_l1
PARAMS (9) = 101325.0; % e0_P_sys
PARAMS (10) = -7206.7; % f0_b_l2
PARAMS (11) = -6876.0; % f0_b_l1
PARAMS (12) = 50.0; % e0_F
PARAMS (13) = 72.55; % f0_a_l2
PARAMS (14) = 81.768; % f0_a_l1
PARAMS (15) = 0.5; % e0_x_LV_i2

options = optimset('MaxIter',1000,'TolFun',1e-10,'Display','Iter');
RES = fsolve (@(x_iter) getFunVal(x_iter,PARAMS),X_ITER,options);

ROOTS = getFunVal(RES,PARAMS);
ROOTS = ROOTS';
displayResults(RES);
end

function[Y] = getFunVal(X_ITER,PARAMS)

% calculate the function value of a normalized equation system.

% read out variables
e0_B = X_ITER(1);
e0_D = X_ITER(2);
e0_K_i1 = X_ITER(3);
e0_K_i2 = X_ITER(4);
e0_x_L_i1 = X_ITER(5);
e0_x_V_i1 = X_ITER(6);
e0_x_L_i2 = X_ITER(7);
e0_x_V_i2 = X_ITER(8);

% read out parameters
f0_e_l2 = PARAMS(1);
f0_e_l1 = PARAMS(2);
f0_d_l1 = PARAMS(3);
e0_x_LV_i1 = PARAMS(4);
f0_d_l2 = PARAMS(5);
f0_c_l2 = PARAMS(6);
e0_T = PARAMS(7);
f0_c_l1 = PARAMS(8);
e0_P_sys = PARAMS(9);
f0_b_l2 = PARAMS(10);
f0_b_l1 = PARAMS(11);
e0_F = PARAMS(12);
f0_a_l2 = PARAMS(13);
f0_a_l1 = PARAMS(14);
e0_x_LV_i2 = PARAMS(15);

% perform direct function calls
f0_e_l2 = getFunVal(RES,PARAMS)

% display results

end
13.12. Generated program code for selected example cases

13.12.1

```matlab
161 e0_P_LV_o_i2 = fun_12673__plv_fun_dippr(e0_T,f0_e_l2,
   f0_a_12,f0_d_12,f0_c_12,f0_b_12);
162
163 % evaluate the function values
164 Y (1) = e0_x_LV_i1 * e0_F - ( e0_x_L_i1 * e0_B + e0_x_V_i1 * e0_D );
165 Y (2) = e0_x_LV_i2 * e0_F - ( e0_x_L_i2 * e0_B + e0_x_V_i2 * e0_D );
166 Y (3) = e0_K_i1 - ((e0_P_LV_o_i1 + e0_x_L_i1));
167 Y (4) = e0_x_V_i1 - ( e0_K_i1 * e0_x_L_i1 );
168 Y (5) = 1.0 - ((e0_x_L_i1 + e0_x_L_i2));
169 Y (7) = e0_K_i1 - ((e0_P_LV_o_i1)/(e0_P_sys));
170 Y (8) = e0_K_i2 - ((e0_P_LV_o_i2)/(e0_P_sys));
171
172 % target language does not support function calls
173 % code for Aspen Custom Modeler:
174
175 model mosaic_model
176 
178 function [std_v] = fun_12673__plv_fun_dippr(std_greek_theta,
179 std_e,std_a,std_d,std_c,std_b)
180 std_v = exp ( std_a + ( std_b)/( std_greek_theta ) + std_c * loge ( std_greek_theta ) + std_d * power (( std_greek_theta ), std_e ));
181 end
182
183 function [] = displayResults(X_ITER)
184 % print variable values to display
185 disp(['e0_B ', num2str(X_ITER(1))]);
186 disp(['e0_D ', num2str(X_ITER(2))]);
187 disp(['e0_K_i1 ', num2str(X_ITER(3))]);
188 disp(['e0_K_i2 ', num2str(X_ITER(4))]);
189 disp(['e0_x_L_i1 ', num2str(X_ITER(5))]);
190 disp(['e0_x_L_i2 ', num2str(X_ITER(6))]);
191 disp(['e0_x_V_i1 ', num2str(X_ITER(7))]);
192 disp(['e0_x_V_i2 ', num2str(X_ITER(8))]);
193
194 end
```

13.12.1.1

```matlab
1 model mosaic_model
2 
4 // parameters
5 f0_e_l2 as NoType (2.0);
6 f0_e_l1 as NoType (2.0);
7 e0_P_LV_o_i2 as NoType (0.0);
8 f0_d_12 as NoType (4.046E-6);
9 e0_T as NoType (353.0);
10 f0_d_11 as NoType (7.1926E-6);
11 f0_c_11 as NoType (-8.7078);
12 e0_P_sys as NoType (101325.0);
13 f0_c_12 as NoType (-7.1385);
14 f0_b_11 as NoType (-7206.7);
15 f0_b_12 as NoType (-6876.0);
16 f0_a_12 as NoType (72.55);
17 f0_a_11 as NoType (81.766);
18 e0_F as NoType (50.0);
19 e0_x_LV_i2 as NoType (0.5);
20 
21 // variables
22 e0_P_LV_o_i2 as NoType (0.0);
23 e0_P_LV_o_i1 as NoType (0.0);
24 e0_K_i2 as NoType (1.0);
25 e0_K_i1 as NoType (1.0);
26 e0_x_V_i2 as NoType (0.5);
27 e0_x_L_i2 as NoType (0.5);
28 e0_D as NoType (25.0);
29 e0_B as NoType (25.0);
30 e0_x_V_i1 as NoType (0.5);
31 e0_x_L_i1 as NoType (0.5);
32 
33 // equations
34 0 = e0_x_LV_i1*e0_F-(e0_x_L_i1*e0_B+e0_x_V_i1*e0_D) ;
35 0 = e0_x_LV_i2*e0_F-(e0_x_L_i2*e0_B+e0_x_V_i2*e0_D) ;
36 0 = e0_x_V_i1-(e0_K_i1*e0_x_L_i1) ;
37 0 = e0_x_V_i2-(e0_K_i2*e0_x_L_i2) ;
38 0 = 1.0-((e0_x_L_i1+e0_x_L_i2)) ;
39 0 = 1.0-((e0_P_LV_o_i1)/(e0_P_sys)) ;
40 0 = e0_K_i2-((e0_P_LV_o_i2)/(e0_P_sys)) ;
41 0 = exp(f0_a_11*(f0_b_11)/(e0_T)+f0_c_11+log(e0_T)+f0_d_11
   *((e0_T)^*(f0_e_l1))-(e0_P_LV_o_i1)) ;
42 0 = exp(f0_a_12*(f0_b_12)/(e0_T)+f0_c_12+log(e0_T)+f0_d_12
   *((e0_T)^*(f0_e_l2))-(e0_P_LV_o_i2)) ;
```
13.12. Generated program code for selected example cases

```matlab
// fix the parameters
f0_e_l2 : fixed;
f0_e_l1 : fixed;
e0_x_LV_i1 : fixed;
f0_d_l2 : fixed;
e0_T : fixed;
f0_d_l1 : fixed;
f0_c_l1 : fixed;
e0_P_sys : fixed;
f0_c_l2 : fixed;
f0_b_l2 : fixed;
f0_b_l1 : fixed;
f0_a_l2 : fixed;
f0_a_l1 : fixed;
e0_F : fixed;
e0_x_LV_i2 : fixed;
end
```

13.12.3. Use of CAPE-OPEN interfaces

Matlab and CAPE-OPEN Thermo Import

```matlab
function [ROOTS]=solveEquationSystem()
%********************************************************
% The name spaces have been normalized. The following %
% table shows the attribution. %
% Normalized Name --> Original Name %
% ********************************************************
% e0 --> e [0] %
%********************************************************
% The variables are named according to the notation %
% provided in the Mosaic model. %
% The variable names can be read as follows: %
% ********************************************************
% e0_B %
% B: bottom output stream [mol/s]
```
% begin cape-open initialization
% ---------------------------------------------
% -> change your default settings here:
defaultManager = 'TEA (CAPE-OPEN 1.1)';
defaultPackage = 'methanol water';
% -> execute this method to start the CAPE-OPEN configuration tool.
% ---------------------------------------------
[co_package_manager, co_package] = configureCapeOpenPhysProps(
defaultManager, defaultPackage);
global co_handle;
co_handle = capeOpenGetPackage(co_package_manager, co_package);
disp('Used CAPE-OPEN configuration:');
dispCapeOpenConfiguration(co_package_manager, co_package);
% end cape-open initialization

% load variable init values
X_ITER(1) = 25.0;  % e0_B
X_ITER(2) = 25.0;  % e0_D
X_ITER(3) = 1.0;   % e0_K_i1
X_ITER(4) = 1.0;   % e0_K_i2
X_ITER(5) = 0.5;   % e0_x_L_i1
X_ITER(6) = 0.5;   % e0_x_V_i1
X_ITER(7) = 0.5;   % e0_x_L_i2
X_ITER(8) = 0.5;   % e0_x_V_i2

% load parameters
PARAMS(1) = 50.0; % e0_F
PARAMS(2) = 0.5;  % e0_x_LV_i2
PARAMS(3) = 0.5;  % e0_x_LV_i1
PARAMS(4) = 353.0; % e0_T
PARAMS(5) = 101325.0; % e0_P_sys
options = optimset('MaxIter',1000,'TolFun',1e-10,'Display',
'Iter');
RES = fzero(@(x_iter) getFunVal(x_iter,PARAMS),X_ITER,
options);
ROOTS = getFunVal(RES,PARAMS);
ROOTS = ROOTS';
displayResults(RES);
end

function [Y] = getFunVal(X_ITER,PARAMS)

% read out variables
e0_B = X_ITER(1);
e0_D = X_ITER(2);
e0_K_i1 = X_ITER(3);
e0_K_i2 = X_ITER(4);
e0_x_L_i1 = X_ITER(5);
e0_x_L_i2 = X_ITER(6);
e0_x_V_i1 = X_ITER(7);
e0_x_V_i2 = X_ITER(8);

% read out parameters
e0_F = PARAMS(1);
e0_x_LV_i2 = PARAMS(2);
e0_x_LV_i1 = PARAMS(3);
e0_T = PARAMS(4);
e0_P_sys = PARAMS(5);

% perform direct function calls
std_p_LV_lALL = fun_12679__plv_fun_cape_open(e0_T);
e0_P_LV_o_i1 = std_p_LV_lALL(1);
e0_P_LV_o_i2 = std_p_LV_lALL(2);

% evaluate the function values
Y(1) = e0_x_LV_i1 * e0_F - (e0_x_L_i1 * e0_B + e0_x_V_i1 * e0_D);
Y(2) = e0_x_LV_i2 * e0_F - (e0_x_L_i2 * e0_B + e0_x_V_i2 * e0_D);
Y(3) = (e0_x_V_i1 - (e0_K_i1 * e0_x_L_i1));
Y(4) = (e0_x_V_i2 - (e0_K_i2 * e0_x_L_i2));
Y(5) = 1.0 - (((e0_x_L_i1 + e0_x_L_i2));
Y(6) = 1.0 - (((e0_x_V_i1 + e0_x_V_i2));
Y(7) = (e0_K_i1 - ((e0_P_LV_o_i1)/(e0_P_sys)));
Y(8) = (e0_K_i2 - ((e0_P_LV_o_i2)/(e0_P_sys)));
end
function [std_p_LV_lALL] = fun_12679__plv_fun_cape_open (std_greek_theta)
    global co_handle;
    std_p_LV_lALL = capeOpenTDepProp (co_handle,'vaporPressure', std_greek_theta);
end

function [] = displayResults(X_ITER)
    % ... print variables including their name to console
    % ... code left out here for the sake of conciseness
end

% provide the cape open handles.
% allow the user to choose own configurations based
% on the CAPE-OPEN environment on the user’s machine.
function [MANAGER,PACKAGE] = configureCapeOpenPhysProps(MANAGER,PACKAGE)
    % ... calls to cape-open thermo import ...
    % ... code left out here for the sake of conciseness
end

% display the current cape open configuration
% according to the specified handles
function [] = dispCapeOpenConfiguration(MANAGER,PACKAGE)
    % ... code left out here for the sake of conciseness
end

function [OK] = testConfiguration(MANAGER,PACKAGE)
    % ... calls to cape-open thermo import ...
    % ... code left out here for the sake of conciseness
end

---

# The variables are named according to the notation provided in the Mosaic model.

# The variable names can be read as follows:
# ==================================================================================
# e0_B
# B: bottom output stream [mol/s]
# e0_D
# D: head output stream [mol/s]
# e0_x_L_i
# x: mole fraction [mol/mol] Superscripts L: liquid Indices i: compound index
# e0_x_V_i
# x: mole fraction [mol/mol] Superscripts V: vapor Indices i: compound index
# e0_x_LV_i
# x: mole fraction [mol/mol] Superscripts L: liquid V: vapor Indices i: compound index
# e0_F
# F: material stream [mol/s]
# e0_K_i
# K: equilibrium constant [-] Indices i: compound index

---

# The name spaces have been normalized. The following table shows the attribution:
# Normalized Name --> Original Name
# ==================================================================================
# e0 --> [12680]
13.12. Generated program code for selected example cases

---

DECLARE TYPE

e0_B_type = 25.0 : -1.0E9 : 1.0E9

e0_D_type = 25.0 : -1.0E9 : 1.0E9

e0_K_i1_type = 1.0 : -1.0E9 : 1.0E9

e0_K_i2_type = 1.0 : -1.0E9 : 1.0E9

e0_P_LV_o_i1_type = 0.0 : -1.0E9 : 1.0E9

END

MODEL MOSAIC_export

PARAMETER

# read out parameters

e0_F AS REAL

e0_P_sys AS REAL

e0_T AS REAL

e0_x_LV_i1 AS REAL

e0_x_LV_i2 AS REAL

e0_x_V_i1 AS REAL

e0_x_V_i2 AS REAL

SET

# set parameters

e0_F := 50.0;
e0_P_sys := 101325.0;
e0_T := 353.0;
e0_x_LV_i1 := 0.5;
e0_x_LV_i2 := 0.5;

Phys_prop := "myPropertiesFile.MFL";
NoComp := Phys_prop.NumberOfComponents;

EQUATION

# evaluate the function values

0 = e0_x_LV_i1 * e0_F - (e0_x_L_i1 * e0_B + e0_x_V_i1 * e0_D);
0 = e0_x_LV_i2 * e0_F - (e0_x_L_i2 * e0_B + e0_x_V_i2 * e0_D);
0 = e0_x_V_i1 - (e0_K_i1 * e0_x_L_i1);
0 = e0_x_V_i2 - (e0_K_i2 * e0_x_L_i2);
0 = 1.0 - ((e0_x_L_i1 + e0_x_L_i2));
0 = e0_K_i1 - ((e0_P_LV_o_i1)/(e0_P_sys));
0 = e0_K_i2 - ((e0_P_LV_o_i2)/(e0_P_sys));
# call external routines
std_p_LV_1ALL1 = Phys_prop.VapourPressure(e0_T);
e0_P_LV_o_i1 = std_p_LV_1ALL1(1);
e0_P_LV_o_i2 = std_p_LV_1ALL1(2);
13.13. Generated or imported MathML code of documentation-level expressions

The MosaicMathML expression of equation

\[ K_{L,V}^{j,i} = \gamma_{j,i} \cdot \frac{P_{o,j,i}^{L,V}}{p_j} \quad (9.1) \]

The MosaicMathML expression of equation

\[ P_{o,j}^{L,V} = (10)^{A_i - \frac{B_i}{T+c_i}} \quad (9.2) \]
The Presentation MathML expression for

\[ F_j \cdot z_{j,i} + L_{j+1} \cdot x_{j+1,i} + V_{j-1} = L_j \cdot x_{j,i} + V_j \cdot y_{j,i} \]

created with Firemath [Bon12], where the application of the elements \( mo \), \( mi \), and \( mn \) was irregular by intention:
\[ V_j \mathbin{\cdot} y_{j-1} = L_j \mathbin{\cdot} x_{j,i} + V_j \mathbin{\cdot} y_{j,i} \]
\[
\sum_{j=1}^{y} \mathbf{x}_{j,i} + \mathbf{V}_{j} \mathbf{y}_{j,i} = \mathbf{L}_{j}
\]